The Debugger™ V2

Documentation V2.9.0

343 Trenton Way
Menlo Park CA 94025

Last revised Feb 20, 1996

Steve's Comments Feb 20, 1996

The first version of this manual was done in 1988 and quite frankly some parts of it have not aged gracefully with the years.

Pages 1 - 5 have some general stuff of interest.

Pages 6 - 9 are a bit dated, but mostly truthful.

Pages 10 - 29 walk you through 'The Debugger' features using a 68K assembly language program DebugTut as the vehicle. The section on Floating point arithmetic is particularly interesting as the author of it was an expert in that area. I have left this section in for historical reasons. On the other hand, most of us do source level debugging today, and in that case, your time is better spent watching the Debugger Tutorial on the CD-ROM, or spending some time with the 'Soft-MMU Exerciser' program.

Pages 30 on are the reference section, and I have spent some time removing most of the omissions and lies from it. In many cases you will note that it refers to 'Debugger Tech Notes' or Debugger Tech Note #n or DTN #n.

All DTN's reside in the file: 'Debugger Tech Notes' on the CD-ROM in Acrobat format.

Also, one should keep in mind that this manual was written prior to the introduction of System 7, and still contains some references to a program called MultiFinder.

In System 7, MultiFinder has mutated into the 'Process Manager' which is hidden in the System file as a set of 'scod' resources that are loaded at boot time prior to launching the Finder.

The net effect on what you read in the manual is:

IN System 7, MultiFinder is aways present, and in the sense of System 6 is always the start-up application, even though the debugger start-up dialog box proclaims that Finder is the start-up.

Introduction

Credits

The Debugger was programmed by Steve Jasik, aka the Head Nose, with assistance from Harry Starr, the Crocodile Dundee of programming. The Tutorial section and DebugTut were written by John Locke.

Portions of ***The Debugger*** duplicate resources found in the Macintosh System file. That file is the copyrighted work of Apple Computer Inc.

When the documentation uses a first–person pronoun -- I, me, my, mine, etc. -- Jasik is speaking.

Copyrights

The disk is not copy protected. Proceeds from the sale of ***The Debugger*** are used to support the author and his cohorts, whose goals include making a living by creating useful programming tools.
Your purchase furthers these goals.

Your Rights and Privileges

Your purchase of ***The Debugger*** gives you a number of rights and privileges :

- You have a **"per person" License** to use the program on any computer to which you have access.
- You may obtain updates within the year (1995, 1996, …) that you purchased it in.
- You can obtain tech support (see below section).
  One request : please check the manual and the program carefully before asking for help.
- In the unlikely event that the floppy disk has **media errors**, you may return it for replacement.

Updates

The Debugger is updated to correct bugs, add features and to keep current with OS and new hardware..
Free updates are mailed out to registered owners on an irregular basis (3 to 6 months) during the first year of ownership. After that, you will be billed for continuing support, which includes updates for that year.

Registration

Owners should send in their registration so as to receive automatic updates which are sent out .

Version Date

The Debugger does not have a Version number, per se, but ***The Debugger*** has a build date which is displayed in the -Notes- window on startup, and in the title of the White labeled floppy disk.

Scope

Features and examples in this documentation apply to versions of ***The Debugger*** built after 2/18/96.

Help/Tech Support

We all like flawless programs. Reality does not always cooperate. Make me aware of any problems with ***The Debugger***; I will fix them as soon as possible. Send your feedback.

Tech support is available by phone during normal working hours (Pacific Standard Time please) or via e-mail.

Our addresses are: Internet: **macnosy@jasik.com** and Compuserv: 76004,2067

Addenda to this Manual

Check for an "Addenda" file on the release disk that you received with this manual or for separate printed sheets (Debugger Tech Notes).

Some History

At one time both a +/SE and a Universal Version of The Debugger existed. The +/SE version was discontinued in 1990, but some allusions to it may remain.
In any case features discussed in the manual always apply to the Universal version.

A Short Private Glossary

We try to stick to standard Apple and Motorola terminologies and try to minimize private jargon. What follows is a brief list of lapses in this policy. Be sure to check out the section **Program Conventions** for the private jargon of ***The Debugger***.

Δ	Stands in for an otherwise invisible space, as in the file name ΔROM.
680x0	One of the Motorola family of processors (68000, 68020, 68030 and 68040).
aux	auxiliary (as in auxiliary file, NOT A/UX)
Cmd	The **Com**man**d**/Apple/cloverleaf key.
fba	Stands for first byte address, the beginning location of a piece of code or data
IM	Inside Macintosh, our guide.
menuName:commandName	This is how we refer to a command from a menu. Example: the **Files:Open** command
OS	The Macintosh Operating System routines, as in: **OS** Trap call
PC	A 680x0 processor's Program Counter register.
PP	Short for Problem Program.
PP_id	The name of the **PP**, used to build generalized, kindred names. For example, ***PP_id***`.txt` signifies a text file associated with the PP.
Problem Program	The code you're debugging.
select x	means double click on x so as to highlight it
TB	The Macintosh Toolbox routines, as in: **TB** Trap call
tbl	table

Packing List

In your version of **The Debugger V2 and MacNosy** you should have the following items:

A CD-ROM labeled “The Debugger V2 and MacNosy”

In addition, the package **may** contain:

A floppy disk with a patch file to The Debugger, …

A cover letter

A registration slip on yellow or orange paper (NOT included if you bought it directly from me).

Version Numbering

Both the White floppy disk and The Debugger have a date on them which defines the exact version that you have.
Note that this tends to change on a weekly basis as I fix bugs or add features.
The date on which The Debugger was built is displays as a line in the “-Notes-” window.

```
( USR_SG =   BB45A0 The Debugger - Universal Ver 2.9.0 2/18/96 )
```

About The Debugger

What It Is

The Debugger is both a **assembly–level** symbolic debugger and a **source-level** debugger. A debugger lets you monitor the execution of another program -- called the **Problem Program**, or **PP** -- without affecting behavior of the PP. A **symbolic** debugger knows about the symbols of the PP. A **source–level** debugger can correlate the object code to the source code of the PP. An **assembly-level** debugger allows you to deal directly with machine instructions presented to you in mnemonic form.

Operational Features

The Debugger operates in a full multi–window Macintosh environment, it displays information about the state of the PP and the system environment in a set of text windows.

In most instances, you control ***The Debugger*** by selecting an item in a window and issuing a keyboard command. In this way, for example, you can easily set breakpoints, watchpoints, intercept trap calls, etc. You can view registers, memory, the heap, the stack, global variables, trap parameters, and other important Macintosh data structures. Anything you see displayed on a window can readily be transferred to a **-Notes-** window or saved to disk for later review.

The Fruits Of Experience

I have been programming for over 20 years. A fair amount of that time has been spent validating programs, and writing programs that aid in that work. Fairly early in my career, I found a way to quickly test and debug my programs: I wrote code that displayed intermediate output in a format that made sense to me.

Intelligent Behavior

Much of the behavior we classify as intelligent involves the recognition of patterns. MacNosy and ***The Debugger*** I try to present information about computer programs in a way that facilitates pattern recognition. Data is filtered so that the user sees only familiar Macintosh code and data structures, not raw numeric values. That is, both code and data are organized and formatted into presentations that are conducive to picking out a significant condition from the background of normal or irrelevant details.

The Debugger helps you get to that well-hidden bug by showing you everything in natural form. Variables are displayed according to their type: integers in decimal, strings as text, absolute addresses in hexadecimal, and handles in dereferenced form. Structures are shown with labeled fields containing typed values.

Why It Is

I was led to write ***The Debugger*** by my disappointment with the available Macintosh debugging tools. I wanted something that would have the power to expose the structures of the Mac OS and the increasingly sophisticated programs we are all writing. I think I have succeeded.

More About Debuggers

By definition, a debugger can monitor the execution of a PP without affecting the behavior of the PP. This requires careful coding. Good debuggers circumvent Heisenberg's Uncertainty Principal.

Debuggers typically provide a number of ways to monitor a PP. Here are a few: You can set breakpoints, locations in the code of the PP where it will stop executing and pass control over to ***The Debugger***. You can set watchpoints, areas of memory that are monitored for changes. You can view displays of the state of the PP and of the operating system, showing the values of variables and the flow of program control.

The Debugger implements watchpoints for small areas (1 to 16 bytes) via fast "compiled code". An active watchpoint slows down the PP by a factor of 2 to 4. Another way to narrow down the problem area is to attach an **action clause** to the **Trap Entry Trace** or **Proc Entry/Exit Trace** commands.

A debugger/monitor needs to protect the areas of memory it occupies from inadvertent modification by the PP and the operating system. MMU Protection as discussed in Debugger Tech Note #7 is a partial solution to the problem of catching attempts of a PP to read or write to places that it should not.

Under the Hood

Some Technical Insight

One of my objectives was to use the standard Macintosh interface. My first task was to figure out how Switcher worked. Then The Debugger could run in its own heap zone, located above BufPtr (the Last Byte Address of memory that an application may use). Using Nosy and the Switcher internal documentation, I was able to disassemble Switcher and understand its LaunchSubTask procedure. As this is a non–trivial exercise, I will leave the details for some other time. If you cannot wait, disassemble xDbgr_Startup (the application which launches The Debugger into its heap zone).

Separating The Debugger And The PP

The Debugger separates itself from the PP by the following means:

a) It runs in a separate heap zone that is above BufPtr.
b) It maintains a private copy of the System globals (0 to the beginning of System Heap).
c) It copies the vertical blank (VBL) task queue so that user VBL tasks do not interfere with it.
d) It copies the unit table so DA's don't interfere with it
e) When running in single screen mode it does a screen swap.
f) Its resource file contains copies of System resources it might use.
g) If a 6888x FPU is present it preserves the PP's copy of the FPU regs.

Some Goals

A debugger can be viewed as an extension of a Problem Program (PP). It must be able to interactively display information about PP's state, change the environment, and create opportunities for transfers of control so the user can inspect the PP's behavior.

A debugger should present information in formats that are natural to the PP and let the programmer easily recognize any underlying patterns.

A Mac debugger should take advantage of the Mac interface so a user can select an item by clicking, then issue a command that reveals the structure of the selected item. In Nosy and The Debugger this is implemented via the **Tables:Fields of** command (**Cmd–space**), “Show the structure of ...”.

A Macintosh debugger must realize that the Mac is not a mainframe. There is no clear separation between the PP and the Mac OS. Stated another way, you are "in bed with the Mac," and any false move may be dangerous to your health. The primary reasons for this situation are the complexity of the Mac interface, the lack of memory protection, and the lack of validation of parameters to Trap calls. Other than time and experience, which bring understanding, little can be done about the complexity of the Mac interface. The lack of memory protection can be solved by an add-on hardware board or MMU, but most of us will have to live without this protection for a while. The lack of validation of Trap parameters can be eliminated during the debugging phases of a program by activating Discipline.

Based on these observations and learning from earlier implementations of Trap Discipline, I created the slogan "Beyond Discipline, Into Bondage" and set out to implement it in "The Debugger" in a way that would automatically place a larger set of constraints on the PP. "The Debugger" tries to catch a larger class of errors before the program runs amuck and forces the programmer to find a suitable part of his body to scratch to stimulate mental activity.

Some of the other goals were that The Debugger should run on all of the machines with a 680x0 CPU chip (present and future) and take advantage of bigger and multiple screens as they become available.

The Art of Debugging

Unit And System Testing

Program testing may reveal flaws in your program. When the testing is done on a new section of code to test specific features, it is called Unit Testing. When it is done on the whole program by somebody else in a less controlled environment, it is called System testing. The result is usually a bug report to you in one form or another (letter bombs are no longer in vogue). Given anomalous program behavior, your task is to use the available tools to determine the cause of the problem as quickly as possible, then fix it. The author of a program should be able to locate most of its bugs in a short time (under 10 minutes), given some reasonable printouts that relate the intermediate actions of the program to its source code.

An Example From My CDC Days

As an example, when I developed and maintained the Code Generator for Compilers on the CDC 6600/7600 Systems, I had an in–house version of the compiler that was capable of printing the Instruction Sequences that the compiler was processing. When I received a bug report, I turned on some of the debugging printout, and ran the failing program through the compiler. In 90 percent of the cases this was sufficient to pinpoint the offending source code. There were one or two exceptions to this however. In particular, I remember one "off by 1" bug that took the better part of two weeks to locate.

Displaying Internal Structures

To summarize the last section, time spent in coding routines to display your internal structures will save you time and make your life much easier in the long run. Note that The Debugger contains two displays in the **Misc Menu**, **Sts Dump** and **Dbgr Tbl Dump**, which I have used to help locate a few bugs in The Debugger itself. And in the absence of a hardware emulator (cost $10K to $20K), I used one of the other currently–available debuggers to debug mine. Presently, The Debugger can debug a copy of itself!

Macintosh Programming Difficulties

Programming on the Macintosh is fraught with difficulties. The combination of:

a) A sophisticated, event–driven user interface
b) ROM code that is complicated, sometimes poorly written and documented
(lack of error checking of parameters to the trap routines, etc)
c) lack of memory protection

can make it very difficult to track down bugs. It is easy to make a simple mistake, then spend hours getting back to the source of the error. In a large mainframe system, you can run the PP in full-scale emulation mode, saving the results on disk, and, after a suitable lunch break, come back and inspect the results at leisure. Here in the micro world, we are driven by somewhat different constraints. Many of us are under time pressure to get the code out, and few of us have 5 or 10 megabytes of free disk space to hold the output of a full scale simulation (myself included).

Some Useful Strategies

Stated another way, we're looking for the least expensive way to locate the cause of a given bug. The first path of attack is to inspect the program and system state at the point of error for clues to the cause of the bug. Tools such as the **Stack Crawl**, the **System syms** display, **Heap Dumps**, and memory displays provide useful information.

If at this point you can't work back to the source of the error, then it is time to consider the cost of bringing in more potent weapons. In ascending order of performance expense, the available tools are:

a) **Trap Discipline** (slows the problem program down 10 - 20%)
b) **MMU Protection** if applicable (slows the problem program down 30%)
c) **Breakpoints**, **trap intercepts** and print statements
(minimal slowdown of the program but humans are slow to read the displays)
d) **Watchpoints** (slows program down by a factor of 2 to 4)

The **Watchpoint** feature of The Debugger helps you pinpoint problems that arise from the lack of memory protection or range checking. After the execution of every instruction, The Debugger will check a small area of memory (1 to 16 bytes) for any change. This will slow the Problem Program performance by a factor of 2 to 4. This is fast enough for most purposes. Watchpoints are an extremely useful technique for identifying the exact source of some very nasty types of problems.

Installation

Files on the Disk

There are many files on the disk containing ***The Debugger***. Files required to run ***The Debugger*** are in **boldface**; the others are optional, or are needed by MacNosy or are bonuses.
Note that some of the files or folders may shift over to the blue disk.
Here is a list of them:

In the **Dbgr/Nosy Files** folder:

The Debugger	Main code for ***The Debugger***.
ROM.dsi	Default .dsi file for the ROM. Use it to configure ***The Debugger***.
ROM/CODE.snt	ROM analysis from MacNosy that lets ***The Debugger*** be smart about the ROM

NOTE:The default ROM/CODE.snt file is now configured for the "$067C" Universal ROMs that are used in the Mac IIci, IIfx, IIsi, Quadra's and PowerBooks.

Other ROM .snt files for the Mac II, IIx, IIcx and SE/30 (256K ROMs), are in a separate folder.

NosyII	MacNosy, the Macintosh code recovery tool
sij_*xxx*	Tables used by ***The Debugger***, with *xxx* replaced by a descriptive string
ΔROM	A ghost file need by Nosy so you can select ROM to disassemble.
-D_Help-	The text file supporting the Help function of ***The Debugger***.
-Help-	The text file supporting the Help function of MacNosy.
yyy.npl	Nosy Pattern Libraries. Files that help MacNosy to name procedures.
Amacs	A text file of MPW macros to assemble code disassembled by *Nosy*

In the **put files in System fldr** folder:

xDbgr_Startup	An INIT to launch ***The Debugger*** at "INIT 31" time.
MacsBug	A program that patches _SetTrapAddress to record patches to the System.
PPCTraceEnabler	only for PowerMac's running System 7.1.2 (included in System 7.5.x)
Finder.dsi	Default .dsi file for Finder. **(System 6 only)**
MultiFinder.dsi	Default .dsi file for MultiFinder. **(System 6 only)**

In the **Tutorial** folder:

DebugTut	An application you can use to learn ***The Debugger***.
DebugTut.dsi	A '.dsi' file ***The Debugger*** uses with DebugTut. (Read it)
DebugTut.map	A map file ***The Debugger*** uses so it can work with DebugTut.
RunDbgr	an application which launches ***The Debugger*** from the Finder. **(System 6 only)**
InformDbgr	a small program to back patch the JSR to an MPW Tool. Install it in your MPW folder and run it once in you user startup file if you wish to debug MPW tools in the MPW environment.
Sample.psi	illustrates the way in which MacApp™ users may shorten the names of their procedures.
TC_Mangle	used to relocate Think C project files so that Nosy can form cross reference info.

In the **IBS V1 for MPW 3.x** folder
Contains files for the Incremental Build System. Use IBS_Install to install it.
See the IBS Version 1 Notes for details.

Installation Notes

Minimal System

You need a Macintosh with at least 1 megabyte of RAM, a 800K floppy drive, a 128K or newer ROM, and the HFS file system. ***The Debugger*** won't work on an XL or a Mac with the original 64K ROM.

If you are using System 7, then it will be difficult to run The Debugger with less than 8 Meg of RAM.

The Debugger does **NOT get along with the version of Virtual Memory implemented by Sytem 7 or 'Ram Doubler' by Connectiix** (a sneaky form of VM).

Two Ways to Start The Debugger

With System 7, ***The Debugger*** can only be started at INIT time by **xDbgr_Startup** which must be in the Extensions folder.

Feb 96 - Warning: I have not used RunDbgr in years, and I don't think it will work.

With System 6, ***The Debugger*** can be used in two ways. It can be launched at INIT 31 time by **xDbgr_Startup** and lurk in the background, or it can be launched by **RunDbgr** from the Finder or some other application such as the MPW Shell. The size of memory in your machine will determine which mode you choose as *The Debugger* needs at least 680K to operate in.
Note that you can NOT use RunDbgr when MultiFinder is the startup.

In a small machine, the use of *The Debugger* will be limited to those times when the PP is to be debugged. One can Launch *The Debugger* via **RunDbgr** from the Finder or the Shell used for program development (MPW, LightspeedC, etc) After you are finished debugging, you can use the Shutdown Debugger command to return to Shell

When ***The Debugger*** is launched at INIT 31 time by the INIT file, **xDbgr_Startup**, it lurks in the background (**Background Mode**) until activated by a System error, exception, or the launch of a task that has an associated auxiliary file.

In both modes ***The Debugger*** intercepts _Launch and various flavors of _GetResource trap calls. When such a call involves a Think C PROJ file or a file for which there is an auxiliary (**.map**, **.sym** or **.snt**) file in the same folder, ***The Debugger*** assumes you want to debug this program or code resource and it creates internal tables for it. Further, ***The Debugger*** optionally establishes a courtesy breakpoint at the first instruction of the PP.

Installation Procedure -

For the revised Installation Procedure see Debugger Tech Note #0

Special MacsBug Must Be In Place

In all modes a special file called **MacsBug** must be in place in the System folder at the time the system is booted. This file contains code for a trap-patch recording function needed by ***The Debugger***. This code is installed by the same Mac Boot code mechanism that installs Apple's debugger.

Custom–Fitting ROM/CODE.snt To Your System

Run Nosy, select the file " ROM" in the "Dbgr/Nosy Files" folder, answer all prompts with a Return. When you get to Window mode, **Cmd-W** to start a treewalk. Respond to the Treewalk dialog with a Return. When you get back to Window mode, select the **Save .snt, reRead .aci** in the **Misc** Menu. Answer NO to the dialog, and **Quit Nosy**.

32 Bit Systems

Both Nosy and *The Debugger* are "32 bit clean".

Foreign Systems and Keyboards

The use of a US Keyboard is highly recommended. A foreign keyboard (French, etc) can be used if one adjusts some of the command key equivalents. The resource "`Dcfg`" consists of two Ascii characters. the first is the swap-screen character (~) and the second is the default option-interrupt character (backslash). You may change this resource in ResEdit.
In some cases (Arabic Systems), you may need to put a copy of the 'KCHR,0' resource from a US System in ***The Debugger*** so that it work with those systems.

Multi Screen Operation (Mac II, IIx, ...)

When the "split screen" button is checked in The opening dialog, The Debugger takes over the Startup screen (the one that the Smiling Mac appears on), and splits the GD (Graphic Device) list so that the rest of the world only sees the remaining screens. You can configure the Startup screen in the Control Panel by **holding down the option key** when you click on the **Monitors** icon.

Where Things Should Live

The files in the Dbgr Files folder should stay right where they are. The PP and its **.map**, **.snt** or **.sym** file must be in the same folder. In **RunDbgr Mode**, they must be on the same volume as RunDbgr and the Dbgr Files folder. In **Background Mode**, the Dbgr Files folder must be on the **startup volume**.

Debugger Memory Requirements

When running *The Debugger* in background mode, you should give it about 1320K if MultiFinder is the startup and more (1400K) if the Finder is. When Multifinder is the startup, *The Debugger* forms its symbol tables for tasks in MultiFinder's heap, while when Finder is the startup it builds the tables in its own heap zone. As an aid to estimating how much space you will need, the Apple Icon in the menu bar has been replaced by a dynamic display of the amount of free space in The Debugger's heap zone.

In single screen mode, *The Debugger* also allocates a screen swap buffer equal to the number of pixels of the screen, over and above the space it uses for its heap zone. In the case of 1024 by 768 color monitor running in 8-bit color mode it will occupy 768K Bytes! Trying to run *The Debugger* in other than 1 bit per pixel mode on a single screen Mac II with 1 Meg of memory may cause it to blowup.

Installing The Debugger at INIT time - frequently encountered problems

This section exists because we do not have the time or money to test all possible combinations of hardware and software that a Mac may be configured with. The software that accompanies some hardware products (Radius Accelerators, Radius Displays and some hard disks) is incompatible with *The Debugger*.

The author currently uses a Quadra 950 for development and a 9500/164, 8110/80, 520C, IIfx and 840AV for testing. We no longer do testing on: Mac II's or Mac +'s (Classic).

When *The Debugger* starts up it makes a copy of the "world" (trap vectors, etc.) so that it **will not be affected by patches to the traps** that are set by programs that come after it. (like MultiFinder).
To debug INITs that come alphabetically before *xDbgr_Startup* you may rename it or the INIT.

The Debugger is not compatible with all INITs. Some of them must be loaded after *The Debugger* , or removed from the System. In particular "SuitcaseII" should be renamed to "zSuitcaseII" so that it loads after *The Debugger*.

To avoid interference between the command-option key equivalents in QuicKeys™ and those in *The Debugger*, you may wish to rename QuicKeys™ to zQuicKeys™ .

Other INITs that are known to cause *The Debugger* problems are MacroMaker, SuperLaserSpool™, Tops™ (Version 2), InBox™, and some of the INIT management programs (InitPicker Version 1, etc).

See Debugger Tech NOte #0 for a more up to date list of problem Extensions.

Radius Accelerators and displays are another set of products that have caused a number of problems due to their extensive patching of the Traps. Possible solutions are discussed in addenda sheets.

If you cannot get The Debugger to install at INIT time, then you should first try to remove **all INITs**, and reboot. If you can get it to run that way, then reintroduce the INITs one at a time until you find the one that causes problems. The other alternative is to set Finder as the startup, boot without The Debugger, and launch **RunDbgr** from the Finder with the Startup APPL set to Finder or MultiFinder.
If this procedure fails, then give me a call.

Hard Disk Hints

If you are using a hard disk, make sure you have a backup floppy to boot from, and a copy of your System file on your Hard disk in case it gets corrupted. Sometimes the Mac will not boot from a hard disk after a crash. If you boot it from the floppy, the Finder will usually be able to repair the hard disk and make it bootable again. A minimal bootable backup floppy should contain current versions of the System file and Finder.

If you have MPW, then you can use **RezDet** to check for a corrupt System file.

In extreme situations, you will need to reset the **parameter RAM** in your Mac. Here is how: On machines with removable batteries, just pull the battery, then replace it. On machines with soldered–in batteries (the SE and II) : Boot up from a floppy. In System 6, hold down **Shift–Option–Cmd** while opening the **Control Panel DA**. Click the Yes button on the warning dialog.

On system 7 the magic key combo is **option-command P and R keys during boot**

Use a hard-disk-backup utility to maintain current backups of all your hard disk files.
Try to develop good backup habits before you learn from experience.

Tutorial

Assembly-Level Debugging

By **assembly-level** debugging we mean debugging at the machine-instruction level. When we are debugging at the assembly level, we want to be able to see down to the lowest level of detail in both code and data. We need to be able to see such things as the CPU register-contents as well as the detailed sequence and location of the instructions. We may want to know how long it takes for a routine to execute. But, while we are certainly interested in being able to zero in on fine points, it is important to be able to view the program and data structures in **symbolic** form so that we do not become distracted by trivial detail

You are about to see that ***The Debugger*** has powerful easy-to-use features that reveal code and data in an entirely natural and readable **symbolic** form. That is, data and code are displayed to reflect the data types and the data and code structures that are fundamental to applications and code resources designed for the Macintosh environment. Perhaps equally as important, ***The Debugger*** will also provide a structured display of the system environment: globals, heaps, records and much else.

The Debugger also provides discipline checks on the parameters of most trap calls and provides many mechanisms for testing for combinations of conditions that may be associated with a specific problem you are investigating.

Further, by means of ***The Debugger***'s multiple windows you are able to display a great many things on the screen simultaneously and hence accelerate your rate of discovery of bugs and problems. You can also easily save anything from any window at any time to files so that you can review your observations later.

Because ***The Debugger*** is appropriately rich in features and the Macintosh environment is rather complex, you may find it easier to get started using some basic features in a fairly simple environment before you begin to apply the full range of features to more difficult problem programs. This **tutorial** section will introduce you to many of the features of ***The Debugger*** by making it easy for you to trace through some actual code while trying many of the tools ***The Debugger*** has to offer.

Getting Acquainted With *DebugTut*

Our vehicle for getting started with assembly-level debugging is an **application** program called "***DebugTut***". ***DebugTut*** has features in common with many applications programs. It is structured around a conventional event-loop, has a dialog window with some controls, has a second graphics-output window, can write text and graphics to either window and can run in background under MultiFinder. ***DebugTut*** is unusual only in that its function is to allow the user to conveniently execute code under control of ***The Debugger***.

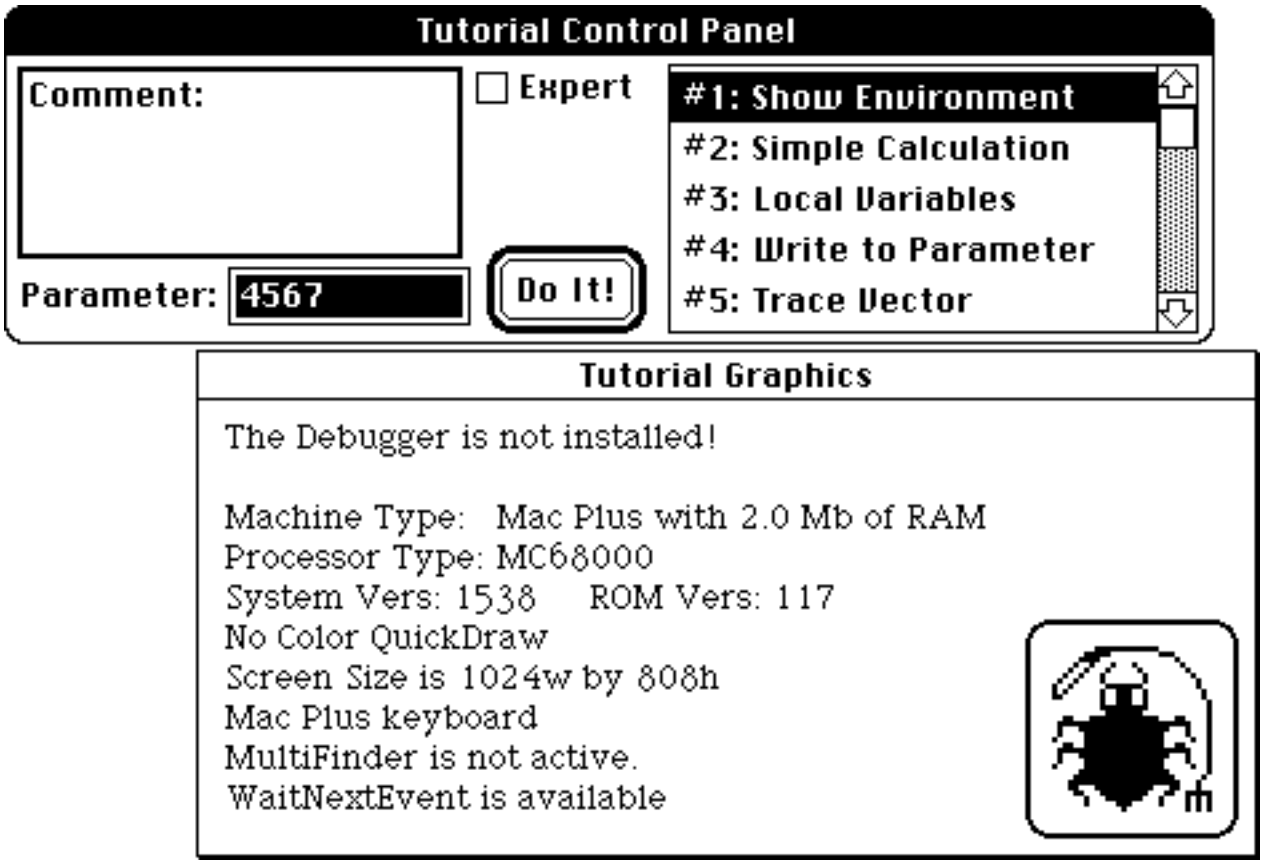
Start by launching ***DebugTut***. Just double-click on the icon: 

Note that ***DebugTut*** produces two windows: "Tutorial Control Panel" and "Tutorial Graphics". They should look like the windows on the next page.

You can position these windows wherever convenient on your screen but you cannot change their size. You can terminate the execution of ***DebugTut*** by **Cmd**-Q from the keyboard.

The control panel is a dialog window which allows you to select and execute any one of 21 routines. The scrolling-text box on the right allows you to select a routine. The selected routine will be executed when you double-click on its title or when you press the "Do It" button while its title is highlighted. Pressing the Return or Enter key while the Control-Panel window is the active window will also cause the selected routine to execute.

The comment box on the left side of the control panel will report any results of executing a test. The edit-text box labeled "Parameter" is used for input. The check-box labeled "Expert" is used to modify the behavior of ***DebugTut*** when it is running in background under MultiFinder.



You can safely attempt to execute any of the routines whether or not ***The Debugger*** is present. But most of the routines can only work properly under the control of ***The Debugger***. Tests numbered 1, 2, 3, 4, 5, 19, and 21 will not cause any problems without ***The Debugger*** and are allowed to execute fully even if ***The Debugger*** is not present. (Most of these tests do not produce any useful results without ***The Debugger***). In the absence of ***The Debugger*** all the other routines will produce messages in the comment box that indicate that the selected test has been aborted.

Try executing a few of these tests to get used to ***DebugTut***. Here are a few tests you could check out:

Test #1, "Show Environment", reports your machine environment on the "Tutorial Graphics" window. Test #1 was automatically selected and executed when ***DebugTut*** was launched. Does it correctly describe your system?.

Test #2, "Simple Calculation", expects a decimal integer in the "Parameter" edit-text box. It calculates the square of this integer and reports the result in the comment box. If you have not altered the default parameter of 4567, the result should be 20857489. Choose a few values in the range (-32768 to +32,767) and verify that Test #2 does calculate the square of the parameter correctly.

Test #21, "Not Implemented", simply posts an appropriate dialog for 2 seconds.

We will have more to say about all of the tests as run under the control of ***The Debugger***.

Preparing to Launch *DebugTut* **Under Control of** *The Debugger*

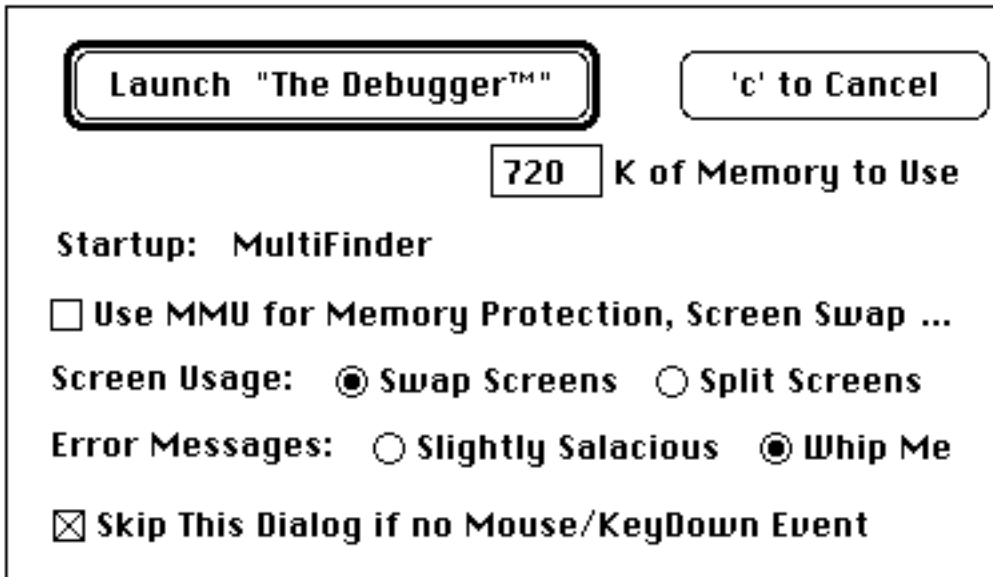
Now that you have seen how to initiate a ***DebugTut*** test routine from its control panel, you are perhaps ready to try running test routines under the control of ***The Debugger***. If you have not already done so, terminate the ***DebugTut*** application by **Cmd-Q** or by "Quit" under the File-menu.

Verify that you have these three files in the same folder on your disk:

DebugTut DebugTut.map and DebugTut.dsi.

Install ***The Debugger*** as per Debugger Tech Note #0 and reboot.

During the loading of INITs you will be presented with The Debugger startup dialog:



Use the edit-box to set the memory assigned to ***The Debugger*** to be 720K or larger, and select either swap or split screen operation. It you have only one monitor, then Swap Screens will always be used.

Press the "Launch" button. You will be presented with an alert inviting you to indicate where ***The Debugger*** is located.

Press retun to dismiss this dialog. Use the resultant familiar file-select dialog to locate and designate ***The Debugger*** file.

If all is well, ***The Debugger*** will be launched. You will see it set up a new menu bar and put up two windows: "**ROM.dsi**" and "**-Notes-**". The n The Debugger will "drop out" and the Finder will launch and establish the desktop. Find the folder containing DebugTut and open it. Double-click on DebugTut so as to launch it.

The Debugger will put a message in it's -Notes- window that it has read the DebugTut.map file and open up a Code blks window. ***The Debugger*** has stopped DebugTut prior to it executing its first instruction.

"**∂3.-Code Blks-**" will contain a list of the names of the code blocks for ***DebugTut***, beginning with "start" and "eventLoop". In the text on the **-Notes-** window, you should see the type and version-date of your copy of ***The Debugger***.

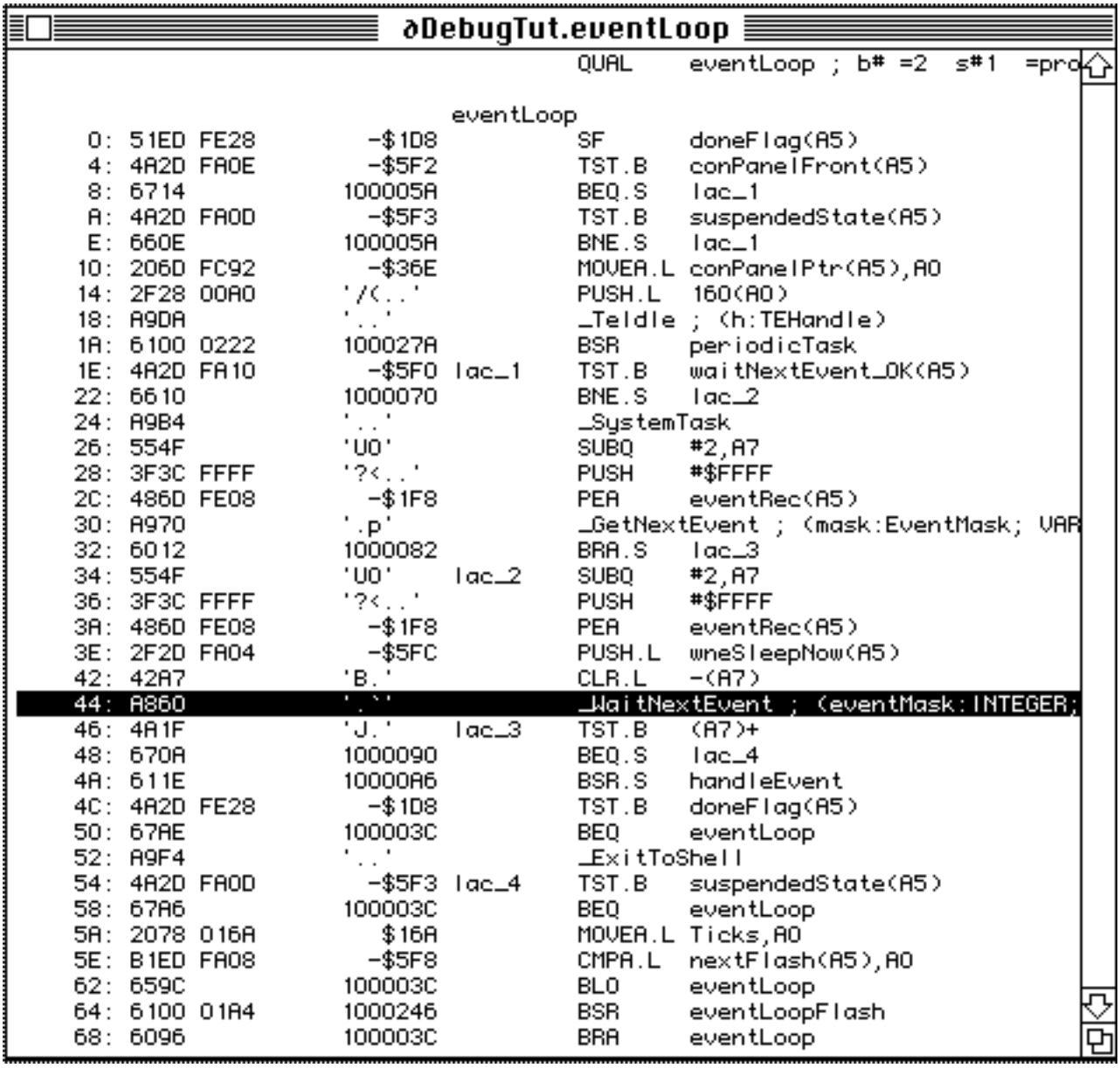
Note that the prefix "∂", as in the window name, **∂3.-Code Blks-**, designates "**task**". Tasks are code entities other than the System, such as an application or a driver, for which The Debugger has prepared symbol tables. You can obtain a list of the current tasks by selecting **Task and File info** from the Misc menu (Ω-**T**). "Ω" is a designator of a keyboard combination requiring you to press the option and command keys. That is, Ω-**T** is option-command-T). If you check the current task list, you will find that ***DebugTut*** is task "3".

Spend a few minutes looking at the menus (but do not make any selections yet). In particular, note the leftmost menu labeled 97K or some similar number of K. (This is the menu called the **-D- Menu** as described in the Technical Reference Section 6.16). This menu label shows you how much of the memory which you assigned to ***The Debugger*** back at the main startup dialog remains unused. If you pull down this menu you will discover an item: **Flush Volumes and Boot**. When you want to reboot the machine, select **Flush Volumes and Boot**. (When you are more familiar with ***The Debugger***, you will learn of other ways to return to the desktop or to change to another Problem Program).

The Technical Reference Section of this manual has a subsection devoted to each of the menus that you have been viewing. On some future occasion you should spend a little time reading through these menu descriptions.

Let us learn some simple things before we actually launch ***DebugTut***.

Select any code block by double-clicking on that block name in the ***∂*1.-Code Blks-** window. Then enter **Cmd-D** on the keyboard. This will create a new window containing a symbolic structured disassembly of that block. If you selected "eventLoop", here is the resultant window:



Note that "lac_1", "lac_2", and "lac_3" are labels with scope local to this block. Such labels are discovered and named by ***The Debugger*** on its own without the aid of any external file. Names of routines like "handleEvent" and global variables like "eventRec" have been drawn from the **.map** file.

You can produce a disassembly of any routine named in a disassembly window by double-clicking on that name to highlight it and then keying a **Cmd-D**. Try this on "handleEvent" as seen in the "eventLoop" disassembly. Note that **Cmd-D** is equivalent to "**Code|Data Blk**" under the Display-menu. Note that you can dismiss the front window by clicking in its go-away box, by keying **Cmd-K** or by selecting "Close" under the File-menu.

Actual Launch of *DebugTut*

We are now ready to launch ***DebugTut***. To do this, pull down the Go-menu and select "**Exit to Program**" (**Cmd-E**). ***DebugTut*** will now take control and create its two windows and its menu bar. (If you are running on one screen, the windows and menu bar of ***The Debugger*** will disappear. Note that

the “Tutorial Control Panel” now sports a ***Debugger*** icon to remind you that ***The Debugger*** is just waiting in the wings ready to take control when it is appropriate. Check the environment summary on the Tutorial Graphics window. The top line should say “The Debugger is watching over you!”.

Back to *The Debugger*: “Show me the structure of ...”

How can we transfer control from ***DebugTut*** back to ***The Debugger***? There are a great many ways. For example, we will soon see ***DebugTut*** execute a one of two special trap instructions: **Debugger** or **DebugStr**, designated for the purpose of accessing a debugger. Because ***DebugTut*** is currently executing a conventional eventloop we can expect that it will respond to the special key combination **option-**\. That is, ***The Debugger*** is watching events. If ***The Debugger*** sees a keyboard entry combination **option-**\, it is set up to seize control at the end of the GetNextEvent trap call which processes this keyboard event.

Try this now. Key **option-**\ and see what happens. You should get an immediate response as ***The Debugger*** takes control and puts up windows documenting the state of the system as it was as the **option-**\ event was recognized. Note first that ***The Debugger*** has generated a disassembly of a ROM routine. On that disassembly an instruction is highlighted: the instruction following the instruction that triggered the break to ***The Debugger***. On the preceding line you can see a GetNextEvent instruction. What happened was that the event loop of ***DebugTut*** executed a MultiFinder-friendly WaitNextEvent trap and this lead eventually to the execution of a GetNextEvent instruction by the Mac OS. ***The Debugger*** gained control at the completion of that GetNextEvent trap.

While we are back in ***The Debugger***, note that three new windows have been opened: **-Registers-**, **-Stack State-** and **-Trap/User Call-**.

You can see that the **-Trap/User Call-** window gives us more details on the _GetNextEvent trap call that gave control to ***The Debugger***. And the **-Stack State-** window documents the call chain by which we reached the breakpoint highlighted in the ROM disassembly. The most recent events are at the top of the list. Thus, reading from the bottom, we can see that at a specified address in eventLoop, a WaitNextEvent trap called a routine in ROM which in turn lead to the GetNextEvent instruction.

Take a moment to examine the **-Registers-** window. The contents of the Di and Ai registers are of no special interest at the moment. But note that the current graphics port is named as “Tutorial Control P...”. That sounds right. Further this port is designated as gp@168146 or some such address. (The actual address will vary depending on the system environment). To learn about the strength of the ***The Debugger*** in showing structures in comprehensible form, double-click on the **gp@...**, highlighting it, then press **Cmd-space**. (i.e. With the command key down, press the space bar). The result should be a new window containing a display of the fields of the current graph port exactly as defined in the QuickDraw chapter of Inside Macintosh (page I-148).

On this latest window you can see that the clip region, “clipRgn” is a certain “Region_@...” Double-click on the **Region_@...** and key in **Cmd-space** (for “show me the structure of”) and you get the structure of that clip region with its rgnSize and rgnBBox fields specified.

Resuming Execution of *DebugTut*

But where are we? Recall that we launched ***DebugTut*** and then broke back to ***The Debugger*** by means of the special key combination **option-**\ that allows ***The Debugger*** to break into most event loops. Now after the above exploration of the structures of graph ports and regions, Press **Cmd-E** to resume ***DebugTut*** .

DebugTut is now back in control. Ensure that the content of the “Parameter” edit-box is the integer 4567. Now select Test #2 and cause it to be executed by double-clicking on its title in the scrolling-text box (or just highlight the title and press the “**Do It**” button). Control will promptly revert to ***The Debugger***. Why? Examine the disassembly of routine “simpleCalc” which has just been added to the ***The Debugger***’s screen. The highlighted instruction,

```
BSR       getDecimalParam
```

is the next instruction that will execute if ***DebugTut*** is allowed to resume. The previous instruction above it is the mechanism by which ***The Debugger*** gained control. That is, `DebugStr` caused the transfer of control. On detecting a `DebugStr` trap, ***The Debugger*** takes control and expects that there will be the address of a string on top of the stack. Note that in the current instance, the instruction sequence is:

```
PEA   stringVar1(A5)   ; push address of string
_DebugStr           ; break to The Debugger
```

The Debugger responded to this by seizing control, popping the address of the string off the stack and using that address to write the string to the **-Notes-** window. Check the **-Notes-** window, "About to execute Test #2: Simple Calculation" was posted in just this fashion. But we can check this further. Go to the **Misc** menu and select "**Update Values**" (or key **Cmd-U**). This produces a **-Value-** window listing many of the global variables of ***DebugTut*** along with their values formatted according to their types. `stringVar1` is listed as type String with value "`About to execute Test #2: Simple Calculation. `", as expected.

But how does ***The Debugger*** know that there is a global variable called `stringVar1` of type String and where in memory to find it? Recall that we provided two auxiliary files: DebugTut.map and DebugTut.dsi. The **.map** file lists the name of each global variable along with its offset off register A5 (the frame pointer for ***DebugTut***'s world). This **.map** file was produced by the linker as it built ***DebugTut*** from the object file assembled by the MPW Assembler according to the assembly language source for ***DebugTut***. The **.dsi** (mnemonic for Debugger Startup Information) file was typed by hand. A part of this **.dsi** file is a list of global variable names and their types. In particular DebugTut.dsi contains the line:

```
stringVar1:String
```

Thus from the .map and the **.dsi** files plus the current state of the global memory area of the Problem Program, ***The Debugger*** can generate the contents of the **-Values-** window.

With the **-Values-** window in front of you, you can easily scroll down and learn that a type can designate a rather complex record and that record will be displayed with fields named and typed. As proof of this, examine the variable `dStorageCP` (mnemonic for "dialog Storage for Control Panel") typed as a window record. As you have come to learn, you can further select and **Cmd-space** ("show the structure of") such substructures as the ControlRecord bearing in turn The ControlList of the control-panel dialog.

Now let us return to following the execution of Test #2. But the window with our current disassembly is buried in a pile of windows! ***The Debugger*** has a useful feature for dealing with this common problem. Recall that we need to find the "∂DebugTut.simpleCalc" window. Place the cursor anywhere in the middle of the screen and execute a **option-key/mouseclick**. This generates a pop-up menu listing all current windows. Just select "∂DebugTut.simpleCalc". You can also take a moment to dispose of most other windows that have a go-away box; we can easily regenerate any of these later.

Where are we now? We are at the beginning of test #2 in routine `simpleCalc .` The current instruction is highlighted in the simpleCalc window. We can still see the message generated in the **-Notes-** window by **DebugStr**. And if we were interested, the **–Registers-** window reports the absolute address location of the program counter as PC=xxxxxx.

Single Stepping Through simpleCalc

Let us step through simpleCalc instruction by instruction. But we will choose not to step into traps or subroutines. Pull down the Go-menu. The item we need is "**Step Thru**" or **Cmd-comma**.

Our first **Cmd-comma** executes subroutine `getDecimalParam .` You are to assume that this subroutine fetches the decimal string which you set up in the "Parameter" edit box on the Tutorial Control Panel, converts it to a long integer and places the result in a global variable called "`parameter `". Is that what appears to have happened? Key **Cmd-U** to update the **-Values-** display. Check for a global called "`parameter `". Yes! Its type is LongInt and its value is shown in decimal as 4567, the exact decimal value we set up in the edit-box on the control panel.

Step again with another **Cmd-comma**. From the disassembly we see that we are next moving a value from `parameter` to register D0. Check the **-Registers-** window. D0 is now 11D7. What is that in real money , you ask? (***The Debugger*** always reports CPU-register contents in hex). Highlight the 11D7 in the -Registers- window. Pull down the Tables-menu and select "**Convert Hex to Dec**" or **Cmd-**. The result goes to the **-Notes-** window: 4567 as we might have anticipated.

Note also that when you step through a program, a trail of useful information is written into the disassembly window. For example, on the instruction line above where the content of `parameter` was moved to register D0, you can see the absolute address of `parameter` (the @xxx) and the long value that was moved (11D7).

Step along a few instructions and follow the action. The 11D7 value is copied into register D1. Then both copies of the value 11D7 are treated as signed integers and are multiplied together. The product is generated in D0. Next you see that product is moved to the `parameter` global. Updating the **-Values-** window allows us to see the result in decimal: 20857489 (which is 4567 squared).

Step a little further until the `_Pack7` trap has been executed. At this point the **-Trap/User Call-** window gives us a useful interpretation of the action of this trap. We see that this trap was entered with a selector param of zero which is a number to string conversion. The result is the string, "20857489".

Step on past the `moveString` subroutine call. Update the **-Values-** window (**Cmd-U**). We can see a **bullet** (•) to the left of `stringVar2` : this indicates a new value (i.e. changed since the last update of the **-Values-** window). We can see that `stringVar2` now has the first half of a message: "The square of parameter is ". Stepping 3 more times past the `appendString` subroutine call and we see that `stringVar2` now has the square-value string appended, completing the message.

Finally, one last step past the `commentVar2` subroutine allows us to see the complete message written to the Comment box on the control panel of ***DebugTut***.

For present purposes, this is the end of our tracing through Test #2. To return control to ***DebugTut***, execute **Exit to Program** from the Go-menu or key **Cmd-E** and you will be ready to select another test.

Take a Closer Look if You Like

Perhaps you were unhappy about taking it on faith that a routine like `moveString` does correctly move a Pascal-type string from one location in memory to another. For brevity, we chose above not to step into subroutines. But investigate the **Go** menu. "**Step Into**" or **Cmd-?** lets you step at the single instruction level and does step into subroutines or traps.

Rerun Test #2. Use "**Step Thru**" (**Cmd-comma**) repeatedly until an interesting subroutine call is about to execute. Then **Cmd-?** will **step into** the first instruction of that subroutine. As you might well expect by now, ***The Debugger*** will put up a new window containing the disassembly of the subroutine. You can the **Cmd-comma** through that subroutine. Should you encounter an RTS instruction, the window containing the calling procedure will automatically come to the front again.

Stack Frames and Local Variables

This exercise is intended to illustrate how to use *The Debugger* to look at stack contents and stack frames. We will look at the behavior of a Pascal-type subroutine.

If you are not familiar with stack frames, local variables, and methods of passing parameters to a subroutine via the stack, you should read Inside Macintosh pages I-95 through I-98 and/or Chapters 5 and 6 on compiled code in Scott Knaster's book "How to Write Macintosh Software", Second Edition, Hayden Books, 1988.

In order to provide an easy-to-understand example of a routine that generates much stack activity, the code we will examine calculates three simple quantities using a classic recursion method. Given a small integer '**n'** this code will calculate **n!**, **n^n**, and the **sum of the integers** from 1 to n. The code is based on a subroutine named "`recurseSub` " that calculates each of these three functions of **n recursively**. That is, it uses the relations:

n!	=	n *(n-1)!
n^n	=	n*n^(n-1)
sum(n)	=	n+sum(n-1)

Thus `recurseSub` can calculate 9! by calling itself to calculate 8! by calling itself to calculate 7! etc. until it needs to evaluate 1! which is defined to be one.

Because these relations are similar in form, one routine can calculate all three functions in the same recursive sequence. But note that the relation for **n^n** has a quirk not found in the other two relations. The righthand side of the relation for calculating **n^n** is **n** times **n^(n-1)**. But **n^(n-1)** is a function that involves both **n** and **n-1**. Clearly, to calculate 9^9 by this recursion, at some stage we will calculate 9 times 9^3, not 4 times 4^3. That is we will have to carry the original parameter, 9 throughout the recursive sequence. This implies that `recurseSub` must receive two parameters, "**n_original**" and "**n_local**" and will return three results, **n_local**!, **sum(n_local)**, and **n_original^n_local**

In fact all of this works out quite simply in practice. Let us trace through the actual code

Start with ***DebugTut*** in control with ***The Debugger*** already launched and waiting to regain control. Select Test **#3: Local Variables** and press the "**Do It!**" button. ***The Debugger*** will gain control in a code block called "**locVarsTest**" after a DebugStr trap. In the disassembly window you can see that we are about to push five values onto the stack, call **recurseSub**, and recover three values from the stack and place them in global-variable locations.

Because we will be interested in stack content, it will be useful to create a window that is dynamically tied to the stack pointer. That is we want a window that is always updated and which shows the numerical content of the stack starting from the address which is the current content of register A7. This is easily done. With **nothing selected** on the currently uppermost window, issue a **Cmd-space**. This will summon a command dialog box. Type "*@RA7" follow by a carriage return. The window we need should now present itself.

Step through (**Cmd-comma**) 6 instructions while watching the ***@RA7** window. The stack will receive five entries in this order:

- a zero word to make room for the **sum(n)** result
- a zero long.to make room for the **n^n** result
- a zero long.to make room for the **n!** result
- ,a word of value 4 to serve as **n_original**
- ,a word of value 4 to serve as the first **n_local**,

Now step into the subroutine, **recurseSub**, by a **Cmd-?**. The ***@RA7** window should now show you the return address associated with the call. You can discover where this address is located by highlighting the return address in the ***@RA7** window and issuing a **Cmd-D**. You confirmed that this is in fact the expected return to the **locVarsTest** code block. Now step through (**Cmd-comma**) the LINK and MOVEM instructions at the start of **recurseSub**. The ***@RA7** window now shows you a lot more is on the stack.
But from here on in, a non-symbolic display of the stack as provided by the ***@RA7** window becomes a very clumsy way of following the stack activity. ***The Debugger*** is about to come to the rescue, however. First look again at the disassembly of **recurseSub**.

At the very top of the window is a compact summary of the stack frame. ***The Debugger*** has provided arbitrary names for references to the stack frame and summarized these names here. Locations in the stack frame at higher addresses than the frame pointer are arbitrarily called "param*x*"'s. These may indeed be parameters or they may be locations that communicate results back to the caller. Locations in the stack frame at lower addresses than the frame pointer are given names starting with the letter "v", such as **vgg-2**. These lower locations are always local variables and are almost never access by code beyond the current subroutine.

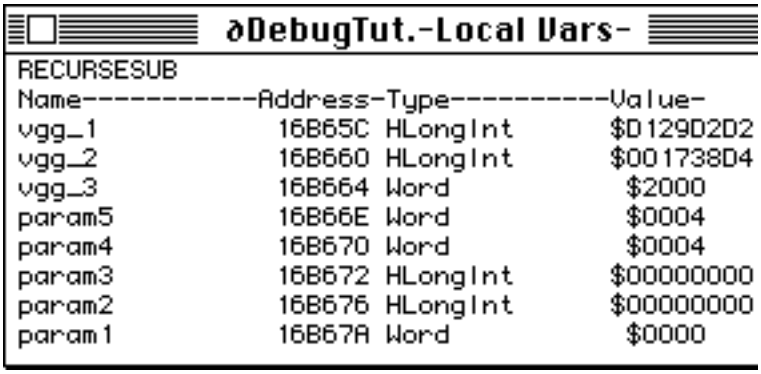
Here is an interpretation of the stack frame summary on your screen:

Debugger Name	Mnemonic Name	FramePtr Offset	Type
vgg_1	fact_local	-10	HLongInt
vgg_2	nToN_local	-6	HLongInt
vgg_3	sum_local	-2	Word
param5	n_local	8	Word
param4	n_orig	10	Word
param3	fact_result	12	HLongInt
param2	nToN_result	16	HLongInt
param1	sum_result	20	Word

Here is a rendition of the disassembly of recurseSub enhanced with comments:

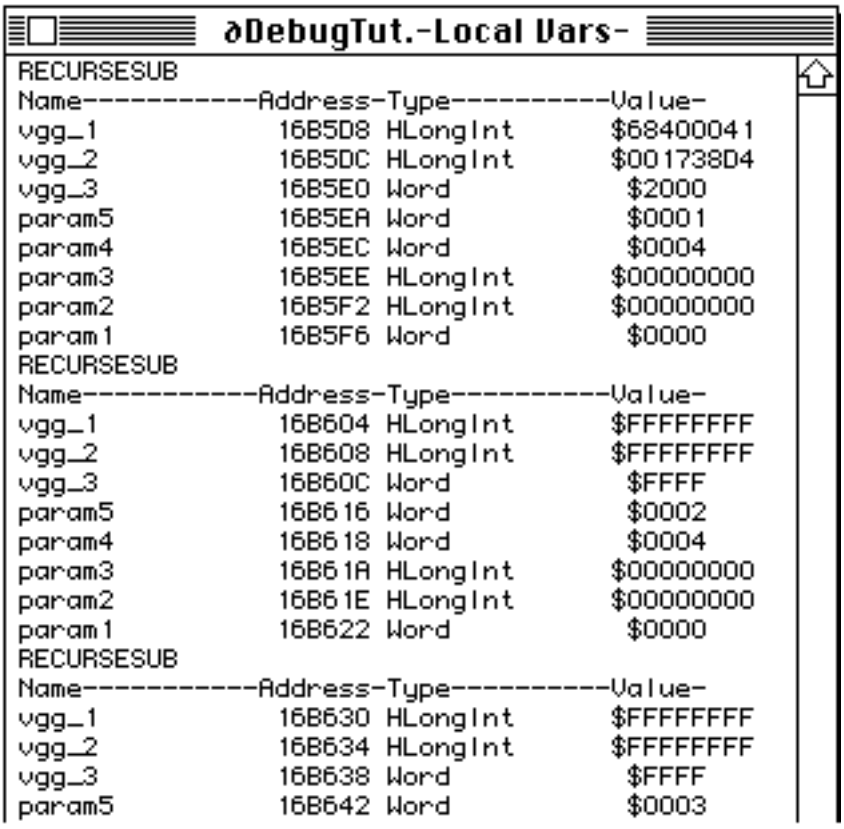
```
recurseSub      LINK        A6,#-$A           ; make room for local vars
4:              MOVEM.L     D0-D2,-(A7)

8:              CMPI        #1,param5(A6)     ; test local ‘n’
E:              BNE         lgg_1             ; if n <> 1

      ;         bottom of recursion, local ‘n’ is one. Which means:
                ; factorial:=1, n^1:=n, sum:=1 ...

12:             MOVEQ       #1,D0
14:             MOVE.L      D0,param3(A6)     ; factorial result
18:             MOVE        param4(A6),D1     ; the original ‘n’
1C:             EXT.L       D1
1E:             MOVE.L      D1,param2(A6)     ; n^n result
22:             MOVE        D0,param1(A6)     ; sum result
26:             BRA         lgg_2

2A:   lgg_1     ; normal case where bottom has not been reached

                ; call recurseSub recursively...
                CLR         -(A7)
2C:             CLR.L       -(A7)
2E:             CLR.L       -(A7)
30:             PUSH        param4(A6)        ; original ‘n’
34:             PUSH        param5(A6)        ; local ‘n’
38:             SUBQ        #1,(A7)           ; parameter is now (n-1)
3A:             BSR         recurseSub        ; the recursive call
3C:             POP.L       vgg_1(A6)         ; factorial(n-1)
40:             POP.L       vgg_2(A6)         ; n^(n-1)
44:             POP         vgg_3(A6)         ; sum(n-1)

                ; now calculate factorial, n^n and sum
                ; from values received from the recurseSub call ...

48:             MOVE        param5(A6),D1
4C:             MOVE.L      vgg_1(A6),D0
50:             BSR         MULT16X32         ; D0.L := D1.W * D0.L
54:             MOVE.L      D0,param3(A6)     ; factorial partial result
58:             MOVE        param4(A6),D1
5C:             MOVE.L      vgg_2(A6),D0
60:             BSR         MULT16X32         ; D0.L := D1.W * D0.L
64:             MOVE.L      D0,param2(A6)     ; n^n partial result
68:             MOVE        vgg_3(A6),param1(A6)
6E:             MOVE        param5(A6),D0
72:             ADD         D0,param1(A6)     ; partial sum

76:   lgg_2     MOVEM.L     (A7)+,D0-D2
7A:             UNLK        A6                ; discard local variables
7C:             MOVE.L      (A7),4(A7)
80:             ADDQ        #4,A7             ; discard parameters...
82:             RTS
```

We are not particularly interested in the details of the calculations involved here. We just want to see how we may best observe stack frame events. There is a feature of ***The Debugger*** which is designed to help here. Generate a **-Local Vars-** window by issuing a **Cmd-shift-U** or by the menu selection **Misc:Update Locals**.



Consult the table provided above that gives you the mnemonic interpretation of these variables. The three "vgg" local variables contain junk at this stage. But the five parameters are exactly those which you observed being loaded onto the stack in the **locVarsTest** code block. The two versions of **n** each have value 4. The three results are all cleared to zero. As the recursion proceeds, we expect param5 (i.e. n_local) will change from 4 to 3 to 2 etc. By contrast param4 (n_orig) may be expected to stay constant. But keep in mind that we are describing a process that achieves recursion by `recurseSub` calling itself and creating a new stack frame each time this occurs. Thus the expected progression in param5 will be observed by comparing one stack frame with a later one.

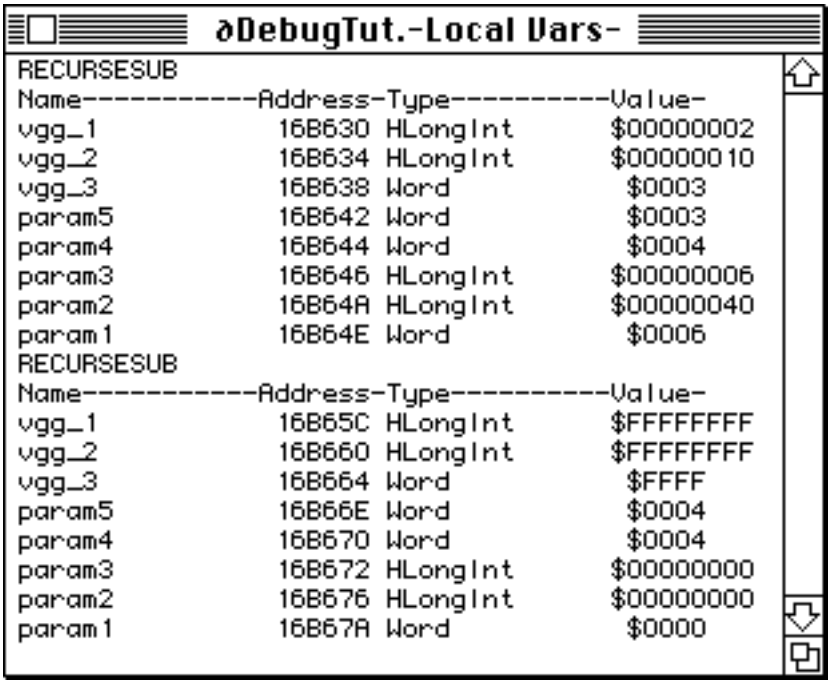
The most convenient way to observe the development of this calculation and to see the stack frames is to now set a breakpoint right at recurseSub. Highlight "recurseSub" in any window and issue a **Cmd-B**. Now go through the following sequence three times and take note of the **-Local Vars-** window: **Cmd-E** then **Cmd-shift-U**. After each **Cmd-shift-U** you show see an update of the **-Local Vars-** window showing a new stack frame attributed to recurseSub has appeared at the top of the **-Local Vars-** window. After the third **Cmd-shift-U**, this is what you should see:



Note that the most recently created frame, the one at the top of the above figure, features a param5 equal to one. This is an indicator that we have reached the bottom of the recursion. If you examine the commented listing of `recurseSub` above, you will note that when param5 (n_local) is one, recurseSub does not call itself again. Rather `recurseSub` places level-one results in param1, param2 and param3 and returns to the caller. The caller is `recurseSub` , of course.

If you follow the listing further, a return to `recurseSub` , leads to some calculation followed by results being written to param1, param2 and param3 and a return to the caller. The recursion thus unwinds with each return to caller being accompanied by the destruction of one stack frame. Level-2 results are posted to the level-2 stack frame then the level-1 stack frame is discarded. Level-3 results are posted to the level-3 stack frame then the level-2 stack frame is discarded and so on. In our example, this will continue until level-4 results are posted to the stack frame accessible to the three moves from the stack at the end of code block **locVarsTest**

To see the recursion unwind as has just been described, a new breakpoint is required. Set a breakpoint at **recurseSub+76**. (Highlight that instruction line in the `recurseSub` window and issue a **Cmd-B**.Then issue **Cmd-E** / **Cmd-shift-U** pairs and you will see the low-level stack frames disappear one by one from the **-Local Vars-** window. After you have done this tree times this is how the **-Local Vars-** window should look:



Note that param1 in the upper frame (level-3) is sum(3) = 3+2+1=6. param2 is 4^3=64=$40. And param3 is 3!=3*2*1=6. Thus it appears that recurseSub is doing its job.

Issue one last **Cmd-E** / **Cmd-shift-U** sequence and verify that the level-4 results are what they should be. Finally step through (Cmd-comma) the last five instructions of recurseSub, stepping through the final RTS instruction and observe the return to **locVarsTest**.

Use **Cmd-U** to generate the **-Values-** window. Locate global variables FACTFINAL, NNFINAL and SIGMAFINAL. Step through the three instructions that pick the results from recurseSub off the stack and that place the results in these global variables. (The sum goes into SIGMAFINAL). Watch as each of the three globals receives a value (**Cmd-U** to update the **-Values-** window) and confirm that 4!, 4^4, and 4+3+2+1 are correctly represented here.

Before leaving this exercise, try repeating the calculation for some parameter other than 4. (recurseSub can accept any integer from 2 to 9). That is do not follow the internal action of recurseSub at all but just run the calculation. To achieve this, select **Stops:Clear All Bkpts** and thus dismiss the two breakpoints we set earlier in recurseSub. Next set the program counter to **locVarsTest**+1A, the CLR -(SP) instruction which was the first of three instructions providing space for results from the call to recurseSub. Next set a breakpoint at the BRA TESTEXIT instruction that terminates **locVarsTest**. Now step through the three CLR-instructions and the MOVEQ #4,D0 instruction as well. To alter the value of 4 now in D0 to your choice of 2 through 9, use **Cmd-Y** (or **Go:Set...**). Then type **RD0=7**, say, followed by a carriage return. Verify in the **-Registers-** window that your choice of parameter is now in D0.

A **Cmd-E** should now execute the remainder of **locVarsTest**. Check the **-Values-** window to see if the expected values were generated.

To return control to ***DebugTut***, press **Cmd-E** and you will be ready to select another test.

Writing a Message to an EditText Field

This exercise is intended to get you used to looking for items in the application heap and elsewhere. It writes a message to the edit-text item labeled “Parameter” in the Control Panel of DebugTut.

Start with ***DebugTut*** in control with ***The Debugger*** already launched and waiting to regain control. Select Test **#4: Write to Parameter** and press the “**Do It!**” button. ***The Debugger*** will gain control in a code block called “`writeParamTest` ” after a DebugStr trap.

Step through (**Cmd-comma**) the next instruction. This loads register A0 with a pointer to the message we intend to write. Notice that the address of the message (a Pascal string) has been posted with an @-

sign prefix in the `writeParamTest` disassembly window to the left of the LEA instruction. This address is also available in the -Registers- window as the content of A0. Highlight one of these two versions of the address and issue a **Cmd-space**. This will produce a data window which should show the string "I did it!" prefixed by a length-byte of 9.

Next step into (**CMD-?**) the subroutine `writeToParam` . Step through (**Cmd-comma**) the next 7 instructions such that the GetDItem trap has just executed. The encounter with the trap produced a **-Trap/User Call-** window summarizing the trap parameters and results. One of these results is a handle to the static-text item, item #3, of the ***DebugTut*** control panel dialog. To learn more about this item, we need to look at the application heap for ***DebugTut***.

Press **Cmd-H** and a heapzone window for ***DebugTut*** is generated. There are many items in this heap.

The **-Trap/User Call-** window tells us that we are dealing with a grafport at a certain address. But if we are interested in dialog items, we can expect that there is a dialog record also at the same address. For example if the **-Trap/User Call-** window reported that the item in question is on a dialog specified as ^GrafPort_@16B946, then you will find an "Item list of `dr@16B946` " listed as being in the application heap. (You will notice several other references to this same dialog record as well).

Double-click on the "`dr@xxx` " to highlight it and issue a **Cmd-space**. A dialog-record window will appear. One of the fields of this record is "textH", specifying the location of a text-edit record. Double-click on this location specification (in the form "`^^TERec_@xxx` ") and issue a **Cmd-space**. This action will generate a text-edit record window which will provide the length and location of the text currently set in the "Parameter" text edit box. If you have not altered this text since DebugTut started up, You should find that the current content is "4567" with length 4. A further scan of the heap window will reveal the this little piece of text occupies a small block on the heap.

You will find an entry something like this ...

```
H tx@026852      4 <...> @023BA8 Item # 3 EditText of dr@16B946
```

If you continue to step through `writeParamTest` , you will see the message "I did it!" as it is written to the edit-text window via the SetIText trap. If you then attempt to follow the above path to `"...Item # 3 EditText of dr@16B946"` you will discover that the block that previously contained "4567" (or whatever) has been released and a new block containing "I did it!" has been created.

To familiarize yourself with accessing the System heap, press **Cmd-shift-H**. You might also bring up a **-ROM Patch Info-** window at this time by the menu selection **Display:ROM Patch Info**. Many of the entries in the system heap are patches to traps and the **-ROM Patch Info-** window is the key to finding out where the patches are located and who created them.

Finally, take a moment to see how easy it is to save anything you see on any window. This is an area where ***The Debugger*** is particularly proficient. For example, highlight anything you like, in any window. Issue a **Cmd-N** and that item is written to the **-Notes-** window. Activate a window. **Cmd-A** highlights all of it. You can copy (**Cmd-C**) anything and paste (**Cmd-V**) it into the **-Notes-** window. The content of the -Notes- window can be saved as a text file. Try out **File:Save** or **File:Save As...** .

It is often useful to read a TEXT file while using The Debugger. Try to open one via **Cmd-O**.

When you are done exploring, return control to ***DebugTut***. by pressing **Cmd-E**.

Using the Trace Vector

This exercise will show you how to use the trace vector of the MC680x0 to enhance the range of conditions that can be detected by ***The Debugger***.

There are times when you would like to make an unusual test very frequently. The trace-vector mechanism of the MC680x0 allows you to perform any test which you can express as code, as often as once for each and every instruction of the PP. When you have detected a condition of interest, you can have The Debugger regain control or perhaps post notes to the -Notes- window.

The trace example we will discuss here has three main code blocks accessed as subroutines:

```
BSR      traceOn          ; initialize trace variables and enable trace
BSR      traceObject      ; code sequence to be traced
BSR      traceOff         ; end the tracing
```

The Debugger uses the trace vector (Vector # 9) in several of its operations, such as supporting "Step Into". This means that it is necessary to save the content of the trace vector, as set by ***The Debugger***, replace it with our own vector while we test, and finally to restore it to the state set by ***The Debugger***

The routine "`traceOn `" saves the trace vector as set by ***The Debugger*** in the global variable called "traceSave" It then replaces the trace vector with a pointer to a routine called "`traceHndlr `". Finally it sets the trace-bit, bit number 7 of the system byte of the status register.

With the trace-bit set, the `traceHndlr` routine is executed after the execution of every PP instruction. Any test to be performed is coded into the **traceHndlr** routine. DebugTut includes a very simple test to illustrate the technique. It tests the next instruction to be executed in the PP. If that instruction is a NOP, **traceHndlr** writes the address of that NOP to a circular list which can retain the addresses of the most recent 31 `NOP`'s executed. The circular list is initialized by "`traceOn `".

Detecting `NOP`'s is a simple test that illustrates the mechanism. Range-checking one or more global variables might be the kind of test that would be useful in real-world debugging situations.

The routine "`traceOff `" ends the tracing by clearing the trace-bit of the system byte of the status register. "`traceOff `" then restores the original trace vector as set by ***The Debugger***

Here is a commented list for the trace-handler in DebugTut.

```
;               traceHndlr - respond to a trace exception
traceHndlr      MOVEM.L       A0/D0,-(SP)
                MOVE.L        10(SP),A0          ; Get saved program counter from stack.
                                                 ; A0 = ptr to the next inst to be traced.
                ; now test the next instruction to be traced ...
                MOVE          (A0),D0            ; first word of the instruction
                CMP           #$4E71,D0
                BNE.S         trContinue         ; if not a NOP instruction
                MOVE.L        traceSave(A5),traceVector ; restore original Trace Vector
;               O.K. to break to The Debugger here
traceBreak      MOVE.L        A1,-(SP)
                MOVE.L        listPtr(A5),A1     ; get pointer to circular list
                ADDQ          #4,A1              ; point to next location in list
                CMP.L         listTop(A5),A1     ; check if beyond top of list
                BLS           @2
                LEA           traceList(A5),A1   ; reset list pointer

@2              MOVE.L        A0,(A1)            ; write traced address to circular list
                MOVE.L        A1,listPtr(A5)     ; write new list pointer
                ADDQ          #4,A1              ; point to oldest address on list
                CMP.L         listTop(A5),A1     ; check if beyond top of list
                BLS           @4
                LEA           traceList(A5),A1   ; reset list pointer
@4              MOVE.L        #-1,(A1)           ; mark oldest address on list
; last chance to break to The Debugger before we replace the trace vector again
                LEA           traceHndlr,A0
                MOVE.L        A0,traceVector     ; replace vector with our own
                MOVE.L        (SP)+,A1
trContinue      MOVEM.L       (SP)+,A0/D0
                RTE
```

In order to not slow down the PP excessively, the test coded into the trace handler should be compact and efficient for the normal case in which the test fails and the PP is to resume as quickly as possible.

As the comments in the above listing reveal note, for the case where the test passes, the abnormal case, there is a section of the code within which normal breakpoints or action clauses (conditional bkpts) may be can be located and any of the features of ***The Debugger*** may be employed. However, no breakpoints should be used in the `traceObject` code itself.

With ***The Debugger*** in the background, select Test **#5: Trace Vector** and press the "**Do It!**" button. ***The Debugger*** will gain control in "`traceTest `" after a DebugStr trap.

Bring up a disassembly windows for `traceObject` (highlight the name and issue a **Cmd-D**). Notice that this code consists of a loop which will execute 19 times and provide 5 `NOP`'s per pass.

Generate a **-Values-** window (**Cmd-U**). Locate the trace variables relating to the circular list: `traceList, listTop` AND `listPtr .` When these variables have been initialize by running the trace test, the circular list extends from the address of (not the content of) `traceList` up to a higher address saved in TRACETOP. The most recent entry in the list is designated by the pointer, `listPtr .` Older entries are generally at lower addresses than newer entries. The oldest entry is replaced by -1 as a marker. `traceList` also has been defined to be of type Ptr4 in the ".dsi" file. That is, the **-Values-** window shows the content of `traceList` to be the lowest four addresses contain in the circular list

Set a breakpoint at the branch to `TESTEXIT` at the end of `traceTest .` Issue a **Cmd-E** to execute this test on `traceObject` **.** Display the content of the entire trace list. Verify that all addresses in the list correspond to the five NOP's in `traceObject` : highlight an address in the list and issue a Cmd-D.

When you are done exploring, return control to ***DebugTut***. by pressing **Cmd-E**.

The Classic Crashes

The following 12 exercises are very simple. Each illustrates a classic mechanism by which a program can crash and each allows you to see how ***The Debugger*** reports each of these conditions. In all cases ***The Debugger*** brings the PP to an orderly halt and informs you clearly what the problem is. In some cases DebugTut may be resumed after the crash. In other cases it will be necessary to boot the system.

Select the indicated test and press the "**Do It!**" button. ***The Debugger*** will gain control after a DebugStr trap prior to encountering the crash. Explore the code ahead via appropriate disassembly windows, then issue a **Cmd-E** to cause the crash.

Illegal-Instruction Exception	Test #6 **Cmd-E** to resume DebugTut.
Illegal F-Line Instruction Exception	Test #7 **Cmd-P** to set PC to BRA testExit then **Cmd-E**.
Division-by-Zero Exception	Test #8 **Cmd-E** to resume DebugTut.
CHK-Instruction Exception	Test #9 **Cmd-E** to resume DebugTut.
Trap-on-Overflow Exception	Test #10 **Cmd-E** to resume DebugTut.
Undefined A-Line Exception	Test #11 **Cmd-P** to set PC to BRA testExit then **Cmd-E**
Address Exception	Test #12 Exit by booting the system
Dereferencing Location Zero	Test #13 Exit by booting the system
Dereferencing Location Minus One	Test #14 Exit by booting the system
Unloading Current Segment, Benign	Test #15 **Cmd-E** to resume DebugTut.
Unloading Current Segment, Disastrous	Test #16 Exit by booting the system
Unloading Calling Segment	Test #17 Exit by booting the system

Jump-Tracing Exercise

NOTE:You must have a system with a MC68020 or MC68030 to run this example.

This section will acquaint you with the "**Go:Trace Jumps**" mode of ***The Debugger***.

Sometimes the evidence available when a PP crashes is insufficient to pinpoint the origin of it. The **-Stack State-** window is usually helpful on these occasions but only procedure calls are recorded; flow changes associated with branches and jumps do not appear here. If the PP undergoes complex branching within a procedure, it may be vital to your diagnosis to know the complete program flow within some procedure.

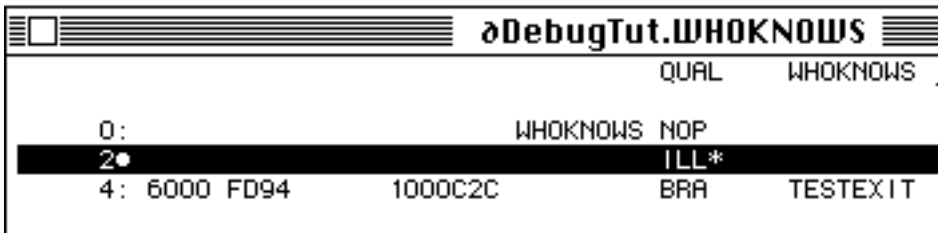
When enabled, the **Go:Trace Jumps** mode posts the last 20 changes in flow to a -Jumps- window. Let us see how this works.
Select Test **#18: Jump Tracing** and press the "**Do It!**" button. ***The Debugger*** will gain control in a code block called `wildJump` after a DebugStr trap.

Use the Go menu to select **Trace Jumps**. Confirm that there is now a check mark adjacent to the **Trace Jumps** item in the Go Menu indicating that the **Trace Jumps** mode is active.

Note that the disassembly of wildJump indicates that the next instruction to be executed is:

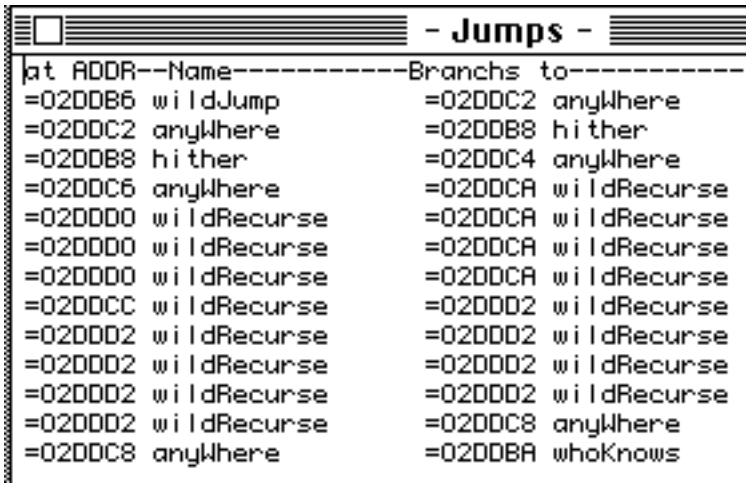
```
BRA  anyWhere
```

Let the processor take you take you to `anyWhere` and thence into the unknown by issuing a **Cmd-E**. The result will be promptly posted. You will crash in a routine called `whoKnows` at an illegal instruction. Here is the the structured disassembly which will now be on your screen:



The Notes window should contain : `"• Illegal Instruction @ PC =nnnn whoKnows+2"`

Also on your screen should be a **-Jumps-** window like this:



Note that the **-Jumps-** window provides a line with both symbolic and absolute addresses for every change of program flow. The value of the stack pointer (A7) is also recorded in the rightmost column. By examining the stack pointer column, you can often distinguish between **branches** and, **calls** and **subroutine** returns: but you must account for data pushes and pops within the routines.

Use **Cmd-D** to display structured disassemblies of `wildJump` and `wildRecurse` . (Highlight the address to the left of these labels in the **-Jumps-** window in turn and issue **Cmd-D**). Now you can easily reconstruct the execution sequence. The `BRA` to `ANYWHERE` is recorded on the second line of the **-Jumps-** window. The **-Jumps-** window and the disassemblies tell you:

- there were three **branches** : from `wildJump` to `anyWhere` to HITHER to YON
- `YON` **called** `wildRecurse`
- `wildRecurse` **called** itself three times
- `wildRecurse` **branched** to `recurseDone`
- there were three SUBROUTINE **returns** from `recurseDone` to `recurseDone`
- there was one SUBROUTINE **return** from `recurseDone` to YON,
- there was one final **branch** from YON to `whoKnows`

and we already know that `whoKnows` crashed. Now you have the full story of the sequence that led to the illegal instruction.

In a real-life debugging situation, you would probably be examining several of the routines listed on the the **-Jumps-** window for a programming error or unexpected condition that would develop a crash. These routines might be lengthy and involve data movement to and from the stack. And the crash itself might not be so neatly repeatable as is our ***DebugTut*** example here. But at least the **Trace Jumps** mode can give you a list of suspect routines on which you can focus.

To go on to some other ***DebugTut*** example:

Use the Go menu again to deselect the **Trace Jumps** item. Confirm that there is **no check mark** adjacent to it item in the Go menu. That is, the **Trace Jumps** mode should now be **inactive**.

Highlight the entire line, `BRA TESTEXIT` , in the `whoKnows` disassembly window and set the program counter to this instruction by issuing a **Cmd-P**. Now return control to ***DebugTut*** by issuing a **Cmd-E**.

A Numerical Exercise for the FPU

NOTE: You must have a Mac with a MC68020 /30 and a MC68881/82 FPU to run this exercise.

This section will give you the experience of tracing the execution of a lengthy numerical calculation using an FPU. This calculation also provides a rigorous test of much of the functionality of your FPU.

The calculation that will be traced here is the production of a long sequence of integers by a pseudo-random number generator of this form:

$$R_n = AR_{n-1} \bmod M$$

where: R_n is the n^{th} integer in the sequence.

A is a constant integer multiplier

M is the modulus

That is, we multiply the current pseudo-random number by A. Then we divide that product by M. The remainder of that division is the next pseudo-random number.

The choice of 'A', the multiplier, and 'M', the modulus, determine the period of the generator (the number of different values of R will be produced before the sequence repeats) and the degree of correlation between a value R_n and nearby members of the sequence such as R_{n+1} R_{n+2} etc.

In the current exercise:

$$M = 2{,}147{,}483{,}647 = 2^{31} - 1$$

and $A = 314{,}159{,}268$ (close to pi times 10^8)

With the above choice of M and A, it can be proved that the period is $2^{31} - 2$ or 2,147,483,646. That is, this generator produces every positive integer that can be represented by 31 bits with the exception of zero and the largest such integer, $2^{31}-1$.

Arbitrarily, we set the first integer, R_0, to be 304,288,525 and call this the **seed** for our generator. Since every integer from 1 to 2,147,483,646 is produced by the generator in a pseudo-random sequence that ultimately repeats, it does not matter much where we start the process.

By contrast, 'M' and 'A' have in fact been chosen thoughtfully and are not arbitrary values. Most choices of 'M' and 'A' lead to very short sequences.

In the current exercise we start with the above **seed** value for and calculate R_n for 'n' from 1 to 1,000,000. Of course, the generator will produce more than 2000 times as many pseudo-random integers as this; but, we do not want to wait very long for a result. On a Mac II it will take about eleven seconds to run through a million values (as opposed to more than 6 hours to generate all of the sequence).

Why choose this pseudo-random number generator as the basis for testing a MC68881 or MC68882?

First, you can predict the value of the nth integer by an independent and quick direct process. (We will not show how this is done here). The value that the FPU generates for the millionth member of the

sequence depends on all previous values in the sequence. Thus you can have the FPU slog through a million multiplications and divisions with one guaranteed-different operand at each stage. If the millionth result is correct, it is highly likely that all of this computational effort is correct.

Secondly, the multiplications and divisions involve almost the entire width of the registers of the FPU: 31 by 31-bit multiplications and 62-bit dividends divided by 31-bit divisors producing 31-bit remainders. These are nearly the widest such calculations a MC68881 or MC68882 can perform **exactly**. (These two FPUs can execute up to 32 by 32-bit multiplications exactly). A further check on the FPU is thus available: check after the sequence has been calculated to confirm that the **accrued-exception** portion of the floating-point control register (FPCR) indicates that the FPU has never executed an inexact operation since the last time the FPU was initialized.

Finally, it is easy to break up this calculation into 8 sections and exercise all eight FPU registers with 125,000 iterations of the calculation being made with each FPU register in turn holding a portion of the random sequence.

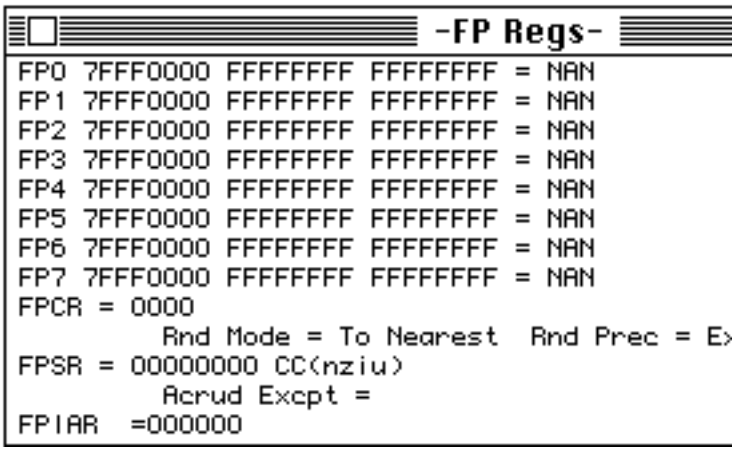
This is by no means a complete test of the functionality of the FPU. But, every register and most data paths are thoroughly exercised.

Now that you know what is to be calculated and why, you can proceed to trace the code.

Start with ***DebugTut*** in control with ***The Debugger*** already launched and waiting to regain control. Select Test **#20: FPU Numerical Exer.** and press the "**Do It!**" button. ***The Debugger*** will gain control in a code block called FPUNUMEX after a DebugStr trap.

The pseudo-random number calculations are accomplished in the subroutine fpuNumEx2. **Step Into** fpuNumEx2 by issuing one **Cmd-?**.

Step Thru (**Cmd-comma**) twice and note that the `FRESTORE` instruction has completely reset the FPU.
Use **Misc:fp Regs/REGS** to open the **-FP Regs-** window. The 8 FPU registers should each hold an IEEE "Not-a-Number", **NAN**. The **accrued-exception** bits should all be cleared, as indicated by the blank to the right of "Acrud Excpt =" in the **-FP Regs-** window.
This is what you should see at this point:



The next instruction: `FMOVE.L #$BEADFADE,FPIAR`

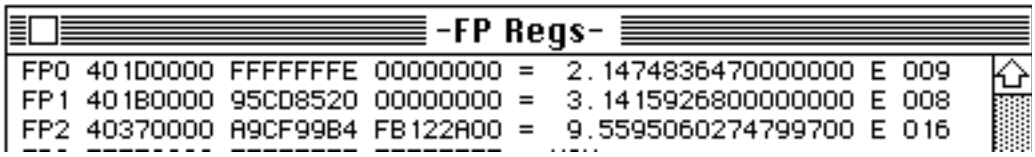
loads the Instruction Address Register (FPIAR) with an arbitrary address that we can readily recognize. **Step Thru** (**Cmd-comma**) this instruction. The least significant 6 hexadecimal digits, "ADFADE" of this address should now be displayed in the **-FP Regs-** window. If all is well, no floating-point exceptions should occur in what we are about to calculate and consequently we should expect to still see "FPIAR =ADFADE" at the end of our calculation. (If a floating-point exception were to occur, the FPU would save the address in the FPIAR of the floating-point instruction that generated the exception).
The next several instructions are most easily explained with the help of a commented source listing:

```
        MOVE.L      #testreps,D2          ; number of repetitions per register
        FMOVE.L     #rndModulus,FP0       ; 2^31-1, our modulus
        FMOVE.L     #multLp,FP1           ; long-period multiplier
        FMOVE.L     #seedStd,FP2          ; set seed value

        ; let us see these in decimal outside the FPU in our globals area ...

        FMOVE.P     FP0,decimal0(A5)      ; 2^31-1, our modulus
        FMOVE.P     FP1,decimal1(A5)      ; long-period multiplier
        FMOVE.P     FP2,decimal2(A5)      ; standard seed for random# generator

        ; loop to generate first 125,000 random integers

@1      FMUL        FP1,FP2
```

```
FMOD        FP0,FP2
SUBQ.L      #1,D2
BNE.S       @1
```

First we set up a loop-repetition count in the CPU register D2. "`testreps` " has been equated to 125,000 decimal. We are going to calculate the millionth integer in the pseudo-random sequence by running 8 similar loops 125,000 times each.

Next we load the three values defining the generator into the FPU. Since each of these values is within the range of a 32-bit representation, we can bring them into the FPU from an immediate, long source:



FP0, FP1, FP2 := the modulus, the multiplier and the seed

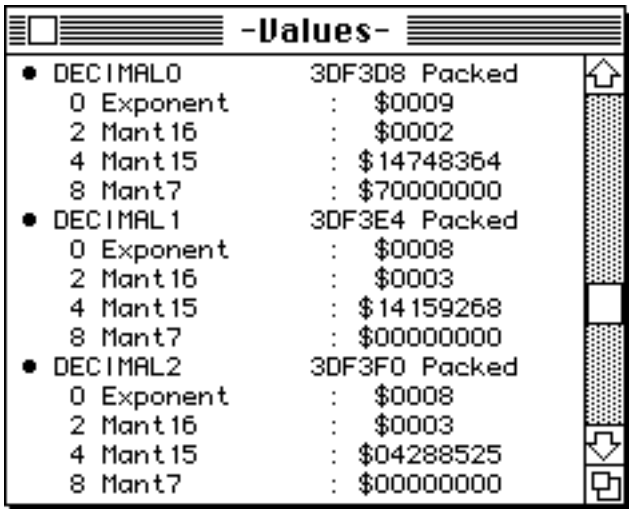
Step Thru (**Cmd-comma**) four times so that FP2 has just been set to the **seed** integer. The **-FP Regs-** window should look like the below (check the contents of FP0, FP1 and FP2 against the definitions above for this generator):

The next three instructions copy the contents of FP0 - FP2 in packed decimal form to 3 records in the global area of ***DebugTut***., `decimal0, decimal1, decimal2` . The file, **DebugTut.dsi**, which was read by ***The Debugger*** at the launch of ***DebugTut*** defines a record type called "Packed" and defines `decimal0, decimal1, decimal2` to be of type "Packed". Here are the relevant lines from **DebugTut.dsi**:

```
=type
    Packed=RECORD
    Exponent:Word;
    Mant16:Word;
    Mant15:HLongInt
;
    Mant7:HLongInt;

    End;

    =Val
    decimal0:Packed
```

A "Packed" record is 12 bytes long, the length of an IEEE packed-decimal floating-point format. If you check the IEEE definition of this format, you will find that the fields of the above record definition correspond in a convenient way to the IEEE exponent and mantissa "nibbles". In other words, let ***The Debugger*** display a BCD value typed as above, and you can read it directly as a decimal value.

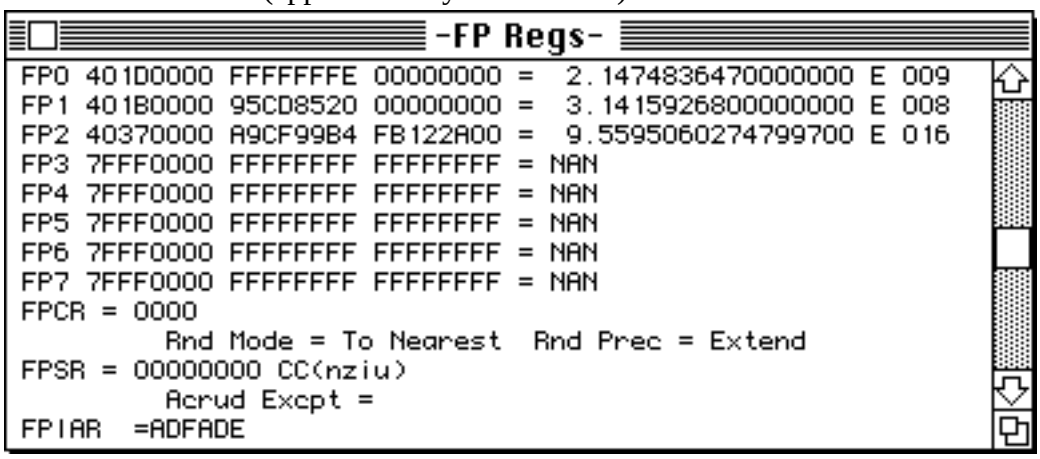
Step Thru (**Cmd-comma**) three times so that the instruction labeled "@1" is to be executed next.. (In the structured disassembly provided by ***The Debugger***, the loop label will be displayed as l*xx*_1, where *xx* is a pair of lower case letters assigned to the fpuNumEx2 code block). Bring up the display of global variables, the **-Values-** window, by **Cmd-u** (must be lower-case 'u'). The three "Packed" replicas of FP0, FP1 and FP2 should be displayed somewhere in the list like this:



At this stage, the random number generator has been initialized. Now, each pass through the four-instruction "@1"-loop should generate a new integer in the pseudo-random sequence with the current

value of R_n being developed in FP2. That is, in the "@1"-loop, the generator is implemented by the multiplication (`FMUL`) by the constant multiplier followed by the reduction "mod M" achieved by the `FMOD` instruction. The two other instructions of the "@1"-loop are CPU instructions to count 125,000 members of the sequence.

Step Thru (**Cmd-comma**) once and the result of the `FMUL` instruction can be seen in the **-FP Regs-** display of the contents of **FP2** (approximately 9.5 x 10^16):

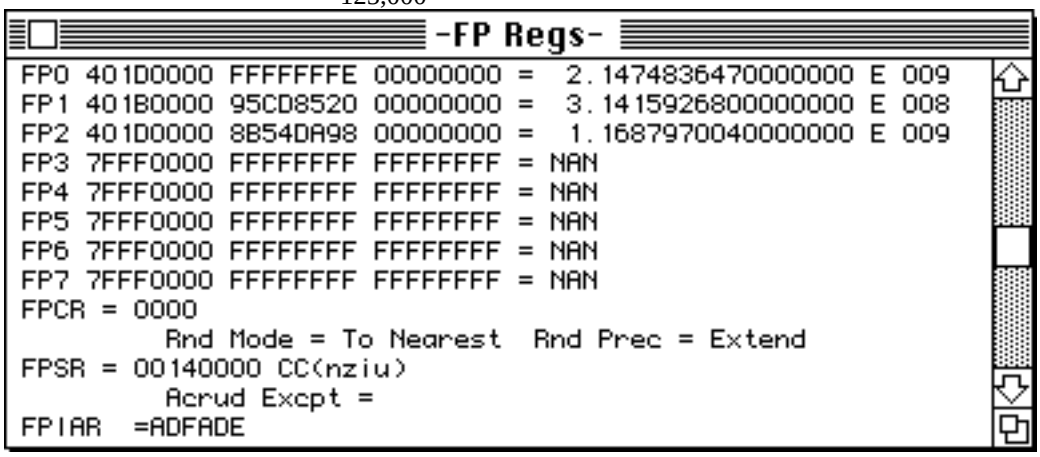


Step Thru (**Cmd-comma**) once again and the result of the FMOD instruction can be seen in the **-FP Regs-** display of the contents of **FP2** (approximately 1.8 x 10^9):
That is, R_1 is precisely 1,822,253,754.

Let us next check $R_{125,000}$. Obviously we want to let the "@1"-loop run its full course. Set a breakpoint at the instruction following `BNE.S @1` , i.e. at:

```
MOVE.L  #testreps,D2
```

Highlight the above line on the fpuNumEx2 window and issue a **Cmd-B**. This line then will be marked with a bullet indicating a breakpoint. Give control to ***DebugTut*** via **Cmd-E**. The loop will take a little more than one second to execute. $R_{125,000}$ can be seen to be 1,168,797,004 as seen here:



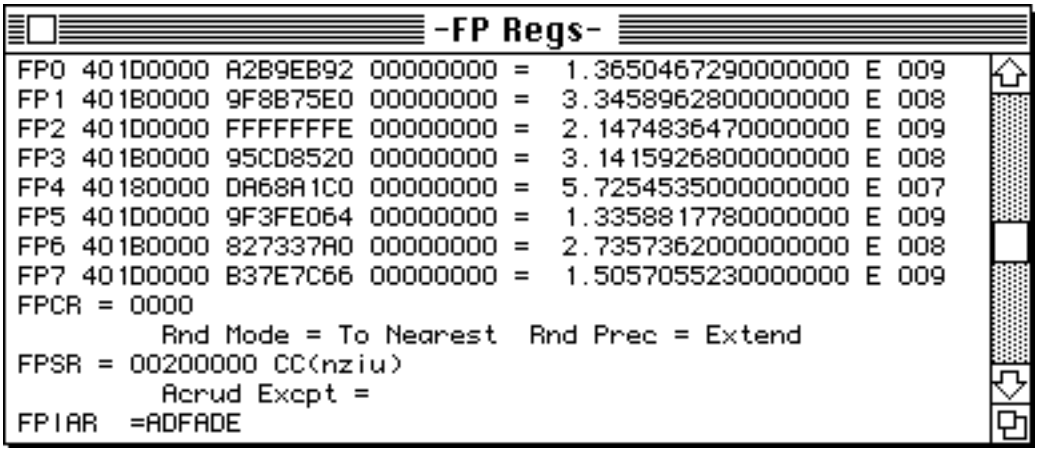
This value is indeed correct as verified by independent, direct means. Also, note that the field "`Acrud Excpt =` " is still blank, indicating that all FPU operations thus far have been exact.

This would be a good time to scan down the fpuNumEx2 window and note that there are 7 more loops very similar to the "@1"-loop. Each loop gets the result of the previous loop as its "seed". Each loop is set to execute 125,000 times and generates its portion of the random sequence in a different floating-point register. After the last of these loops, @8 (i.e. l*xx*_8) has run to completion, $R_{1,000,000}$ will repose in FP1.

When the @8-loop reaches completion, the next instruction to execute is in the next code block, labeled fpNumDone. Set a breakpoint at fpNumDone by highlighting fpNumDone and issuing a **Cmd-B**.

To generate $R_{1,000,000}$, issue a **Cmd-E**. The seven remaining loops will execute in sequence in a total time of 10 seconds. Then ***The Debugger*** should regain control via the breakpoint we just set at fpNumDone.

$R_{1,000,000}$ is contained in FP1: 334,589,62:

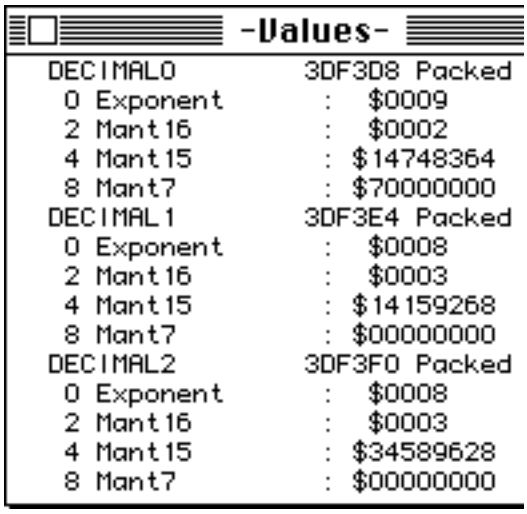


The first three instructions of fpNumDone move:

M	from	FP2	to	global “decimal0”
A	from	FP3	to	global “decimal1”
$R_{1,000,000}$	from	FP1	to	global “decimal2”

Step Thru (**Cmd-comma**) three times so that the above moves are just completed.

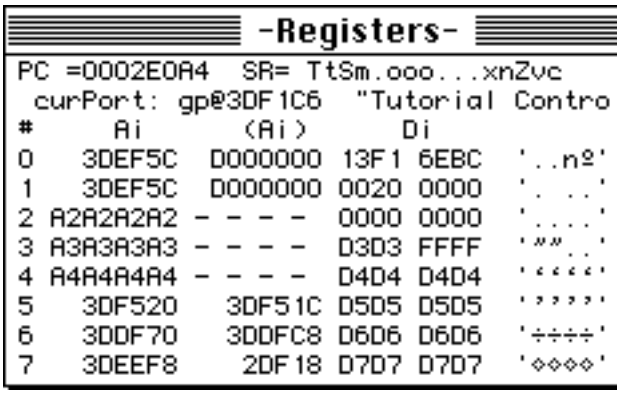
Update the display of the global variables, the **-Values-** window, by **Cmd-u** (must be lower-case ‘u’). The three “Packed” replicas of FP0, FP1 and FP2 should now appear like this:



The next two instructions move $R_{1,000,000}$ from the FPU register FP1 to the CPU register D0 and then test this value against the expected result (shown below in hexadecimal):

```
FMOVE.L     FP1,D0          ; result to CPU
CMP.L       #$13F16EBC,D0   ; check for correct
```

Step Thru (**Cmd-comma**) twice so that the above instructions are just completed. Then the **-Registers-** window should look like this. D0 should contain $13F16EBC. The “**Z**”-flag should show in upper case, indicating that the above CMP.L verified that the expected result was in fact obtained:



Return control to ***DebugTut*** by issuing a **Cmd-E**.

Technical Reference

Data Type Abbreviations (`"TT@nnnn"` in the Heap display and other places)

al	ALRT resource	mn	MENU resource
ap	AppParms	mp	Master pointers
c	CODE resource	pk	PACK resource
cn	CNTL resource	pn	PAT# resource
cr	ControlRecord	pt	PAT resource
cu	CURS resource	rg	Region
dc	DCtlEntry	rm	ResourceMap
df	CDEF, MDEF, or WDEF resource	sc	Scrap
dg	DLOG resource	sn	STR# resource
dl	DITL resource	st	STR resource
dr	DialogRecord	sv	T_SonyVar
dr	DRVR resource	te	TEREC
fc	T_FCBRec	ts	TEScrap
fd	FOND resource	tx	Text
ft	FONT resource	ut	uTable
gp	GrafPort	vc	VCB
ic	ICON resource	wi	WIND resource
in	ICN# resource	wr	WindowRecord
ml	MenuList	rd	RoutineDescriptor

The Debugger's Table of executable Resources

When one of the below resource types is referenced in a "GetResource, ..." request, *The Debugger* will check for the presence of an aux file in the same folder, and if present will form symbol tables for the resource. Use the "=G" dsi command (page 34) to specify a resource type that **may not** be in this table.

If the resource type is in the table, The Debugger will tell you that it is a standard type via '=G'. You may view it's table of names by looking at the '=G Rsrc Table @nn' in the '-Dbgr Status-' window.

4DEX, 4DPX	4th Dimension	MODU	4th Dimension
CDEF	Control defProc	NBPC	AppleTalk Name-Binding Protocol
CACH	File system cache	PDEF	Printing defProc
DRVR	Driver.	PROC	Generic CODE Rsrc
DSAT	DS Alert code.	PATC	Patches
FKEY	Shift-Cmd code.	RSSC	ResEdit Extensions
FMTR	Formatting code.	SERD	RAM serial driver
LDEF	List defProc.	WDEF	Window defProc
MBDF, MDEF	Menu defProc.	XCMD, XFCN	Hypercard external
cdev	Control device	8Bxx	PhotoShop Plug in modules

and the following Apple File Exchange types:
4NFS, MCMC, MCMS, MCPD, MSMS, MSMC, MSPD, tcod, tprc, PDMC, PDPD, PDMS

Register-Based Trap Call Symbols (Trap argument lists)

In the following form descriptions: *reg_id* represents a 680x0 register descriptor: A0, D3, etc.
var_id represents a variable name and
type_id represents a Macintosh type name

form	indicates	example
reg_id | *var_id : type_id*	an input/output (VAR) parameter	A0|IOPB:ParmBlkPtr
reg_id /| *var_id : type_id*	an input parameter	D0/Count:Integer
reg_id \ *var_id : type_id*	an output parameter	D1\Size:Integer
reg_id \ *type_id*	a function result	D0\OSErr

Macros in the ASM windows

The below macro names are used in ASM windows. Their definitions are in the file **Amacs**. POP and PUSH are used in the code portions of an ASM window.

`POP.S xx` is equivalent to `MOVE.S (A7)+,xx`
and `PUSH.S xx` is equivalent to `MOVE.S XX,-(A7) where S = B OR W OR L`

The following macros are used to make clear the format of DATA areas:

```
STR       Pascal String (Leading Length Byte)
WSTR      Pascal String, Word Aligned
ZSTR      C style String (Trailing zero Byte)
WZSTR     C style String, Word Aligned
DZS.s     blocks of storage that are initially zero.
DNAME     Embedded Debugger (or 'Macsbug') Name
DRVHD     setup the header words for a DRVR
DrvAdd    Define addr of processor within the DRVR
QUAL      Constrain scope of local names to current procedure
VEQU      define A6 offset of a Local name
VEND      end of local name definitions
JBIAS     define Jump table bias
ADDR      16 bit address of a label
ADDRL     32 bit address of label
JUMP      Jump table entry for rel jumps generated by Case stmts
CJMP      and one for Think C V3, V4
```

Basic Data Types

The below "**base types**" are the basis for the display of data types in *The Debugger*. See **Tables:Add/Zap Type Defs** for a description of the syntax of Type declarations.

Type	Size in Bytes	Displays as	Comments
BOOLEAN	1	TRUE or FALSE	
Byte	1	0 to 255	Unsigned Byte
SignedByte	1	-128 to 127	Signed Byte
ASCII	1	an Ascii Char	PACKED Array[x] of CHAR
String	2 - 256	string	Pascal String
Cstring	2 - 256	string	C String - Array of UnSigned Byte
Ptr	4	Hex address	blind pointer)
ProcPtr	4	ProcName	
Handle	4	^^hhhhhh	blind handle
INTEGER	2	decimal number	Signed integer
Word	2	Hex number	Unsigned integer
LongInt	4	Decimal	
HLongInt	4	Hex number	
DateInSec	4	mm/dd/yy HH:MM:SS	
CHAR	2	Pascal Char	Char in low order byte
OSErr	2	+-ddd comment	Mac Error Codes
SCSIErr	2	SCSI Error Code	
RefNum	2	$hhhh -> fileName	FCB refNum
VRefNum	2	$hhhh	Vol RefNum
ProcOffset	2	ProcName	DRVR headers only
Real	4	N.NNNN E+-nNN	SANE type
Double	8	N.NNNN E+-nNN	SANE type
Extended	12	N.NNNN E+-NNN	881/2 Format
Extend10	10	N.NNNN E+-NNN	SANE software Format
TrapWord	2	_trapName	Mac trap Word $Axxx
JT_Off	2	procedure name	index into CODE 0 Jump table
Class_ID	2	MacApp Class Name	
Tobject	2	Tobject@nnn ->	ClassName_@nnn (MacApp object)

As the type definition syntax (see page 75) does not include the "PACKED" attribute, one declares a packed array of CHAR as an "`ARRAY[n..m] OF ASCII `".

Tasks

To ***The Debugger***, a task is an application (APPL), a loaded CFM container or an executable resource such as DRVR, INIT, FKEY, XCMD etc. The "Task & File Info" window **[Misc:Task & File info]** displays the list of tasks that ***The Debugger*** knows about. Most of the 'task' relative windows are prefixed with the task name. Others are the ".dsi" file and Heap Zone windows associated with the task.

The current task is displayed in the menu bar in place of the "Windows" menu title. Use the "Use Task" **[Misc:Use Task ...]** command to switch task contexts, or select a task relative window of the task you want to reference. eg. the "-Code Blks-" window or a structured assembly window for the task.

When a task terminates, ***The Debugger*** removes windows and information associated with the task from its displays and tables.

Debugger Modes

The Debugger is capable of running in two modes, "***RunDbgr***" mode and "***Init***" or **background** mode.

"***RunDbgr***" mode (**System 6 only**) is primarily designed to debug a single application, and is useful in limited memory environments. ***The Debugger*** is still capable of debugging "many" tasks in this mode, such as CODE resources, DAs invoked from the Problem Program etc.

"***Init***" mode of operation is when ***The Debugger*** loads at INIT time. In this mode ***The Debugger*** will be launched at INIT time via the "xDbgr_Startup" INIT . It will not be affected by patches to the traps that are set by INITs that come after it because it runs in a "world" (trap vectors) that is a copy of the "world" as it existed when it started up.

In "***Init***" mode, ***The Debugger*** will debug any type of task that it knows about, providing there is an associated auxiliary file (.map, .snt, .SYM, .xSym).

Associated File Names for Applications and Code Resources

The Debugger relies on symbol information to translate addresses to symbols. This information may reside **in any one** of the following:

- a ***MacNosy***–style **..snt** file.
- an MDS or MPW–style **.map** file (`-L >xxx.map` option on LINK directive for MPW)
- an V3.x style **..SYM or .xSym** file (an Apple specification)

one may also specify additional information in a **.dsi** file.

The names of all the associated files are formed by adding the appropriate suffix to the PP name. In the case that the PP is a resource, then then one must also append the resource ID.

More formally, if ***ApplName*** is the name of an Application, then the Aux files are named, ***ApplName*****/CODE.snt**, ***ApplName*****.map**, ***ApplName*****.SYM**, and ***ApplName*****.dsi** is the .dsi name.

The aux files associated with a resource of type **RSRC** and id **hexID**, that will be **loaded from** the file ***RsrcFile*** are: ***RsrcFile/*****RSRC_hexID.**snt , ***RsrcFile/*****RSRC_hexID.**map , ***RsrcFile/*****RSRC_hexID.**SYM and ***RsrcFile/*****RSRC_hexID.**dsi is the dsi file name. **hexID** is the resource ID in Hexadecimal with leading zeros stripped.

examples are: "System/DRVR_F.snt" or "MyDAfile/DRVR_C.snt" or "Mouse/cdev_F020..SYM"

NOTE: The PP and its associated files **must be in the same folder that the APPL or resource is loaded from**.

NOTE: Turning on the debugger flag **Doc_Msgs** will cause the expected 'sym' file names to be displayed in the '-Notes-' window (`NO Aux file: 'expected sym file name'`)

Symantec's Think C™ V4, - V7 project files

ThinkC™ project files contain symbol information about the addresses of procedures and global variables in its ZONE resource, which The Debugger reads when a "project" file is launched, so **no aux file is necessary**. You may supply a .dsi file to supplement the information in the project file.

For information on debugging DA's cdev's ,etc built with Think C, read Debugger Tech Note #5.

Using MacNosy .snt Files

It is best to work with a .map or .SYM file that you generate from the source code for the PP.
But you will need to apply ***MacNosy*** to generate an .snt file when:

1. you do not have the source
2. you are compiling with a product (Turbo or Think Pascal) that doesn't generate a suitable file
3. the extra cross-reference information produced by ***MacNosy*** is useful to you.
   (See the Display-menu section of this manual for details of how to access this information).

If you need to run the PP through ***MacNosy*** to obtain a .snt file, I suggest that you **DON'T** spend any time 'reviewing data'. ***The Debugger*** will change a block mislabeled as data to code when a procedure in that block is executed.

If you follow this advice, you DO lose the ability to initially see the names of such hidden procedures and to set breakpoints in them. But there is a workaround that gets everybody to show up: force all procedures into the inter–segment jump table by creating a dummy segment which references all program procedures.

Here's a Pascal example of how to do this:

In the Problem Program's main segment, include code like this:

```
dummyVar := 0;
if dummyVar <> 0 then dummyProc;        {a proc call that's never executed}
```

Then, in a dummy segment, include code like this :

```
{ declare all procedures }
procedure abc; external; { no need for argument lists }
procedure bcd; external;
{ and so on … }

{this next procedure never gets executed}
procedure dummyProc;
begin
{calls to all procedures, forcing them to the intersegment jump table}
   abc;  bcd;  { and so on … }
end;
```

.

dsi files and commands (Debugger Startup Info)

After ***The Debugger*** forms the tables for a task, it will look for a "**.dsi**" file in the same folder as the task. ***The Debugger*** will open the **.dsi** file in a window, and execute commands in it. Using a **.dsi** file will enable you to initialize ***The Debugger*** with source level information from your PP such as the **ArgList Defs**, **User Type Defs**, etc. You may inhibit **.dsi** file processing by holding down the mouse button.
For an example look at the **ROM.dsi** file in "Dbgr/Nosy files".

A ".dsi" file consists of a set of commands followed by zero or more lines specific to the command. Lines after the first "=End" line are not processed.

A "dsi" command begins with a "=Keyword" **starting in column 1**.
The rest of the command line is ignored, and may be used for comments.

Only the first letter of the Keyword is used to determine it.

The commands that may appear in this file are:

=- (equals minus) - ".dsi" file comments
followed by a block of lines (no equals signs in column 1)

=Type - specify Type information - See Debugger Tech Note #1 for more information
followed by a group of **User Type Defs** (Pascal syntax type definitions)
`'=T,Redef'` allows you to re-define existing users types.

=Val - Associate global variables with their types for display in **"-Values-"**
followed by a group of "**globName:Type;**" Lines to be put in the Values Window.

=AddMenu - Add a Debugger Script to the 'task name' menu
See the file 'Dbgr Scripts' for examples:

```
=A
'menu_Name/CmdKey'
lines of the script (action clause) to be executed
```

To delete an AddMenu one hiltes ONLY the =A and addmenu name lines.

=Bkpt {,Cond_num or "*" for the current CondNum }
followed by a list of Breakpoint addresses (procName+Offset)
"**=Bkpt,***" will attach the last defined (current) Bkpt condition to the list of Breakpoints.

=Cbkpt - Define an "action clause" for use in conditional Bkpts
followed by an action clause

=Define- Define Function key Equivalences

```
KeyNum=Letter
```

Binds the function key F'keynum' to Command-Letter. If Letter is Uppercase then F'keynum' is bound to 'shift-Command-Letter.

=Flags - Set Values of Debugger flags - **See Debugger Tech Note #3 for more information**
followed by assignments of the form "flagName = 0 or 1 ;"
Sample lines (taken from ROM.dsi) are :

```
TBL_SLOP = 0;  { NO 2K of slop in Document Tbl Area for Typ/Arglist additions}
CODE_Blks = 0; { automatically bring up a "-Code Blks-" window for a Task }
CPlus = 1;     { Demangle C++ names }
Inst1_Bkpt = 1;{ Courtesy Bkpt is on }
dsi_CLOSE = 0; { =1 to continue & close .dsi windows ]
      { =0 to stop processing the current .dsi file and leave the window open }
```

=Generic - specify generic CODE resource type to be debugged
followed by a line with a Resource type in columns 1 to 4

=Int - Set Trap intercept (Cmd-J)
followed by a single line with a trapname or range (trapNameA - trapNameB)

=Library - specify CFM LIbrary and 'sym' file name to be debugged (**See DTN #12**)

```
=L
MyCFM_LibName          Case Sensitive
path:SymfileName
```

=Menu - Toggle CheckMark Menu items
followed by lines of the form + or - `MenuNumber,MenuItemNumber {,newCmdKey}`
A "+" sign selects the checkMark item and a minus sign de-selects it.
If the 2ond comma is present, then the character following it becomes the new Cmd key equivalent.

A Sample line is:

```
-7,16 ; Turn off the UnLoadseg Error check Menu item
```

The following 2 lines interchanges Cmd-W and Cmd K :

```
2,3,W  ; CHANGE 'File:Close' cmd letter to 'W'
7,4,k  ; CHANGE  'add/del to Values' cmd-letter to 'k'
```

=Notes - copy lines to the "-Notes-" window
followed by lines of text (no equals signs in column 1)

=Open - Open a text file in a window
followed by a single line with the **full pathname** of a text file. add '`,EX`' to execute as a '.dsi' file and '`,CLO`' to clse the window after processing

i.e. `myDisk:B:C,EX,CLO`

=Proc - define procedure argument lists
followed by a group of Pascal Syntax Procedure/Function declarations.
The proc/func declaration must fit on a single line with NO leading spaces..
The syntax of the arg List is the same as that in the "Tables:OS Traps" window
The format of the line is: "`PROCEDURE procName(arg:typeName; ...);`"

=ROMI
followed by lines with:
1) the addresses of Trap calls for Discipline to ignore. You obtain the address's by selecting the contents of the "**-Show Ignores-**" Window and pasting it into a ".dsi" file
2) a typeName prefixed by a minus which you wish to omit Discipline for (-Char, …)
3) a "_Trapname' prefixed by a minus which you wish to omit Discipline for (-_Open)

=Source - Set Source paths for .SYM file - **See Debugger Tech Note #3 for more information**
followed by one or more lines with **pathnames to the folders** containing the Source files

```
"HardDisk:MPW:MySource ƒ:"
```

=U - Specify MMU Protection flags & Range Info

=X - Specify types for the 'View As' PopUp Menu in 'Type_@nnn' windows
Followed one or more lines: with a type name or a list of type names seperated by commas.
Example:

```
=X
WindowRecord,RoutineDescriptor
Double
```

=Z - Specify 'dcmd' rsrc files to Open at Debugger INIT time.
Followed by a list of file names (full path names if NOT in 'Dbgr/Nosy files').

=End - marks the end of the file, lines following the "=End" are **NOT processed.**

Development System Notes

All Systems: File Names

The names of applications, Think C projects and code resource files must be 21 characters or less.

Source Level Debugging

- For PP's that have a ".SYM" file or Think C projects compiled with "Use Debugger", The Debugger will display the source code for a procedure in a window. The beginning of source statements is marked with '∑" signs in column 1. A Bullet will appear in column 2 of lines that contain an active breakpoint.

  To set or clear a breakpoint in a source window, highlight a line containing a '∑" and **Cmd-B** OR move the cursor to the left edge of the window until it turns into a 'stop sign', then you can click on the mouse to set or clear a breakpoint at the line. Holding down the shift key sets a conditinal Bkpt.

  One may toggle between a source only display and a display of source and interspersed assembly code by holding down the command key and clicking in the content region of the window (**Cmd-click**).

  The source level information in the MPW ".SYM" file is relatively complete. It contains information about the name, type and address of both global and local variables that is used by *The Debugger*.

  For MPW C, the type information for character strings is incomplete, and *The Debugger* tries to make an intelligent guess at types that look like strings. As an alternative you may **typecast** the values of global variables when viewing them in the "-Values-" window by using the commands

  "`varName:string` " for Pascal strings, and "`varName:Cstring` " for C type strings.

  Be aware that *The Debugger* is presently implemented **with flat name spaces** for procedures, and types. The effect of this is that Pascal programmers who use a name more than once in inner procedures will not be able to view the second occurrence of the procedure by name.

  For Macintosh types , which *The Debugger* already has definitions of, it **ignores** the type declaration passed to it in the ".SYM" file, so you should not redefine any of the standard IM types (Rect, etc).

  To maximize the substitution of field offsets in the ASM displays it is suggested that Pascal programmers change the file "QuickDraw.p" as follows:

  1) Redefine WindowPtr

  ```
  WindowPtr = ^WindowRecord;  { ptr to a Window Record, not a grafPort }
  ```

  2) Adjust the definitions of the procedures that will be affected by it by changing the parameter list declaration in `OpenPort, InitPort, SetPort and GetPort to "port:UNIV GrafPtr`

  C programmers can make similar changes to the file "QuickDraw.h".

  To de-mangle C++ names, set The Debugger flag "**CPlus** to 1 prior to launching a C++ program.

  One can not reference the names of local variables in arithmetic expressions. You may force a reference to a global variable to be relative to a task in an expression by writing

  "`∂taskName.varName` ".

  One is advised to **use the "Set PC" command at the Source level with care** as the compiler may have generated code that initializes registers in out of the way places, etc.

  MPW users: Use the =S command in a ".dsi" file to set the paths to the source code. See Tech Note #3.

  Beware of differences in the Pascal call to trapnames and the ASM one that Nosy/Dbgr know about. An example is: Pascal `SetCtlMax` call versus the ASM `_SetMaxCtl trap`.

MPW Pascal: Suggested Compiler Switch Usage

The Debugger needs names. If you are not using MPW's symbolic debugging capability, you will need to set compiler switches in MPW to get them.

Here's a Pascal template showing suggested usage for UNITs. For PROGRAMs, drop the {$Z-} switch.

```
{$D-} {$N+} {$Z+}
UNIT someunit ;

INTERFACE
{global variable declarations}
IMPLEMENTATION
{$Z-}
{procedures and functions in the UNIT}
END.
```

To produce the '**.map'** file, use the following command template:

```
Link -L >Example.map  Example.p.o ...## -L option for old style map file !!
```

If you wish to use '**.SYM**' files, a typical 'compile' and Link commands would be:

```
Pascal -sym ON Example.p
Link   -sym ON Example.p.   O ...
```

Avoid "spurious" out of date messages from The Debugger, make sure that the last operation in your build "**touches**" the ".SYM" file by adding a line:

```
SetFile -m myApp,SYM
```

Remember to **rename the ".SYM" file for resources**, and to move it to the folder that the resource will be executed from.

Think's C™ V3 - V7: Misc Notes

In Think C's Options menu, turn off the generation of Macsbug symbols to save space in the code. *The Debugger* will read the procedure names and global symbol names from the project automatically. To obtain, source level debugging, compile with "**Use Debugger**" on, and clear it before you "Run" your project..

If the Dbgr flag DBG_THC is OFF, then The Debugger will NOT debug your project file.

Think C: Reading Symbol Tables (for System 6 Users on Mac's with less than 4 Meg of RAM).

The Debugger is able to read the symbol table in a Think C project file. If you take the following steps, you will minimize the time needed to transfer from Think C to *The Debugger* in ***RunDbgr mode***.

- Place a COPY of ***RunDbgr*** in the folder containing your project file and prefix it with a blank, so it appears first in the list of names in a Pack3 file select dialog.
- Use ResEdit to edit Think C's **File** menu so the **Transfer** command can be invoked by **Cmd–T**.
- When you want to use ***The Debugger*** on your Think C project, invoke Think C's **File:Transfer** command by pressing **Cmd–T**, and select ***RunDbgr***.
- In ***The Debugger***'s startup dialog, use the **Set/Clear Startup** button to select the project file as the PP

Think C: Validity Of Global Symbols

Think C does code relocation at execution time. This means ***The Debugger***'s disassembly windows won't be able to display valid references to the PP's global symbols until a procedure's segment has been loaded.

Think C: Source Level Debugging of INITs and Executable Resources

See Debugger Tech Note #5 for a complete description.

Expressions and Conditional Breakpoints ("Action clauses")

Both ***Nosy*** and ***The Debugger*** contain a powerful expression evaluator based upon the syntax of **Pascal**. Differences from Pascal syntax are noted below.

The evaluator gives you access to:

- constants (**default radix is Hexadecimal**)
- user global variables, system global variables (the names in the Sys Syms window)
- typecasting via types known to ***The Debugger*** (**Tables:Shift-Record/ALL names** display)
- Debugger variables and functions which are prefixed by a '?".

Numeric constants are interpreted as Hexadecimal unless they are preceded by a "#" sign. To force a constant to Hex, precede it by a "$" sign. **Note that any change of radix persists until explicitly changed again.** Precede hexadecimal constants that start with 'A' thru 'F' with a '0'.

Address versus Value of

The occurrence of a variable XX in an expression represents the value of XX and "@XX" is its address.

Task Names in expressions

One may prefix a task name to a procedure or variable name in an expression with the syntax:
`∂taskName.XX` OR `∂'taskName'.XX` where `XX` is the name of a procedure or variable.
The occurrence of `∂taskName` will cause a task switch that will persist until the next occurrence of a `∂taskName .`

Expressions

All arithmetic is done in 32-bit signed mode. The operators are a subset of the MPW Pascal operators and functions. They are:

+ for addition **-** for subtraction or negation ***** for multiplication **/** for divide (DIV)
| for bitwise OR **&** for bitwise AND **¬** for bitwise negation (logical NOT)

= **<>** **<** **>** **<=** **>=** are the relational operators which return a zero for FALSE and one for TRUE

. (period) is used for selecting a field in a record structure (x.top)

^ dereferences an expression interpreted as an address to a value (eg theZone^.MoreMast)

@ is the **Address-of** operator (@symName , @procName , @_TrapName)

Debugger Expression	**Equivalent Pascal**
MOD(a,b)	"a MOD b"
BXOR(a,b)	"BXOR(a,b)"
BSL(expr,**decnum**)	"BSL(expr,decnum)"
BSR(expr,**decnum**)	"BSL(expr,decnum)"
BTST(expr,**decnum**)	"BTST (expr,decnum)"
Ord(expr)	"Ord(expr)"

where**:**
decnum is the default decimal radix
Chr(expr) returns the low 8 bits as a Longint value **without sign extension**
Byte(expr) returns the low 8 bits as a **sign-extended** Longint value
Typename(expr) coerces expr to typename so that other operators can be applied to it
(eg WindowRecord(RA0).dataHandle^^)
() are the parenthesis used to control the order of evaluation of complex expressions
Registers A0 to A7 and D0 to D7 may be referenced by the name RXi, where X is A or D, and i is 0 to 7. RXi.U or RXi.L designate upper and lower halves of the registers

Write Statements

Write(**list**)	writes a list of print_expression values to the **-Notes-** window
Writeln(**list**)	writes a list of print_expression values to the **-Notes-** window followed by a **newLine**
WR_Entry	outputs the string "cBkpt @ nnnn Procname+nn" to the **-Notes-** window

list is sequence of one or more print_expressions separated by commas
print_expression is of the form: expression{ :Typename } or "?Rec(typeName,addr)"

When an expression appears without a Typename it is displayed as Hex Longint, otherwise it is displayed according to Typename. Some examples are:

theZone	is displayed as	" $001E00"
theZone:Thz	is displayed as	" ^Zone_@01E00"
theZone:Longint	is displayed as	" 7680"
WindowList^.TitleHandle:StringHandle	is displayed as	its String value

Pre-defined Debugger Variables

?Bkpt	- number of times the current Breakpoint has been executed since it was set
?PC	- value of the Program Counter
?SR	- value of the Status Register
?Trap_Num	- trap number of the instruction at ?PC if it is a _Trap
?Len_Globals	- length of User's Global Area (useful for Save/Restore state)
?LR	- value of Link Register (PowerPC)
?CR	- value of Condition Register (PowerPC)
?XER	- value of Integer Exception Register (PowerPC)
?CTR	- value of CTR Register (PowerPC)
?SP	- value of Stack Pointer (A7 or R1)
?mode	- Mode that the user program is in on entry to The Debugger (0 = 68K, 1 = PPC)

Statements and Compound Statements

Assignments have the syntax:

?DbgrVarName := expr; **OR** RegName := expr; **OR** sysGlob := expr;

There is an implicit **BEGIN** ... **END** pair around the outermost block (group) of statements. BEGIN.. END pairs may be used to form compound statements as necessary in other contexts. A semicolon is used to separate statements.

Debugger Statements and Procedures

Pause	- Enter ***The Debugger*** (Default action at the end of a condBkpt)
Resume	- Return to the PP (rest of condBkpt is not evaluated)
?DelBlk(fba:Longint)	- deletes the block that was previously saved by a **?SaveBlk()**
?RestoreRegs(fba:Longint)	- restores the values of a saved register set, saved by **?SaveRegs()**
?DisAsm(addr,mode)	- disassembles the instruction at addr, line to -Notes-.
?Rec(TypeName,addr:Longint)	- used in Write statements to "display" data at addr as "TypeName".
?Redirect('WindowName',xx);	- Redirect output from '-Notes-' to WindowName
?Redirect;	- Redirect output back to '-Notes-'
?dcmd(Name,'arglist',file)	;- Execute Macsbug style 'dcmd' (See DTN 11)
?UserC(procName);	- Execute proc 'procname' in user's program (See DTN 11)
?ProcCall(UserProcName, arg1, ... , argN);	- call a User Proc
?BlockMove(source,dest,size);	-move 'size' bytes from source to dest
?Stack(fba,stk_top,mode);	- Display stack starting at 'fba' and going to 'stk_top' . Initial mode is 'mode'. If stk_top = 0 then stk_top = fba + $0C00 A good value of mode is the pseudo variable '?mode'

Debugger functions

?SaveBlk(FBA:Longint; len:integer):Longint; - Saves a block of memory **len** bytes long beginning at FBA and returns the address where the block is saved. The block is saved in ***The Debugger***'s heap zone. **len** must be less than 30001 bytes.. Example: ?myBlk := ?saveblk($200,40)

?CmpBlk(fba:Longint; len:integer; fba2:Longint):Boolean; - compares two blocks of memory for equality and returns TRUE (1) if they are equal.

?SaveRegs():Longint; - saves the current value of the address and data registers in a 64-byte block

?Heap_Check(Zone_addr:Thz):Boolean; - Check the Heap at Zone_addr for consistency Return TRUE if OK, else post message in **-Notes-** and return FALSE.

?Heap_Scramble(Zone_addr:Thz):Boolean; - Scramble the Heap. Return TRUE. Checks the heap for consistency first.

?Compact_Heap():Boolean; - Executes _CompactMem($80 0000)

?Purge_Heap():Boolean; - Executes _PurgeMem($80 0000)

?Trap_NumOf(TrapName):integer; - returns the "Trap_Num" of a given MacTrap name. To be used in conjunction with the Dbgr variable "?Trap_Num".

?QueryUser(msg:String):Boolean; - Puts up an Alert with the msg, and YES and NO buttons. Returns TRUE if the User clicked in the YES button.

?Userp(procname):Boolean; - Calls a user-defined function that resides in the PP's Heap Zone. The function should reside in CODE segment 1. (It must always be loaded). It has the declaration:

```
procname( SysGlob:Ptr;  USR_PC:Longint;
                Regs:RECORD D0, ... D7, A0 ... A7:Longint END  ):Boolean;
```

The register A5 is set to point to the PP's Global area. SysGlob is a pointer to the System Global area which normally starts at location zero. USR_PC is the value of the Program Counter.

Do not attempt to do any I/O or segment loading during the execution of a Userp.

If Statements

IF relexpr **THEN** T_statement { **ELSE** F_statement }

If relexpr is **non-zero** then the T_ statement is executed, otherwise the F_ statement is executed.

Some examples of the use of IF statements in conditional Breakpoints are:

```
IF (MOD(?BKPT,4) = 0) & (theZone <> SysZone) THEN BEGIN
   WR_Entry;  WriteLn(theZone:Thz);
END;
RESUME;
```

i.e.: Every fourth time the breakpoint is executed, test if the value of theZone is **not** equal to the value of SysZone. If the test is TRUE then ***The Debugger*** will take control and display the location of the Breakpoint and the current value of "theZone"in the **-Notes-** window.
The Resume-statement will then cause ***The Debugger*** to transfer control back to the Problem Program.

```
IF  ?BKPT  <= #50 THEN BEGIN  WriteLn('param = ',ptr(RA6+8)^:integer);  RESUME; END;
```

i.e.: The first 50 times this breakpoint is executed, the string 'param = ' and its value will be printed on a line and then the PP will resume execution. On the 51st entry the PP will pass control to ***The Debugger***, no printing will occur and execution of the PP will not resume.

An alternate way of writing this is:

```
WriteLn('param = ',ptr(RA6+8)^:integer);   IF  ?BKPT  <= #50 THEN RESUME;
```

You may reference variables local to the procedure in a similar way. That is if x is located 8 bytes below A6, then the address of x is "RA6-8" and the value of x may be displayed by the print expression:
"`ptr(RA6-8)^:type `"

While, Repeat-Until loops and CYCLE and LEAVE statements

`WHILE` and `REPEAT - UNTIL` loops are implemented just as they are in Pascal.
You may nest loops up to 8 levels deep.
A `CYCLE` statement will cause a branch to the end of the current loop and a `LEAVE` statement will branch to the statement following the end of the loop.

'Dbgr Scripts' contains a number of examples of WHILE loops. FOR loops are not implemented.

Output Re-direction

Normally commands, write statements in 'action clauses', etc put their output at the end of the '-Notes-' window. The ?ReDirect command lets one define an alternate window for such output. The syntax is:

`?ReDirect('windowName' {,Append} )` ; - If 'windowName' exists then bring it to the front, else create a window with a name 'windowName' . If the Append parameter is present, then output will be appended to the end of the text in the window, else the text will be deleted and the window drawn as empty the next time it is written to.
`?ReDirect;` - re-directs output to the '-Notes-' window in Append mode.

Note: Unlike MPW, The Debugger always redirects output to the end of text in the redirection window.
Note: If the shift key is held down when an Addmenu is executed, then ?ReDirect('wName') will always create a new window with the name 'wName nn' (nn - 1, 2, ...).

Array References

You may reference the elements of a 1-dimensional array via the notation: `ArName[subScript_expr]` .
If ArName is a pointer to an Array, then one writes: `ArName^[subScript_expr]` and so on.
The address is calculated as:
AddrOf(ArName) + sizeOf_Arname_element * (subScript_expr - Lower_bound)

which is just what you would expect. For an example of it's use see the 'Files' script in 'Dbgr Scripts'.
You can reference any of your global variables in your program during the execution of a script.
The way to display an element of an array of `Extended` elements is: `write(myArray[n]:Extended);`

The ?get_value function

When the ?get_value function is executed from inside an AddMenu script it returns the hexadecimal value that the user entered, or was hilited in the front window. Its syntax is:

```
Function ?get_Value('prompt_string'):Longint;
An example of it's use is:  ?fba := ?get_Value('enter 1st byte addr to dump');
```

The Mark Facility (what procedures have I executed)

This facility consists of three procedures and a function "isMarked" which work in conjunction with the **Stops:Proc Entry/Exit Trace** command. The facility can be used to determine what procedures are called by a given procedure or set of procedures (for QA testing, etc).

First one executes a **?clrMarks;** statement with **Cmd-shift-[** to clear the mark bits in the internal block tables of The Debugger.

Then one defines the action clause:

```
?Mark;  RESUME;  { PC is the implicit arg to the Mark procedure }
```

Next one selects the **Stops:Proc Entry/Exit Trace** command with the action clause set to the number of the one we just entered. Control will return to the PP.

When we are finished executing, one can display the list of procedures that were entered by executing **?showMarks;** statement with **Cmd-shift-[**.

The **?isMarked()** function returns TRUE if the procedure at PC has the mark bit set.

Examples of Conditional Breakpoints

1)

```
IF (?bkpt > #50) & (RD0.L <> ?saveD0) then BEGIN
   wr_entry; write(' D0 clobbered'); PAUSE
END
ELSE Resume;
```

`"?saveD0"` is the name of a debugger variable where you have previously saved the lower 16 bits of D0 by the statement ".`?saveD0 := RD0.L;`". The write statement is executed if the code to which the Bkpt is attached has been executed more than 50 times and the low 16 bits of the Register D0 is not equal to the value of the Debugger Variable `"?saveD0"`.

2)

```
?cnt := ?cnt + 1;
Writeln(?cnt:integer, ' ticks = ',ticks:Longint);
if ?Bkpt < #20 THEN resume;
```

This example shows that any statements may precede the IF statement in a conditional Breakpoint. The value of the Debugger variable "`?cnt` " will be automatically initialized to zero. This Bkpt will print the value of `?cnt` , the string "`ticks =` ", the value of `ticks` as a Longint and resume execution of the PP for the first 20 times it is entered. It will then pause. One may attach this action clause to a location in a PP's main event loop.

Note the use of the # sign to force a decimal constant.

3)

```
Writeln(?Bkpt:integer,' delta = ',ticks-?oldticks:integer);
?oldticks := ticks;
if ?Bkpt < #20 THEN resume;
```

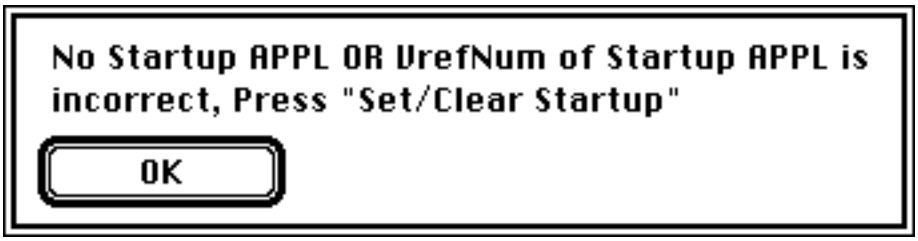
Will display the difference in `ticks` as we go through some loop. **KEEP in mind** that in single screen mode BOTH the screen and the System Global area are being swapped once per execution of the Bkpt. In multi screen mode, only the Global area is being swapped ($1400 - $1E00 bytes)

More examples of scripts are in the file 'Dbgr Scripts' in 'Dbgr/Nosy files'

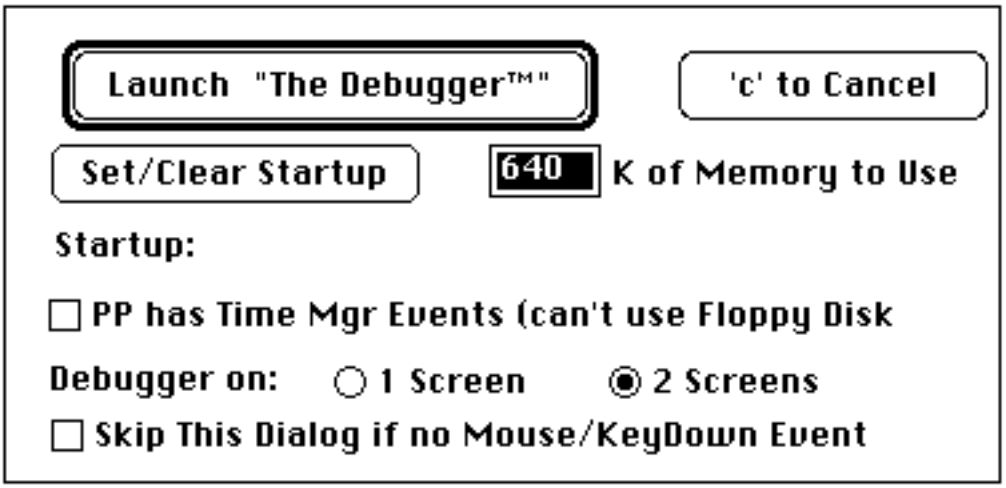
Startup

Starting *The Debugger* *(System 6 only - can you say: obsolete)*

1. Follow the steps outlined in section 6.6 on “Preparing Files for Debugging”.
2. Start up ***RunDbgr*** from the Finder or some other program launcher.
3. If ***RunDbgr*** can't find the PP, up pops a reminder to use the **“Set/Clear Startup”** button on the the startup dialog to select a Problem Program:



4. ***The Debugger***'s startup dialog appears:



5. Starting from the top, the items on the startup dialog are:

   The **Launch "The Debugger™"** button - Click this button (or press Return or Enter) to launch ***The Debugger*** when everything else in the dialog is correctly set.

   The **‘c’ to Cancel** button - Use to abort the startup and return to the shell. As the legend on this button suggests, you can cancel by pressing the letter ‘c’ on the keyboard as well as by clicking in this button.

   The Set/Clear Startup button - Clicking this button brings up a standard file selection dialog that you use for selecting the Problem Program:

   The **K of Memory to Use** editText item -The size of the heap ***The Debugger*** will live in.

   The **Startup:** staticText item - The name of the PP. It does not show the PP's full pathname.

   The Debugger* on:** radio buttons - Use to select single or multiscreen operation. On a Mac II, ***The Debugger will use the Startup Screen (selectable from the control panel; hold down the option key and select Monitors. Move the smiley face to the monitor you want to be the Startup screen.).
   On a Mac with only one screen, these selections will be ignored, and ***The Debugger*** will allocate a buffer to hold the image of its screen.

   The **Error Messages:** radio buttons - Ignore them..

   The **Skip This Dialog…** checkbox item - If you check this item, subsequent launches of RunDbgr will only bring up this startup dialog if you hold down a key or the mouse button.

6. Deal with any relevant startup dialog items, then click the **Launch "The Debugger™"** button or press Return or Enter.
7. If ***RunDbgr*** can't find ***The Debugger*** (first time only), it will post an alert:

   Acknowledge this alert. Then a standard file selection dialog will appear. Use it to select ***The Debugger***. That will show ***RunDbgr*** where ***The Debugger*** is.
8. ***The Debugger*** will start up, posting status information in its **–Notes–** window as it goes. If you've followed the previous steps, everything will work, and you will end up with ***The Debugger*** functional. Besides **–Notes–**, there may be a **–Code Blks–** window showing the blocks of the PP.

   If a file of the form "ApplName.dsi" is present in the same folder as the startup APPL, then ***The Debugger*** will open it and execute the commands in it.

   If there is a problem, reread this documentation up to this point, then carefully rerun Steps 1-7. If disaster persists, get in touch with us.
9. In most cases, the next thing you will do is launch the PP into the application heap. This is done by the **Go:Exit to Program** command (**Cmd–E**). Once the PP is launched successfully, press **Option–** to get back to ***The Debugger***. (**Option–** causes ***The Debugger*** to reassert control after the next call to GetNextEvent by the PP, it is a convenient means of breaking into the event loop of an application).
10. In some cases, you may want to use ***The Debugger*** to set some items before the initial PP launch : breakpoints (using commands from the **Stops** menu), trap intercepts , etc.
    Perform any such set up, then give the **Go:Exit to Program** command described in Step 9.
    You may use **.dsi** files to automate the process.

•••••••••••••••••••••••••••• END of Obsolete material ••••••••••••••••••••••••••••

Initial entry Into the Problem Program

In ***RunDbgr mode*** the PP is not launched into its heap until you give the **Go:Exit to Program** command. In most cases, you'll do this right after your first entry into ***The Debugger***. In some cases, you may want to set some controls (breakpoints, trap intercepts, etc.) by means of a **.dsi** file or by using the menus of ***The Debugger*** before you launch the Problem Program. In either case, **Cmd-E** is the way to go. And, remember, you can always (unless you change things with the **Stop:GNE Intercept Option** command) get back to ***The Debugger*** by pressing **option-** or the **interrupt key**.

Screen Toggle

The **Tilde or backquote key** (~) toggles between ***The Debugger*** screen and the user screen.

Returning to The Debugger

To transfer from the PP to ***The Debugger*** you may:

a) Press the interrupt key, which will bring you back into ***The Debugger*** at an unpredictable point .

b) Press the **option and backslash** keys, and your PP will enter ***The Debugger*** on exit from the next GetNextEvent trap called by the PP (or called from the ROM as a consequence of a WaitNextEvent trap executed by the PP).

Returning to the Problem Program

Press the key combination **Cmd-E** (Exit to Program in the GO Menu)

Shutting Down the PP in Background Mode

When ***The Debugger*** is running in **Background Mode** use the **Shutdown APPL** item of the **-D-** menu.

Various Command Interfaces

The Debugger gives you several ways to supply data to a command:

Standard Mac Interface

Make a selection, then give a command via the menus or keyboard. Many commands check for a selection in the front window and will use it if present.

Keyboard interface

For some commands, if there is no current selection when you give the command, or the current selection is inappropriate to the command, ***The Debugger*** brings up a **command-line dialog box** into which you can type or paste into. When this box appears, the syntax for the command is displayed in reverse contrast. Some of the commands that work this way are:

Tables:Fields of , **Stops:Set Intercept** , **Stops:Set Watchpt**.

You can **paste** into these dialogs with **Cmd-V** and dismiss them with **Cmd-Q**. Clicking in the "**Cmd**" box is equivalent to typing Return.

List Manager Interface

a) **Formal** - This is the kind of list manager interface you see in a typical Macintosh application. Used for file selection (**Open** and **Save As** commands) and the big **Stops:Trap Intercept dialog**.

b) **TextEdit windows** - Select one or more lines, then give a command.
For example: select lines in a structured code-disassembly window, then give one of the breakpoint toggle/set/clear commands. Or select variable names from a file or from the **-Global Vars-** window and then give the command, **Misc:add/DEL to Values**.

Stopping or Pausing

Cmd-period to abort a command OR to stop **Continuous-Step** mode.

Hold down the **Cmd key** to pause **Continuous-Step** mode

Window Types

The Debugger puts up several types of windows to display information about the PP and the system.

Structured disassembly window(s):

Select a procedure name in any window or list, then press **Cmd-D** or use the **Display:Code|Data Block** menu selection to bring up the display. Alternatively, select an item of the form **=addr**, or **$SystemGlobalAddr**.

NOTE: To reselect the current instruction, Select the value of the PC in the **-Registers-** window and **Cmd-D.**

Lines in a structured disassembly window follow this format ...

addr: **B mrs EA EA^** label opcode operands comment

where:

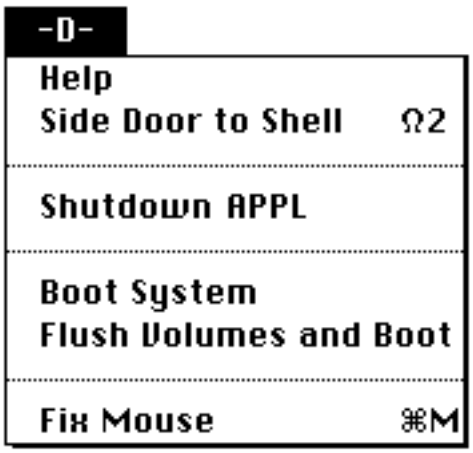
B is a Breakpoint indicator (the bullet, •).
mrs is the mode, register and "size" (0 to 2) of the instruction.
EA is the Effective Address of the Source or the Source/Destination operand of the instruction.
EA^ is the value of **EA** prior to executing the instruction.
Except for the 680x0 Scc instructions, **EA^** is the RESULT of executing the instruction.
"Label opcode operands comment" is the disassembled instruction.

Here is a quick form description of the various other windows ***The Debugger*** can produce:

Window Type	Command	Comment
Dump of the 68000 registers	**Misc:Registers**	Window updated automatically
Dump of the 6888x (FP) registers	**Misc:FP Regs Dump**	Window updated automatically
Stack dump	**Misc:Stack Dump**	
Stack Crawl	**Misc:Stack Crawl**	
Dump of the application heap	**Cmd-H**	See discussion **Misc:Heap Dump**
Dump of the system heap	**Shift-Cmd-H**	See discussion **Misc:Heap Dump**
Dump of any heap	make a selection of the form **hz@addr**, then press **Cmd-Space**	See discussion **Misc:Heap Dump** and **Tables:Fields of**
System symbols	**Cmd-Shift-]**	For each system symbol, shows the address, name, type, and value. See description **Tables:Sys Syms**
Value of a global VAR	select a global variable or System symbol name, then press **Cmd-W**	See **Misc:add to Values**
Hex dump of memory	select **@addr** or **$addr** &**Cmd-Space**, or a selection of anything containing **@addr**, then press **Cmd-shift-space**	See discussion of **Tables:Fields of**
Unstructured disassembly	make a selection then press **Cmd-=**	See **Tables:Code @ addr**
Record or data type definition	select a type name, then press **Cmd-space**	See discussion of **Tables:Fields of**
Record or other data type value	select **TypeName_@addr** & **Cmd-space**	See discussion of **Tables:Fields of**
Resource map	**Display:Rsrc Map** , or **select rm@addr** & **Cmd-space**	See discussion of **Tables:Fields of**
Open files, current Tasks	**Cmd-option-T**	**Misc:Task & File Info**
Values of selected System or PP globals	**Cmd-U**	
Values of local VARs of the active procs	**Cmd-shift-U**	
Contents of a Text file	**Cmd-O**	
Debugger Flag settings	**Cmd-option-Z**	

Menu Reference

The -D- Menu



The **-D- Menu** is a potpourri of commands that let you get online help, transfer program control, and resuscitate the Mouse. At one time, the menu bar contained "**-D-**", but now it displays the amount of unused space in ***The Debugge***r's heap zone.

The **Help** command is easy to use. Select it, and a window named **–Help–** opens up. You can take advantage of the **Edit** menu commands, as follows: Highlighting one of the topics at the top of the **–Help–** window. Press **Cmd-F** to **Find** the next instance of the selected topic.
Press **Cmd-T** to scroll the text so the start of a topic is at the **T**op of the **–Help–** window.

The **Side Door to Shell** command lets you suspend the running application and transfer to the MPW Shell or ObjectMaster™ . It is usually used in conjunction with IBS.

The **Shutdown APPL** command (MultiFinder only) terminates the Task that the PC belongs to (if that task is an APPL or an MPW Tool) by closing the files of the task and issuing an _ExitToShell trap call.

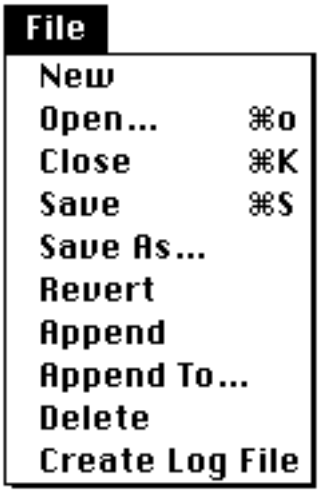
The **reLaunch APPL** command relaunches the PP into the application heap. Control is transferred to the PP. ***The Debugger*** continues to live in its own heap above BufPtr. **IMPORTANT**: do not use this command to do an initial launch of the PP. Use **Cmd-E** for that job.

The **Boot System** command reboots the Macintosh without flushing the file system data structures to disk. Use **Boot System** to reboot the Mac if you think the file system data structures have been corrupted.

The **Flush Volumes and Boot** flushes the file system data structures to disk, then reboots the Macintosh. This command gives you a speedier reboot than the Boot System command Use this command if you know your PP has **NOT** corrupted the file system data structures.

The **Fix Mouse** command (**Cmd–M**) cures frozen-mouse cursor problems by re–initializing the serial ports. **This command only works on the Mac Plus.** It is greyed out on ADB Macs.

The File Menu



The **File Menu** contains a standard set of Macintosh file menu commands, with the addition of **Revert**, **Append**, and **Delete** commands. This discussion focuses on the differences from the standard stuff.

The **New** command creates a new Macintosh text editing window. The window comes up with a name of the form ?-number. The first such window will be named ?-1, the second will be ?-2, and so on. If saved, the saved file's type is **TEXT** and the creator set to **'MPS '** (MPW Editor).

The **Open** command (**Cmd-O**) brings up a standard Open File dialog and lets you open any **TEXT** file. If you highlight a name in the front window, pressing **Shift-Cmd-O** tells ***The Debugger*** to try to open a file with that name on the current volume.

The **Close** command (**Cmd-K**) closes the front window. If the contents of the window have changed since it was opened, ***The Debugger*** gives the standard **Save changes before closing?** dialog.

The **Save** command works as you would expect, as does **Save As**.

The **Revert** command lets you return the file associated with a changed window back to the state of that file when it was last saved.

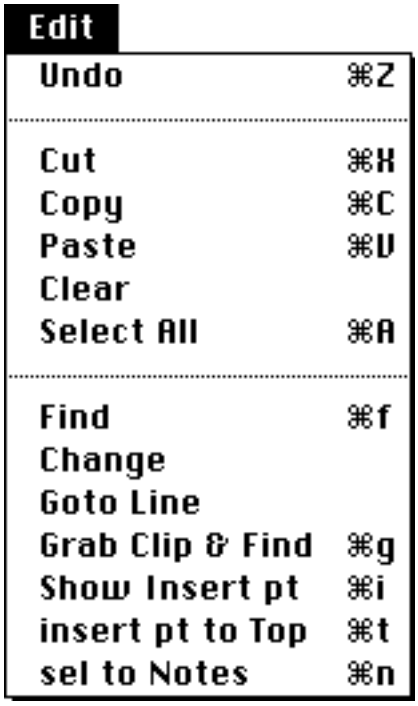
The **Append to PP_id.txt** command adds the current selection to the named file. The file name is formed by adding the suffix **.txt** to the name of the PP.

The **Append To** command works similarly, adding the current selection to a file chosen by you from a standard file selection dialog.

The **Delete** command closes the front window and deletes the associated file.
Not to fear, there is an **Are you sure?** dialog before the act is consummated.

The **Create Log File** creates a window with pertainant info about the state of the machine that can be sent to others in an attempt to help them analyze a 'crash'.
Note: the command is a misnomer as it does NOT save the window to disk,
you have to do that with a 'Save' command.

The Edit Menu



The **Edit Menu** contains the standard Macintosh Edit menu commands, plus additions unique to ***The Debugger***

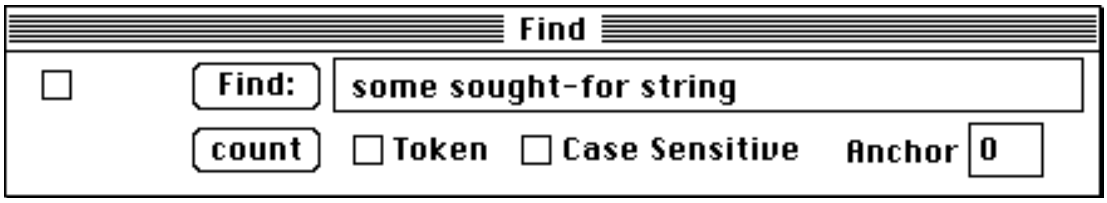
The **Undo** command is used to undo typing, text deletions, etc. Typing in text and then undoing it twice will bring it back and select it for possible execution in some command.

Cut, **Copy**, **Paste**, **Clear**, and **Select** all work in the standard ways.

The **Find** command lets you find arbitrary strings. You can choose **Find** from the menu or press **Cmd-F**. All searches wrap around the text of the window. A search can look for a token (whole word, bounded by whitespace or punctuation) or partial matches. A search can be case sensitive or insensitive. And a search can look for anchored matches, where the target string begins before a certain column position in a line.

If there is a current selection and the **Find** dialog window is not lurking on the desktop, ***The Debugger*** uses the current selection to go on a default search. The default search is token-seeking, case sensitive, and not anchored. Holding down the **Shift** key restricts the search to the label field of each line. The label field starts in column 32.

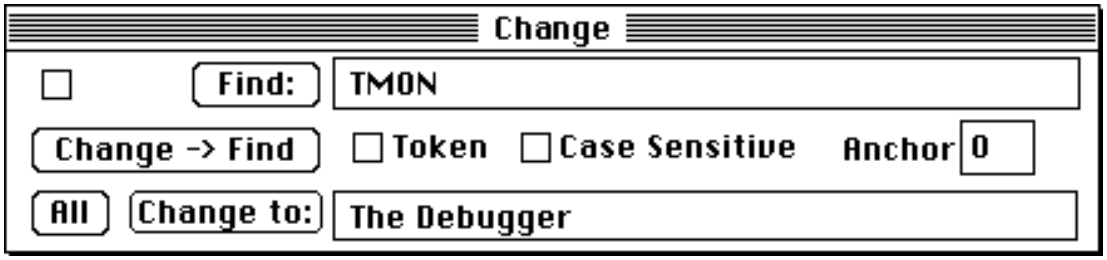
If there is no current selection, just that blinkin' insertion point, or the **Find** dialog window's somewhere on the desktop, the **Find** dialog window comes front. whence you can set the various search parameters:



Clicking the **Find** button, or pressing **Return**, searches for the next occurrence of the string shown in the 'Find:' textEdit item. The count button on the **Find** dialog counts the number of matches. The **Token** and **Case Sensitive** checkboxes let you set those search qualities. If the **Anchor** editText item is a positive integer **nn**, the search is restricted to the first **nn** columns. If **nn** is 0, the search is non–anchored.

Find supports the standard **Cut/Copy/Paste** commands. And the box to the left of the **Find** button is just a **close box**, albeit in a novel position.

The **Change** command lets you change strings. A dialog box appears, with options similar to the **Find** dialog and other Mac **Change** dialogs. Also like **Find**, **Change** supports the standard **Cut/Copy/Paste** commands and has a, well, a **close box**. Here's an action snapshot:



The **Goto Line** command lets you hop to any line in the front window. An obvious little dialog box appears -- fill it in and off you go :

The **Grab Clip & Find** command grabs the contents of the **Clipboard** and uses it as the search string for a **Find** in the current window. The keyboard equivalent is **Cmd-G**.
NOTE: Cmd-G does a case insensitive non-token search and shift-Cmd-G does a case-sensitive token search.

Example: looking at proc8, you want to see how another procedure references it. Select proc8, **Cmd-C** to copy it to the Clipboard, select the name of a procedure that references proc8 from the refs list at the beginning of proc8, **Cmd-D** to bring up a display of the caller, and **Cmd-G** to find the reference. Sounds complex, but this is a standard sequence in ***The Debugger***: fast and useful.

Holding the **Shift key down** when invoking **Grab Clip & Find** does a label–field search, as detailed above for the **Find** command.

The **Show Insert** pt command scrolls the window so the text insertion point is at the window's vertical center. Also invoked by **Cmd-I**.

The **Insert pt to Top** command scrolls the window so the text insertion point is at the top line of the window. That is, it acts as a **local Home** command. The keyboard equivalent is **Cmd-T**. Holding down the **Shift key** when invoking **Insert pt to Top** moves the insertion point to the beginning of the text prior to the scroll, and thus functions as a **global Home** command.

The **sel to Notes** command (**Cmd-N**) copies the current selection to the **-Notes-** window.
Nice way to click quick snapshots of your debugging activities.
Holding down the **shift key** closes the window from which you are copying.

The Display Menu

The **Display Menu** contains commands to display information about the task you are debugging. For most of the commands in this menu, selecting one that has an open window will bring that window to the front.

IMPORTANT: Many of the commands on the **Display Menu** depend on information about the PP that is developed in ***MacNosy*** and transmitted to ***The Debugger*** via an **.snt** file. In the absence of such a PP **.snt** file , you will get results that are inaccurate and/or incomplete.

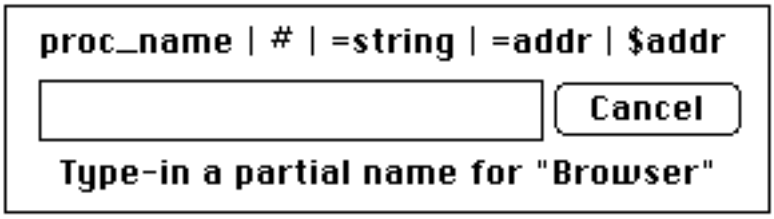
The **Code|Data blk** command displays a block. **Cmd-D** is the keyboard equivalent. In the following discussion, remember that **selecting** and **selected** mean that you have highlighted a string of characters in the front window.

If a valid block name is selected, invoking **Code|Data blk** tells ***The Debugger*** to bring up a display of that block. A valid block name may be the name of a procedure/function in the PP, or one of ***The Debugger*** standard forms, such as “proc123” or “com_45”, or any valid ROM routine name.

If a valid data block name is selected, holding down the **Shift key** while invoking **Code|Data blk** tells ***The Debugger*** to additionally bring up a display of any procedure block immediately preceding the selected data block.

If you select a **partial** procedure name (such as a C++ class name), then Cmd-D will open up a browser window with the names of all the procedures that begin with the entered character string.

If there is no selection, a dialog box appears so you can specify a block :



You have several options in using this dialog:

You can enter (or paste) the name of a **procedure**, **common**, or **data block**.
It is sufficient to type it in to the point of it being unique in the 'name-space' of the task.
For example, entering **com_5** tells ***The Debugger*** to bring up the block of that name.

You can enter an **address**, optionally prefixed with an equals sign (=).
That tells ***The Debugger*** to bring up the block containing code or data living at that address.

NOTE: If the address is not within the "current" task, ***The Debugger*** will switch to the task containing the address. You can see if this has occurred by watching the **Task Menu** title.

If you want to look at the Mac **ROM** routines that are jumped to indirectly by vectors in the System global area, you can enter the **address of the system global, prefixed with a dollar sign** ($). That tells ***The Debugger*** to display a procedure whose address is stored at the address you enter. For example, if you enter $720, and $720 contains the value $80D22E, ***The Debugger*** will display the **ROM** procedure that contains that address. Among other things, this can be used to decipher sequences of the form:

```
PUSH.L   sysGlob   followed by an RTS
```

The **Refs to** command (**Cmd–R**) displays a reference map for the selected name. A .**snt** file. is required.

The **Call chain** command displays the procedures that call a selected procedure, and the procedures that call them, etc., until all the references are tracked down. Think of it as a multi-level **Refs to** command for procedures. This command needs a .**snt** file.

The **Sys syms map** command displays a map of Macintosh **system globals** that are referenced by the PP, along with the parts of the PP that reference them. This command needs a PP .**snt** file, with the 'References' retained from ***MacNosy***

The **Trap refs map** command displays a map of Macintosh traps that are referenced by the PP, along with the parts of the PP that reference them. This command needs a .**snt** file., with the 'References' retained from ***MacNosy***

The **Globals map** command displays a map of the global symbols for the PP, the values of these globals, their types, and the parts of the PP that reference these globals. Everything but the references work without a PP .**snt** file.

The **Rsrc map** command displays a map of the resources of the PP.

The **Strings** command displays a map of all the data blocks that are formatted as ASCII strings (ABn, Str, ZSTR, etc), along with the parts of the PP that reference them. This command needs a PP .**snt** file.

The **Data Blks** command shows the data block names. Untyped blocks are marked with a **> prefix**. Blocks that are not referenced by any other part of the PP are marked with a **? prefix**. A .**snt** file. is required.

The **Case jumps** command displays information about the case statements. A .**snt** file. is required.

The **Mystery procs** command displays information about code constructions ***MacNosy*** was unable to understand. A .**snt** file. is required.

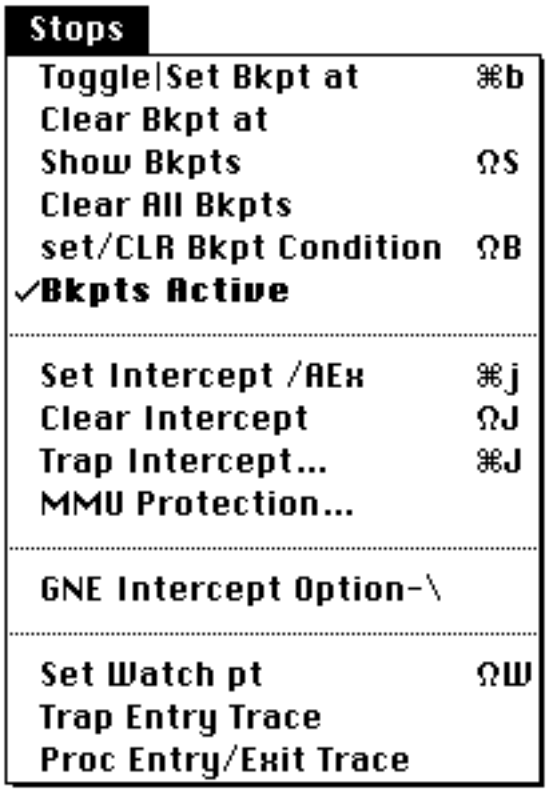
The **ROM Patch Info** command displays the names of the files that have patched the ROM, and a list of the names of the patched traps and the patches to them.

The **Bad Blks** command displays information about blocks that ***MacNosy*** thought were code, but contained illegal instructions. ***MacNosy*** changes such blocks to data. This command needs a PP .**snt** file.

The **Blk tbl** command displays the block table. The block table contains the following information for each and every block in the PP: name, segment number, address of the first byte of the block, length of the block. This command works without a PP .**snt** file.

The **Code Blks** command shows the code block names. Procedures that do not call other procedures (i.e. leaves of the call graph) that have not been given a name beyond the generic **proc***number* are flagged with a >. This command works without a PP .**snt** file.

The Stops Menu



The **Stops Menu** contains commands that let you set conditions under which the PP will stop and transfer control to ***The Debugger***.

The **Toggle|Set Bkpt at** command lets you toggle and set breakpoints at locations in the PP. You can invoke the command from the menu or by pressing **Cmd–B**. Breakpoints (in case you forgot) are marked by a **bullet**: •, as seen in the figure below. The number of breakpoints you can set is limited only by the amount of memory available to ***The Debugger***.

The flavors of this command, depending on the state of the current selection are:

(1) If the current selection is a line in a structured assembly or source window, invoking this command **TOGGLES** a breakpoint at that location. For example, pressing **Cmd–B** when the current selection in an assembly window is this one line:

```
8:•206D FFF0          -$10          MOVEA.L HTE(A5),A
```

will clear the breakpoint at location 8. If **Cmd–B** is applied again to the line, the breakpoint will be set.

(2) If the current selection includes one or more procedure names, or one or more procedure name plus offset expressions, invoking this command **SETS** breakpoints at the indicated location(s).

For example, pressing **Cmd–B** when the current selection in a qualifying window includes these three lines:

```
SCROLLTEXT+20¶
TOGGLESCRAP¶
SETVIEWRECT+4¶
```

will set breakpoints at the three indicated locations.

(3) If there's no current selection, invoking this command brings up the command line dialog box. You can then specify a breakpoint location by entering a procedure name, or a procedure name plus an offset. ***The Debugger*** will then SET a breakpoint at that location.

You may attach an "**action clause**" to the Breakpoint by holding down **the Shift** key when you set a breakpoint. You attach an **action clause** to a breakpoint by using the **set/CLR Bkpt Condition** to define an action clause and to set its "**condNum**". This **condNum** is picked up by future Set **Bkpt** commands.

An **action clause** may be any sequence of statements defined in the expression language; refer to page 38.

Every action clause is implicitly terminated by a **PAUSE** statement.

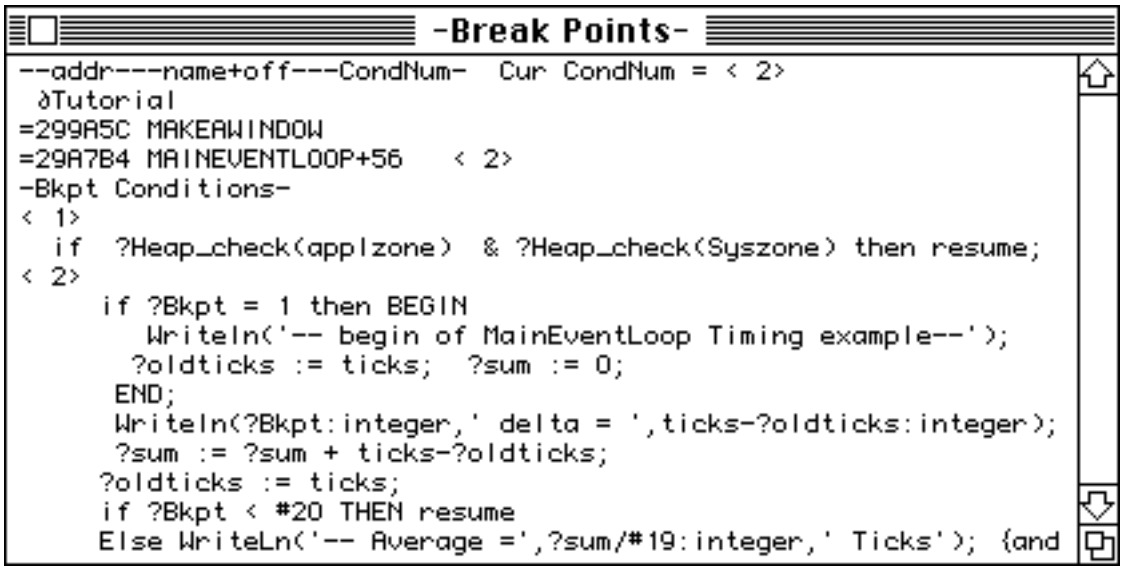
A simple action clause might be of the form:

```
IF rel_expr THEN Write and Writeln stmts; optionally followed by a Resume
```

The sequence of events at a conditional breakpoint is to evaluate `rel_expr`. If its value is not zero the "statements" after the THEN are executed and the results are placed in the **-Notes-** window. If there is a **Resume** statement, the execution of the PP continues, otherwise one pauses in ***The Debugger***.

The **Clr Bkpt at** command lets you clear breakpoints. You can specify breakpoint locations by selecting addresses in a structured assembly window, selecting procedure names or procedure name plus offset expressions in a not structured assembly window, or by entering a procedure name or procedure name plus an offset in the command line dialog box.

The **Show Bkpts** command (Ω-**S**) opens the **-Break Points-** window that shows the addresses of currently–set breakpoints in the form: program address and procedure name plus offset expressions, followed by the number of the attached action clause, if any. The **-Break Points-** window also contains a list of the "action clauses" and their **condNums**. Here is an example of such a window: Note that the breakpoint at `MainEventLoop+56` has action clause number 2 attached to it.



The **Clr All Bkpts** command does just what it says: clears all breakpoints.

The **set/CLR Bkpt Condition** command (Ω-**B**) lets you define **conditional statements** (action clauses) that may be "attached" to Breakpoints.

The conditional statements are referenced in the **Set Bkpt** command by their **condNum**. The **condNum** is the bracketed integer **< 2>** on the line above the action clause in the above window.

The command acts as follows: When the **Shift key** is held down you are presented with a "type-in" dialog box. Enter the number of an action clause to be **deleted**. When the **shift key** is **NOT** held down, you may **add** an action clause by highlighting it in the front window prior to selecting the command. If you do not make a selection, you may **set** the **condNum** for use in future Set Bkpt commands.

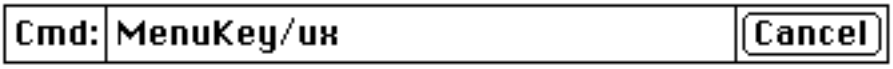
The **Bkpts Active** command lets you **activate**/**deactivate** the current set of breakpoints without removing them. A nice convenience.

The **Set Intercept/AEx** command (**Cmd-J**) lets you set trap intercepts.
If there is a selection that is a valid trap name, that trap that will be intercepted. Note that in a structured ASM window you may select an entire line. If there is no selection, the command-line dialog box appears. You can then type in or paste a trap name, or a range of trapnames (trap-1-trap_2/...)
OR $Annn (trap number).
The trap name can be followed by a switch indicator, /, and one or more of these switches:

A - All: intercept calls from the PP and the ROM
U - User: intercept calls from the PP only
E - Entry intercept on entry to the trap
X - eXit intercept on exit from the trap

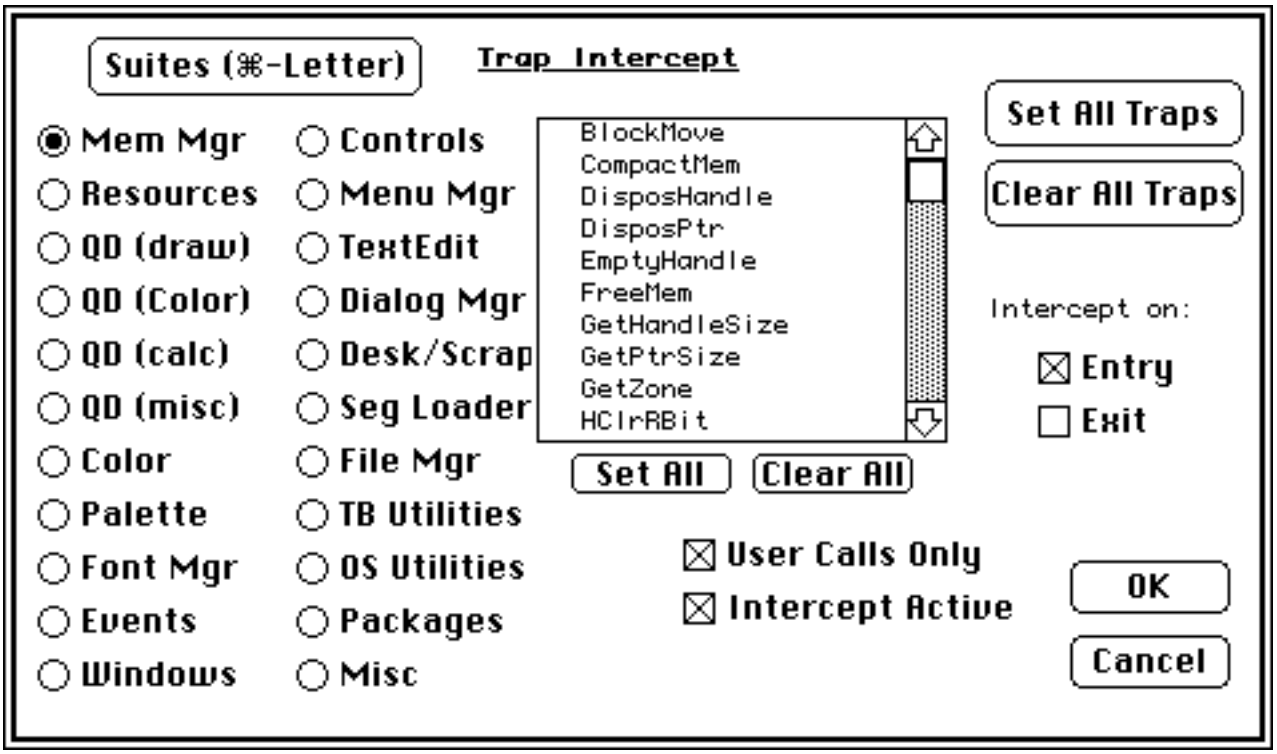
If no switches are specified, the intercept takes on the default switch settings. The current defaults are shown next to **Set Intercept/AEx** in the menu, with a capital letter indicating a switch that is set. The defaults change to the last overt switch setting.

To trap MenuKey calls from the User on Exit, one would enter:



The **Clear Intercept** command (Ω-**J**) lets you clear a trap intercept. If there is a current selection that is a valid trap name, that trap that will no longer be intercepted. If there is no selection, the command line dialog box appears. You can then type in or paste a trap name, or a range of trapnames (trap-1-trap_2).

The **Trap Intercept** command (**Cmd-shift-J**) brings up a dialog box that lets you easily set and clear large numbers of trap intercepts. Here is a picture:



IMPORTANT: **Set the three checkbox items before selecting any trap names.**

If **User Calls Only** is checked, ***The Debugger*** will intercept calls only from the PP. If **User Calls Only** is not checked, ***The Debugger*** will intercept calls from the PP and the **ROM**.

If **Entry** is checked, ***The Debugger*** will intercept **on entry to the trap**.

If **Exit** is checked, ***The Debugger*** will intercept **on exit from the trap**.

The radio buttons on the left let you choose a group, or suite, of traps to work with. You can also select a suite by pressing Cmd along with the first letter of a suite. The traps in the suite show up in the scroll box.

Click on traps in the scroll box to toggle their intercept state. Press **Shift and drag** the mouse for multiple selections. Discontinuous selections are allowed without the need to press a modifier key. **Bullets (•)** mark traps that have intercepts set.

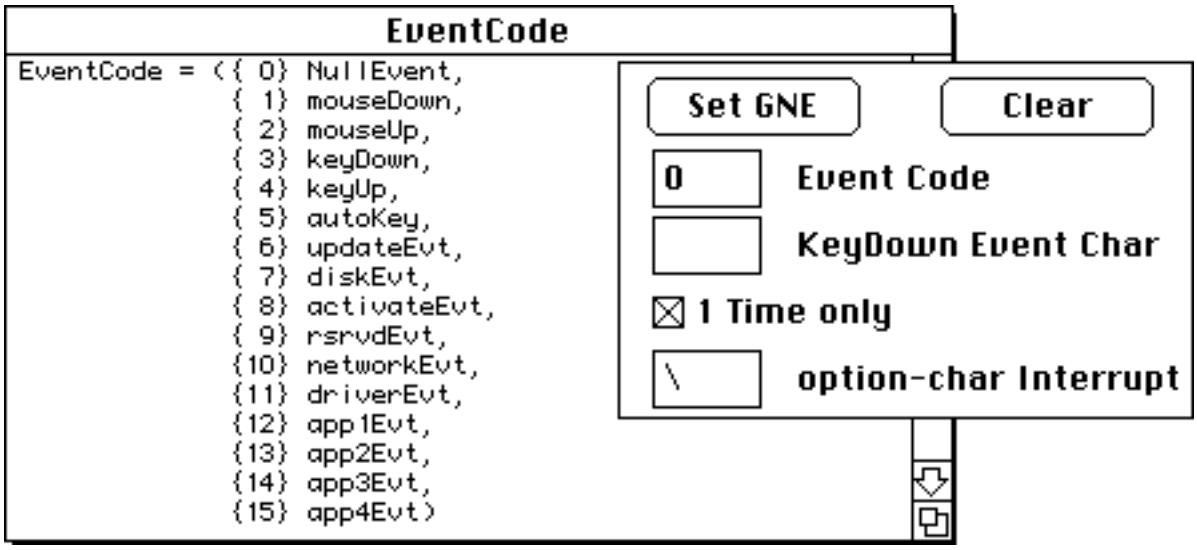
The **Set All** button sets intercepts for all traps in a suite. The **Clear All** button clears intercepts for all traps in a suite. The **Set All Traps** and **Clear All Traps** buttons do just what their titles imply:

NOTE: The GetNextEvent trap is a special case. If you want to set an intercept for the GetNextEvent trap via this **Trap Intercept** command or via **Set Intercept /AEx**, you need to first click the **Clear** button in the dialog box of the **GNE Intercept Option** command. Otherwise, your attempts to set or clear intercepts for GetNextEvent will fail.

The **MMU Protection …** command (68030 Macs only) is used to interactively change the MMU protection setting for a PP. See Debugger Tech Note #7 for details.

The **Soft MMU** command (PowerPC Macs only) is used to turn 'Soft MMU' on/off for a PP. See Debugger Tech Note #13 for details.

The **GNE Intercept Option -** command gives you two intercept possibilities for the vital GetNextEvent trap: the GNE intercept and the Option–char intercept. Intercepts set up via this command occur on exit from GetNextEvent. Invoking the command brings up a dialog box and a reference window showing the EventCode constants:



The **Set GNE** button sets GetNextEvent up for interception according to the settings of the other items of this dialog box. The **Clear** button clears intercepts based on this command. As noted above, you must click this **Clear** button if you want to use one of the other **Stop Menu** commands to set up an intercept of GetNextEvent. You would want to do this to intercept on **Entry** to GetNextEvent. You should also click **Clear** before doing any tracing of ROM code.

The next three items let you set up the GNE intercept. The **Event Code** editText item lets you select an event type to intercept. Just enter the appropriate number from the EventCode reference window.
Control transfers to ***The Debugger*** if GetNextEvent exits with a matching EventCode. If this item is set to 0, NullEvent, this first intercept is disabled.

If you set the **Event Code** item to 3, **KeyDown** events, and specify a character in the **KeyDown Event Char** editText item, ***The Debugger*** gains control on a **KeyDown** event involving that character. If you set the Event Code item to 3 and don't specify a character in the **KeyDown Event Char** editText item, ***The Debugger*** gains control on ANY **KeyDown** event.

Check the **1 Time only** checkbox if you want to set up a one–shot GNE intercept. Clear it if you want a continuous intercept, one that stays in effect until you select this command again.
The **option–char Interrupt** editText item lets you set up the **Option–char** intercept. You can specify a character that will transfer control to ***The Debugger*** when pressed along with the **Option** key. On such an event, ***The Debugger*** will change the GetNextEvent return code to FALSE so your program does not try to process the event. This intercept is active when ***The Debugger*** starts up, and remains active until you click the **Clear** button on this **EventCode** dialog. You can also change the default value of the character by modifying The Debugger's "Dcfg" resource in ResEdit.

The screen snapshot above shows the initial settings of these two intercepts: the GNE intercept is disabled via an **Event Code** item set to 0, the **Option–char** intercept is enabled with \ as the Option character.

A typical debugging sequence is to set these intercepts, exit to the PP, invoke the appropriate intercept event, then (upon entering ***The Debugger***) continue by setting particular breakpoints and/or stepping through some instructions.

The **Set Watch pt** command (Ω-**W**) lets you define an area of memory up to 16 bytes long that will be watched for changes, a.k.a. corruptive events. Invoking the command brings up the command line dialog box. You need to enter an even **hexadecimal FBA** (the address of the first byte in the watchpoint area), a **comma**, and the **decimal length** of the area to be watched. That will set up a **one–shot watchpoint**, enabled for a single corruptive event. If you want a **continuous watchpoint**, enabled for multiple corruptive events, append a comma followed with a 1 to the watchpoint specification.

After you set a watchpoint, control transfers to the PP. Control transfers back to ***The Debugger*** when a corruptive event occurs, i.e. if there is a change in the watchpoint area. **Cmd–E**, as usual, will then take you back to the PP.

A **one–shot watchpoint** is cleared when: a corruptive event occurs, you modify the watchpoint, or you bring up the **Set Watch pt** command line dialog box and click **Cancel**. A **continuous watchpoint** is cleared when: you modify the watchpoint or bring up the **Set Watch pt** command line dialog box and click **Cancel** or **Cmd-Q**.

The syntax for setting watchpoints allows for indirection and/or **offsets** in the **FBA** specification. Here is the complete syntax:

address {^} {+|- *offset*} , *length* { , {1} }

…where *address* is hexadecimal and even, *offset* is hexadecimal and even, and *length* is decimal and rounded off by ***The Debugger*** to the nearest multiple of 1, 2, or 4.

Here's a little table of watchpoint–setting examples (assume that memory location $1234 contains the value $5678):

Specification	**Watchpoint Area**	**Watchpoint Type**
1234,8,1	$1234-$123B	continuous
1234+132,10	$1366-$136F	one–shot
1234-12,15	$1222-$1231	one–shot
1234^,1,	$5678-$5678	continuous
123,1	$123-$124	one–shot
1234^-68,12	$5610-$561B	one–shot

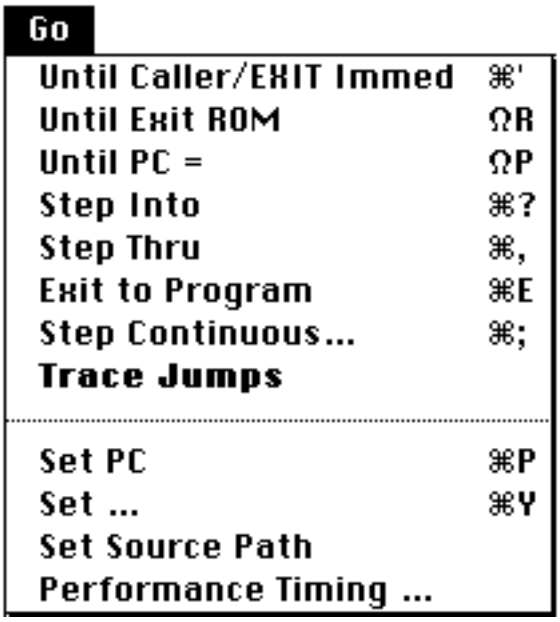
Some final watchpoint thoughts:

- If you're using indirectly-specified watchpoints that use a master pointer, you may find it useful to bring up a display of the associated heap.
- Tracing via the **Go** menu's various **Step** commands takes precedence over watchpoints.
- Watchpoints take precedence over tracing via the **Go:Trace Jump** command.
- If you want to go from the PP to The Debugger to cancel the Watchpoint, etc, then use the Programmer's Switch on the side of the machine.

The **Trap Entry Trace** command lets you specify an **action clause** that is executed prior to every Macintosh Trap call. See the help file **[-D-: Help]** for examples.

The **Proc Entry/Exit Trace** command lets you specify an **action clause** that is executed whenever a JSR, RTS, JMP (Ai), etc. instruction is executed. See the help file **[-D-: Help]** for examples.

The Go Menu



The **Go Menu** contains commands that let you transfer control to the PP and set the conditions under which it will run and transfer control back to ***The Debugger***.

The **Until Caller** command (**Cmd–'**) tells the PP to execute instructions until the procedure that called the current procedure resumes control. I.e. the PP will continue execution until the instruction following the most recently executed JSR/BSR is reached. Control then returns to ***The Debugger***.

If the **Shift** key is held down when this command is invoked, the current procedure is immediately exited. A7 and the PC are restored to their values prior to the JSR/BSR, and, if the called procedure began with a LINK A6, then A6 is reset.

Note: this command fails if the return address of the calling routine has been removed from the stack.

The **Until Exit ROM** command (**Cmd-R**) tells the PP to execute instructions until the instruction following the trap call that entered the ROM is reached. If the **Shift** key is held down when this command's invoked, ROM is immediately exited. A7 and the PC are restored to their values prior to the trap call.

Note: this are some circumstances in which this command can fail.

The **Until PC =** command (**Ω-P**) provides **ROM** breakpoints. Select a line in a **ROM** structured-assembly window at which you want to stop. Then invoke this command. ***The Debugger*** will transfer control to the PP and the PP will then execute instructions in the 680x0's **Trace mode** until the PC matches the selected address. You may also use this command to Go until a RAM address is reached.

This command can **not** be used in conjunction with **Stops:Set Watch pt** or **Go:Trace Jumps**.

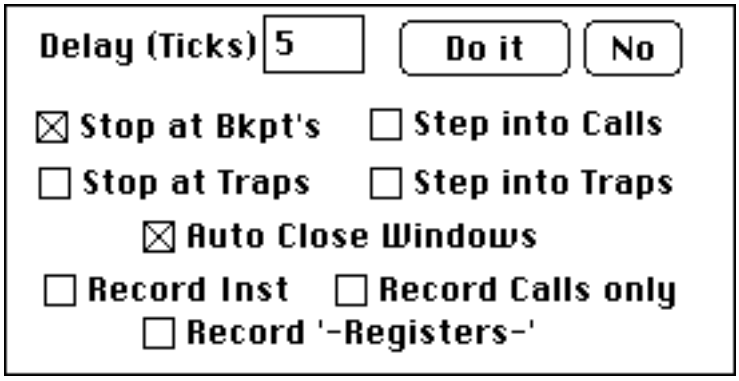
Note: If you keep the **Cmd** key down while invoking the next four commands, ***The Debugger*** waits until you release it before transferring control to the PP:

The **Step Into** command (**Cmd–?**) transfers control to the PP at PC and causes the PP to execute **one** 680x0 instruction. Control then returns to ***The Debugger***.

The Step Thru command (**Cmd–comma**.) is similar to **Step Into**. Invoking it transfers control to the PP. Then, if the instruction at PC is a JSR, BSR, or trap call, the subroutine or trap is executed as one instruction. Otherwise, the instruction at PC is executed and control returns to ***The Debugger***.

The **Exit to Program** command (**Cmd–E**) transfers control to the PP at PC. The PP is launched into its application heap the first time you give this command.

The **Step Continuous** command provides a continuous, or animated, step mode.
It can be invoked by pressing **Cmd–;**. Holding down the shift key bypasses the following dialog box:



The dialog items let you customize the continuous stepping. Most of them should make sense without further explanation. However…

At any point in time, there will be a structured assembly or source window that contains the instruction at the PC. If the **Auto Close Windows** item is checked, this window is shut when the PC moves to an instruction not contained in that window. otherwise the window remains open.

If you check any of the **Record…** checkbox items, information specified by that item gets recorded in the **–Notes–** window. As continuous stepping goes on, a number in the title of the **–Notes–** window tells you how much information has been recorded there. When the number gets to **60K** or so, we suggest you pause, save the contents of **–Notes–** to disk, and reset the window by pressing **Cmd–A** and **Backspace**. Also, recording registers or calls invokes the recording of instructions.

Continuous stepping begins when you click the **Do It** button. You can stop the process by pressing **Cmd–period**. If you press the **Shift** key when invoking this command, the dialog box is bypassed and continuous stepping begins, using the previous settings.

The **Trace Jumps** command enables jump tracing on a Macintosh with a 68020, 68030 or PowerPC CPU. Jump tracing tracks any **change in the flow** of program control (caused by instructions such as BRA, JMP, JSR). **Trace Jumps** can be used to track down wild leaps into the twilight zone. When control returns to ***The Debugger***, a **–Jumps–** window displays the last 20 changes of control-flow executed by the PP

NOTE: Jump tracing is **not** compatible with single–stepping, Memory Watch or Soft MMU.

The **Set PC** command (**Cmd-P**) lets you set the processor's PC register to a new address. There are three ways to specify the new address:

(1) If the front window is a structured assembly or source window, and the current selection includes an instruction's address, the PC will be set to that location in the code.

(2) If the current window is not a structured assembly window, and the current selection is of the form *procedureName*{+*offset*}, the PC will be set to that location in the code.

(3) If there's no current selection, the command line dialog box appears. You can then specify a location in the form *procedureName*{+*offset*}.

The **Set** command (**Cmd-Y**) lets you set the value of a 680x0 register or RAM memory location. A command line dialog box appears, bearing a command syntax prompt.

If you want to set a register's value, the command syntax is: R*xi=hexVal* where:

x is an A or a D to indicate an address or data register,
i is a value in the range 0..7 to indicate the register number,
and *hexVal* is a hexadecimal value for the **WHOLE** register.

For example, the command RA0=35 will set the A0 register of the 680x0 to $0035. Take special care if you dare modify the critical A5, A6, and A7 registers.

If you want to set the value of a memory location, the command syntax is

addr=hexVal{/ b|w|l}

where: *addr* is the hexadecimal address of the memory location
hexVal is the hexadecimal value to which you want to set this location

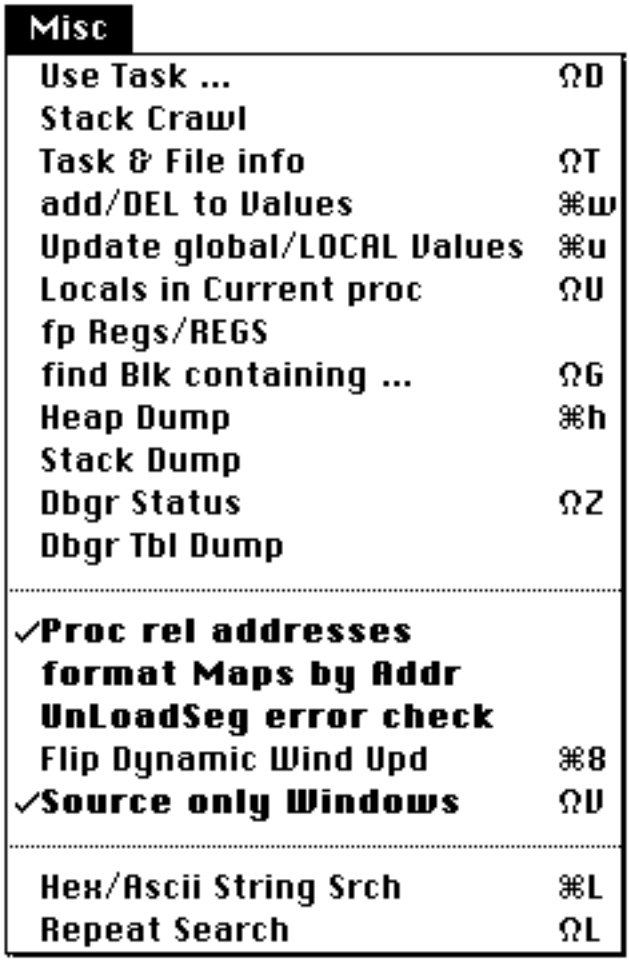
You can add an optional size specifier: **b** for byte, **w** for word, **l** for longword. The command will modify the byte, word, or longword beginning at *addr*. The default size specification is Longword.

Windows that show memory dumps or registers are updated immediately to reflect any changes.

The **Set Source Path** command lets you specify the path to the source files associated with a ".SYM" file.
The command will put up a Standard File Dialog Box. Naviagte your way to the source files and press return or click on the open button. You will then get another dialog asking if you want to specify more source paths.
If you have to set multiple source paths, then you should use the =S .dsi file command.

The **Performance Timing** command lets you monitor the execution of a program so that you can find out where it is spending its time. See Debugger Tech Note #10 for details.

The Misc Menu



The **Misc Menu** is a catch-all for the miscellaneous commands that don't fit in any other category

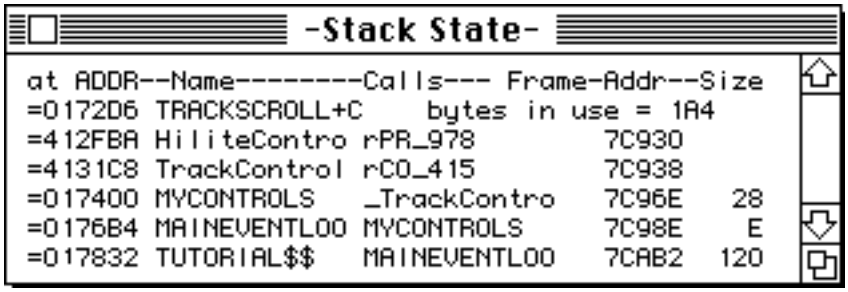
The **Use Task**… command tells ***The Debugger*** to switch to and use the specified task as the information source for a number of commands. These include many of the commands in the **Display Menu**.

An important internal Debugger data structure is the document that it uses as the basis for debugging an APPL or resource. A document for a task is created when that task is loaded into memory by the System and is deleted when the task terminates.

Names of procedures, global variables, etc. are task-relative objects. One may switch tasks by activating a task-relative window. The name of a task-relative window has a prefix of the form, **∂taskName.** Examples of task-relative windows are heap windows or the **.dsi** file window of a task.

The arguments to the **Use Task** command are: the name of the task (as found in the Task & File Info window), the number of the task, or an expression of the form "**=nnn**", where **nnn** is an address that is "in" the task.

The **Stack Crawl** command brings up a **-Stack State-** window that shows you something about currently–active procedures, as revealed by the state of the 680x0's stack. Here is a sample:



The first line of this **-Stack State-** window provides some labels. The second line gives the value of PC; that address expressed as a proc name plus an offset; and the size of the stack frame. Lines three and beyond follow a format indicated by the labels on the top line:

Column	Title	Content
1	at ADDR	address of the call to a procedure.
2	Name	name of the calling procedure.
3	Calls	name of the called procedure.
4	Frame–Addr	location of the called procedure's stack frame.
5	Size	Size of stack frame associated with the called procedure

More specifically, Frame–Addr is the location just above (stackwise) the return address pushed on the stack by the JSR/BSR of the calling procedure. In procs that use LINK/UNLK, this location will hold the saved value of the link register (A6).

Line three of the window holds the front (stackwise) procedure call. Line four holds the procedure call that is next to the top of the stack.

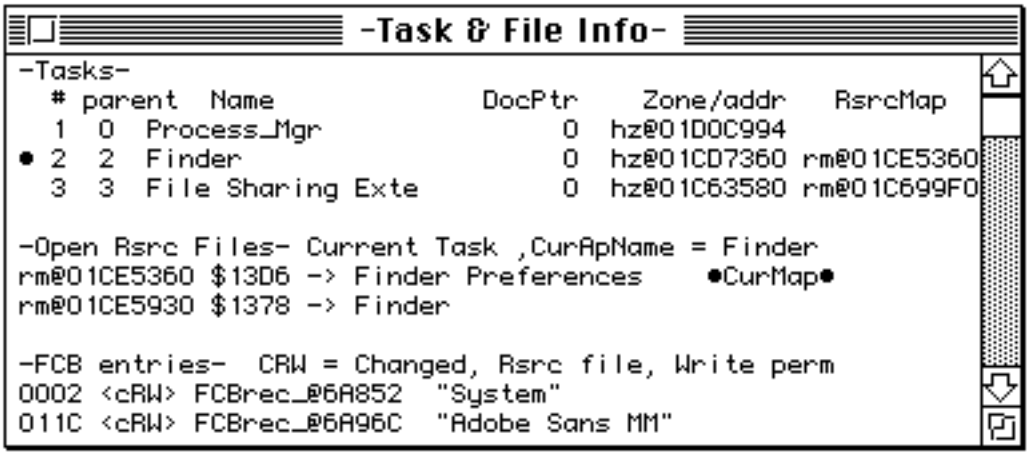
In the above **-Stack State-** window, you can read the call sequence from bottom to top as:

	TUTORIAL$$	called	MAINEVENTLOO
then	MAINEVENTLOO	called	MYCONTROLS
then	MYCONTROLS	called	_TrackContro
etc.			

and currently we are stopped at TRACKSCROLL+C.

The values in the **-Stack State-** window are NOT derived by chasing A6, but by pruning a stack dump.

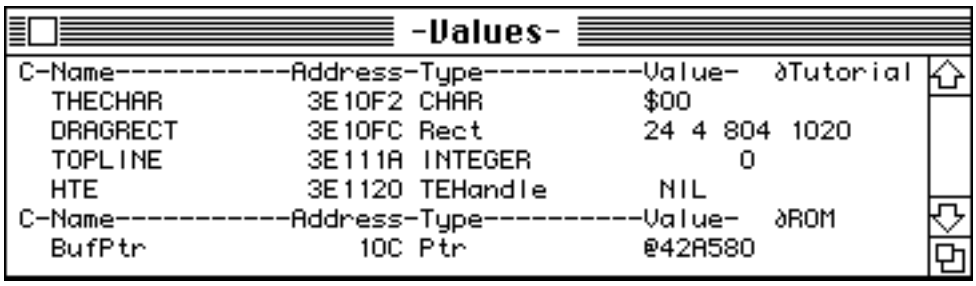
The **Task & File** info command brings up a window that tracks tasks and resource files.



A **–Task & File Info–** window contains three sections, a complete list of tasks in the machine, resource maps from the current task (TopMapHndl list), and file control blocks from the total universe.

The Open Rsrc files section displays the address of the resource map (prefix "rm"), its FCB refNum, and its name. The FCB entries section displays the FCB refNum of the file, the CRW (Changed, Resource_file and Write permission) bits followed by the FCB entry address and name. The CRW bits are are in uppercase if they are set (= 1).

The **add/DEL to Values** command (**Cmd-W**) lets you add and subtract variables from the **-Values-** window that displays current values about selected system and PP global variables.



Each entry in this table provides five fields of information pertaining to a particular global variable:

- First is a change field, labeled **C**. If the first four bytes of the data-object of the variable have changed since the last time the **–Values–** window was updated, a **bullet** (•) shows up in the change field.
- Next comes the **Name** field, holding the name of the variable.
- The third field is **Address**, with the absolute hexadecimal address of the variable.
- The fourth field is **Type**, holding an indication of the type of the variable. We will say more about types in a few paragraphs.
- The fifth field gives the **Value** of the variable. Values are in **hex**. The values of pointer variables are marked by an **@ prefix**; other variables get a **$ prefix**. Actually, only the first four bytes of the value of a variable are reported.

To add variables to the **–Values–** window, select one or more variable lines from another window, then invoke this command from the menu or by pressing **Cmd–W**.
NOTE: Cmd-W of a local var name will bring up the “Local Vars-” window & highlight the name.

The syntax of a variable line is:

{*leadingGarbage*} <*GlobName* | *GlobName:Type*> {;} { *trailingGarbage* }

GlobName is the name of a global variable. *Type* is one of the Macintosh type names known to ***The Debugger***. Invoke the **Tables:Record Names** command to see (and to be able to copy from) a full list of these types. *trailingGarbage* can be almost anything.

leadingGarbage has this syntax:

{*blank**} {-*} {-*offset*} {*blank**} | • *blank*

blank is a space character (ASCII $20). *offset* is the offset of a global variable of the PP relative to the address pointed to by some 680x0 address register (usually A5).

This flexible syntax allows for a very useful technique: prepare a text file that contains lines listing the variables of the PP, in the format *GlobName:TypeP*. For example:

```
MyWindow:WindowPtr;
hTE :TeHandle;
DragRect:Rect;
```

Then you can open this prepared file from ***The Debugger***, select whatever variables you want to look at, and add them to the **-Values-** window with this command.

You can copy variable lines from any window. Two of the most convenient are the windows brought up by the **Display:Globals Map** and the **Tables:Sys Syms** commands. These provide lists of the global variables of the PP and the Mac system, respectively.

The **–Values–** window can be updated manually with the **Misc:Update Values** command, or it may be marked with **Cmd-8** to update automatically.

To remove variables from the **–Values–** window, select one or more of its variable lines, then hold down **Shift** while invoking this command.

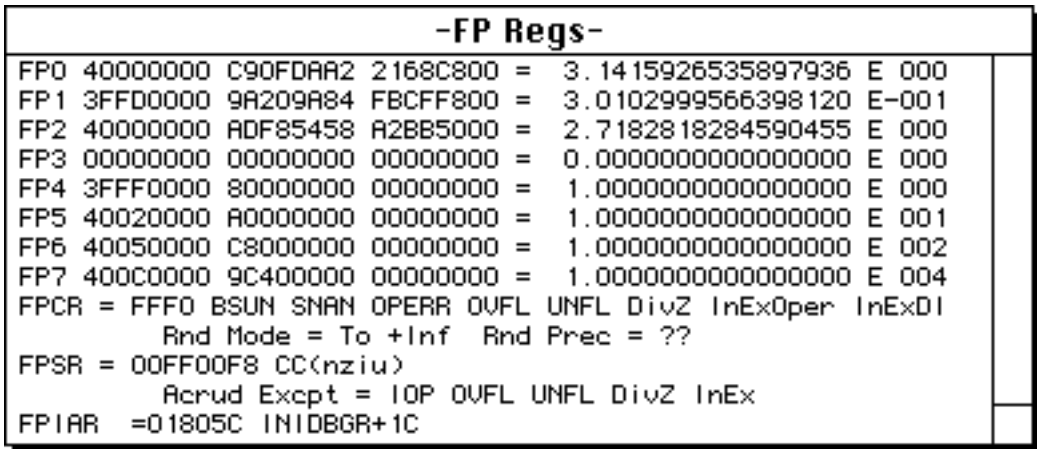
The Debugger knows the types of system globals, and of global variables specified in a ".SYM" file.

The **Update global/LOCALValues** command updates the display of variables in the **–Values–** window. Variables whose values (first 4 bytes only) have changed since the last update are flagged with a bullet (•) in the first field of the entry. . This command can also be invoked by pressing **Cmd–U**.

Holding the **shift key down** displays the values of the local variables of the active procs (up to 200 lines).

The **Locals in Current proc (Ω-U)** command displays the values of the local variables of the current procedure. The "**-Local Vars-**" window may be set for dynamic update with **Cmd-8**.

The **fp Regs / REGS** command, executed **without** the **shift** key held down, brings up a window that shows the contents of floating point registers in this format:



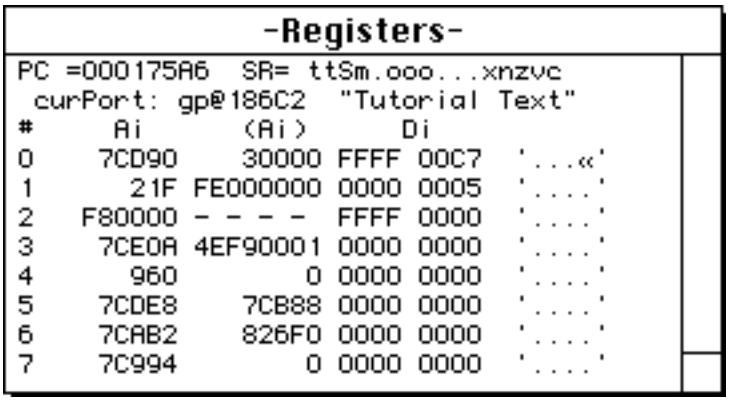
The first eight lines of this window show the contents of the eight floating point registers of the 6888x. Each line contains a register ID, then the value in hex form, then "=", then the mantissa value in decimal form followed by with exponent.

Next come four lines representing the Control and Status registers of the 6888x. For each register, the value held by the register is to the right of the "=". Then there are abbreviations interpreting the contents of each register. Check a Motorola 68881 manual for their meaning.

The final line is for the Address Register of the 6888x. the value held by the Address Register is to the right of the "=" followed by that address is interpreted as a procedure name plus an offset.

For PowerPC Macs, the FP Regs display shows the values of the PowerPC Floating point regs.

The **FP Regs / Registers** command, executed **with** the **shift** key held down, brings up a window that shows the contents of the registers of the 680x0 CPU.



The first line shows the value of the Program Counter and the Status Register. Status Register flags that are set show up as uppercase letters. The second line gives the location and name of the current **grafPort**.

The registers are displayed on lines four through ten. The third line provides five field labels for the registers display:

Field	Content
1.	register number
2.	contents, in hex, of the address register
3.	contents, in hex, of the memory location pointed to by that address register
4.	contents, in hex, of the data register
5.	value of the data register expressed as four ASCII characters. (Values outside the standard ASCII ranges are represented by periods).

Remember: as with most windows in ***The Debugger***, you can select interesting things in the **–Registers–** window, give interesting commands, and get interesting results. For example: you can double–click on the grafPort address in line 2, then press **Cmd–space**, and get a nice structured display of the data of the current grafPort.

The **find Blk containing…** (Ω-**G**) or where command takes an address (which may contain a leading $, typename, etc) and searches the heap zones for the block that contains the specified address.
If it finds a "match", it brings up a display of the heap zone and highlights the line containing the address. The uses of this command should be obvious.

The **Heap Dump** command lets you view heap zones. Pressing **Cmd–H** brings up a window dump of the **application** (PP) **heap** of the current task. **shift-Cmd-H** brings up a dump of the **system heap**.

To view other heaps, select text of the form `hz@`*startAddress*, where *startAddress* is the start of the heap zone, and press **Cmd–Space**..

The first line of a heap dump gives the starting address of that particular heap, in the form `Zone@nnn` . Then come a series of one–line entries for each block in the heap. Here are some sample entries taken from a system heap; refer to them as you read the field descriptions below:

```
•P mp@00143C     80 Masters
•P   @0014C4    1F4
•P   @0016C0     12 TimeVars
•H dc@001AEA     28 <L..> @0014B4 SoundDCE
•H   @001B1A     28 <L..> @0014B0
 H rg@001B4A      A <...> @0014AC Update Rgn of WindowRecord@15462
•P   @001BDA     20 SwitcherTPtr
•P   @001C02    1B8 SysEvtBuf
 H pk@018FD4    704 <.PR> @0159F4 PACK    6 "System" <0002>
   @019704     5D0
 H   @0135C0      8 <..R> @00143C entry NOT FOUND in map list
 H   @013B42    42E <.P.> @00CB7C old Width Table
```

A **bullet** in column 1 indicates an immobile block (non–relocatable or a locked relocatable block). Column 2 indicates the block type: non–relocatable and accessed via pointer,(**P**); relocatable and accessed via handle, (**H**); or free, marked by a **blank**.

The third field contains an abbreviation that describes the type of data object stored in that block. For example, the ninth entry above has **pk** in this field, indication a PACK resource. See **Program Conventions** for a list of these abbreviations.

The 4th field gives the starting address of the data of the block (not its header), in the format *@address*.

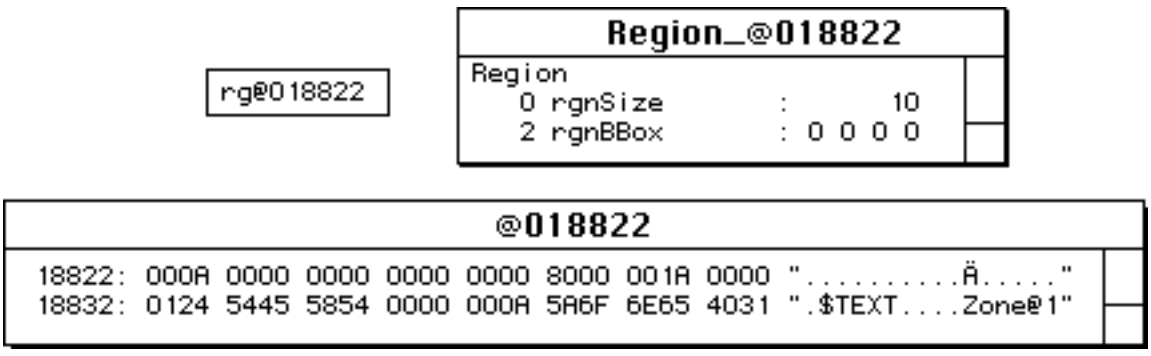
The fifth field gives the size of the block, not including its header.

The sixth field provides three flags, surrounded by < > marks. The flags are **LPR**, standing for **L**ocked, **P**urgable and **R**esource. A period in one of these flag positions means that the attribute does not apply to the block.

Next come fields that vary for different types of blocks. For relocatable blocks, there is the address of the master pointer of the block, preceded by an @. For resources, there is the resource type, the resource ID number, the name of the resource file in quotes, and the file reference number in < > marks.

Parts of some data objects indicate what they are and the data object to which they belong. In the above example on line 6, you can see that the block is a region which is the update region for a window record located at address 15462.

If you select the type *Abbreviation@address* fields, then press **Cmd–Space** (the **Tables:Fields of** command), ***The Debugger*** brings up a structured-dump window of the block. Make the same selection and press **Cmd–Shift–Space**, and you get a hex dump window of the block. Here are some examples:



The **Stack Dump** command brings up a window that displays values in the stack. Here are some sample lines from such a dump:

```
07CA1C  =41418E rPR_1087+26
07CA1A  =220041
07CA18  =017822 TUTORIAL$$+4E
07CA16  =2E0001
```

The first field in such a line gives an address on the stack. Next comes an "=" sign, followed by the contents of the four bytes that begin at that address.

Actually, the first byte (the one actually at that address) is masked out, and the next three three bytes that are located at the address plus 1 are what is shown. If ***The Debugger*** recognizes these bytes as an address inside the ROM or the PP, a third field shows up that expresses the address as a procedure name plus an offset. The values stored on the stack in such cases may be a return address.

Note also that the addresses shown in this stack dump increment two bytes at a time: this is deliberate.

The **Sts Dump** command (Ω-**Z**) brings up a status dump window with the values of the Debugger's flags and other miscellany that the Head Nose uses to debug ***The Debugger***.

The **Dbgr Tbl Dump** command brings up a dump of The Debugger's internal tables .

The **Proc rel addresses** command tells ***The Debugger*** to toggle the way it displays addresses on the left side of a code listing. Normally they are given relative to the beginning of the procedure. Select this command, and the addresses will be displayed relative to the beginning of the segment of the code. A check mark in front of this command indicates the procedure–relative option is on.

The **format Maps by Addr** command tells ***The Debugger*** to toggle the format of Display menu maps. Normally, references just give a procedure name. This command lets you see the references as: `'SegNnum/ segment-relative address'` . When it's on, you'll see a check mark before the menu item. And, if both this option and Proc rel addresses are on, the references are formatted as a procedure name, followed by a plus sign, followed by a procedure–relative offset. Here are some pictures:

normal

```
130 InitMenus     copyROM128
17B InitDialogs   copyROM128
```

seg # & seg rel addr

```
130 InitMenus     1/  8E
17B InitDialogs   1/  8C
```

proc name & proc rel addr

```
130 InitMenus     copyROM128+32
17B InitDialogs   copyROM128+30
```

The **UnLoadSeg error check** command toggles a check on use of the UnLoadSeg trap. When checked, it will pull you into ***The Debugger*** when the PP calls UnLoadSeg and is in fact trying to unload a segment that is "on the current call-chain", a fatal error.

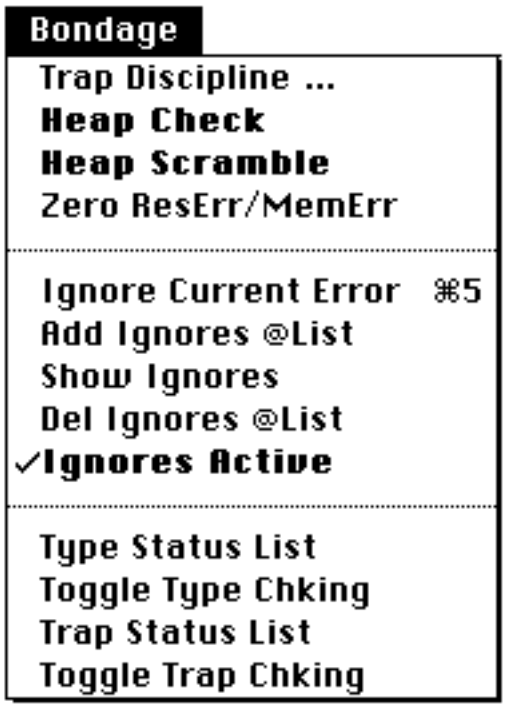
The **Flip Dynamic Wind Update** command toggles the Front Window for dynamic updating (on every entry to ***The Debugger***). A window set for dynamic updating is indicated by prefixing the title of the window with an asterisk (*). Windows generated via **Cmd-space** , the "Values" windows and the 'Stack State' may be set for Dynamic update.

The **Source Only Windows** command (Ω-**V**) sets the default mode for display of CODE windows when a ".SYM" file is associated with a task. When the checkMark is present CODE windows will opened as source only, otherwise you will get source and assembly interspersed.

The **Hex/Ascii String Srch** command (**Cmd-L**) searches memory for the specified Hexadecimal or ASCII string, and if a match is found, it places the message "Match at =nnn" in the **-Notes-** window.
Note that the ASCII-string search is case sensItive. The default values of the search range are **FBA** = 0 and **Len** = the value of BufPtr.

The **Repeat Search** command (Ω-**L**) continues the search at the next address using the last specified search pattern.

The Bondage Menu



The **Bondage Menu** provides commands that control Trap Discipline. When Trap Discipline is active, it checks the arguments to Trap Calls prior to executing them. When Discipline detects an error, it will break into ***The Debugger*** with a message in the **-Notes-** window:

```
"• Discipline, error in Type - ttt ,       TrapName @ PC = nnnn"
```

and the offending argument name will be highlighted in the **-Trap/User Call-** window. The format of that line is:

```
" nn• argName:  argValue"   .
```

In some cases Discipline will issue a more detailed message. One uses the information as follows:

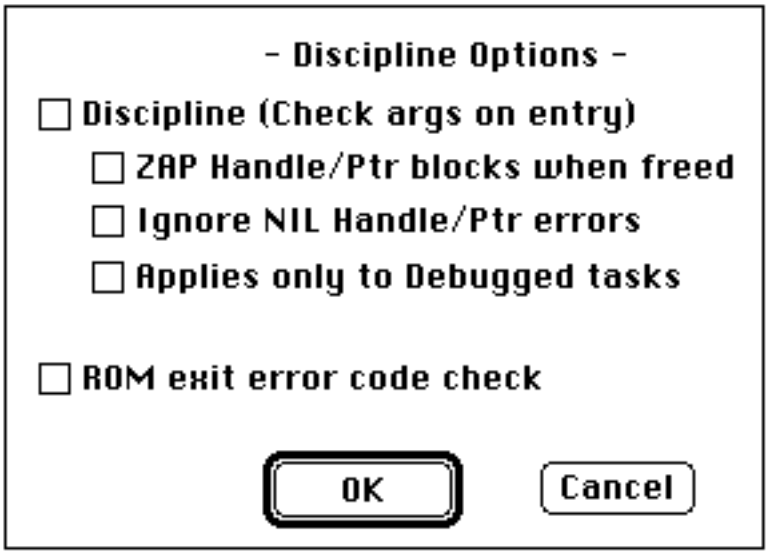
ttt is the typeName of the offending argument and **nn** (if present) is the positive offset in bytes of the argument value or address from register A7. To see its value, **Cmd-space** on the value of A7 in the **-Registers-** window.

The values of the arguments are checked for range errors. For example things that should point into the heap, or a particular heap are checked as such, etc. In the case of the "baseAddr" field of the grafPort I have omitted range checking; with Big Screens and Mac II's it can take on almost any value.

Note that some ROM trap calls are flagged as bad by Discipline, but are OK in the sense that the offending fields will be properly setup when they are used. That is, the local context in which Discipline checks the arguments is insufficient relative to the global context of the use of the field.

A case in point (on the Mac II) is _NewControl which does a _NewHandle without the clear bit set. If memory has been preset to garbage, then Discipline will detect errors in the CtlHdl record on subsequent _SetCTitle, etc trap calls. Given this behavior, the best way to minimize spurious entries into ***The Debugger*** which maximizes the amount of error checking done by Discipline is to introduce a mechanism to ignore trap calls emanating from a given address. This is the **Ignore** mechanism which occupies the middle of the **Bondage** menu. To ignore these errors on future occasions by select the "**Ignore Current Error**" command (**Cmd-5**).

Trap Discipline … - Displays a modal dialog that lets you activate discipline and options associated with it. The three indented options, are only active when Discipline is selected.



The options are:

Zap Handles - Toggles "presetting" of memory. When selected, blocks (in the heap) released by a _DisposHandle or _DisposPtr call will be preset to a garbage value so that any references to it may result in address errors. In addition, if the handle/ptr to the block being released is stored in a global variable, then that variable is also set to garbage so that future references to it will cause an error. The present value of the preset is "$fffD fffD", this value will cause Bus Errors on the 68020.
When the **Ignore NIL Handle/Ptr Errors** box is checked NIL handle/ptr errors will not be reported.
When the applies only to Debugged tasks is selected, errors will be reported only for those tasks that have an Aux file or are a Think C project.,

ROM Exit code check - Toggles checking of the results of Trap calls on exit. Trap calls which set error flags (**MemErr** or **ResErr**) or return an error code of type OSErr are checked, and will cause a break into ***The Debugger*** if they are returning an Error Code (a negative number). This option may be selected independent of Discipline.

NOTE: Rom Exit code checking DISTURBS the execution of ROM patches with "Come From" code.

The Simple Beep sound does not work on the Mac II, and on some occasions, TopMenuItem ($A0A) gets messed up, and menus which should not Scroll do. You may reset it to its correct value ($14) with the **GO:Set ...** command.

Heap Check - Check the Application Heap for consistency prior to every Trap call. Display msg in **-Notes-** if the Heap is inconsistent. To check the System Heap, **Cmd-[** of "?Heap_Check(SysZone)"

Heap Scramble - Scramble the Application Heap as an aid in detecting errors where partially dereferenced handles are being used. This command will slow the execution of the PP down to a crawl.
Hold the Shift key down to scramble the System Heap.

Zero ResErr/MemErr - Zeros the PP's copy of the System globals **ResErr** and **MemErr**

Ignore Current Error (**Cmd-5**) - The value of PC is added to the table of addresses that Discipline will ignore when doing its error checking.
This command is also used to Ignore the current Bus Error when MMU protection is selected.

Add Ignores @List - The selected list of addresses (one per line) is added to the Discipline trap address Ignore table. One may also specify types and trapnames (_tname). See the =ROMI section of the .dsi file discussion on page 32.

Show Ignores - Opens a window with the current list of **Ignore** addresses, tasks, trapnames and types. **NOTE**: the leading blank preceding the equal sign is necessary for the **.dsi** file read routine.

Del Ignores @List - Delete the selected list of addresses (one per line) from the **Discipline trap address Ignore table**. See the =ROMI section of the .dsi file discussion on page 32.

Ignores Active - A Toggle that lets one keep a set of Ignores, but makes them temporarily inactive.

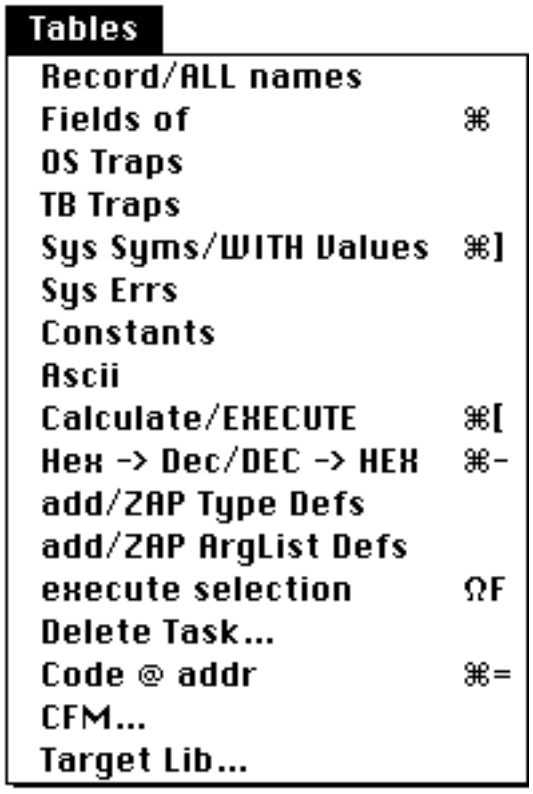
Type Status List - Displays the status (Enabled/Disabled = • / BLANK) and names of the types that Discipline checks.

Toggle Type Chking - Toggle the Enable/Disable flag for the selected types in the **-Type Status-** window. One may select a typename from ANY window and apply this command to it.

Trap Status List - Displays the status (Enabled/Disabled)and names of the Traps that Discipline checks.

Toggle Trap Chking - Toggle the Enable/Disable flag for the selected Traps in the **-Trap Status-** window. One may select a trapname from ANY window and apply this command to it.

The Tables Menu



The **Tables Menu** commands give you a number of information utilities, including a calculator and an on-line, interactive version of Inside Macintosh.

The **Record/ALL names** command displays a list of RECORD (struct's) data types that ***The Debugger*** knows about (built-in System types such as WindowRecord, ...).

If you hold down the Shift key when invoking this command, the list shows ALL data types ***The Debugger*** knows about.
If you select one of these names, then give the Fields of command, you can see a definition of the data type.

The **Fields of** command (**Cmd-space**) has varying effects, depending on the current selection. Here is a little chart (*typeName* is a data type known to the ***The Debugger*** and *address* is a hexadecimal address.) :

current selection	**Fields of** displays…
typeName	structure of a *typeName* data object
typeName@ address	structured display of *typeName* data object at *address*
address	hex/ASCII display of memory at *address*
@address	hex/ASCII display of memory at *address*
=address	unstructured disassembly starting at *address*
`c`*@address*	unstructured disassembly starting at *address*
`hz`*@address*	heap dump of a heap starting at *address*

NOTE: *address* may be **any valid expression.** For example `"*@(RA6+4)"` will generate a dynamic window tied to the frame defined by the current procedure.

If there is no current selection, the command line dialog box comes up, and you can type/paste into it.

Holding down the Shift key forces a hex/ASCII display.

Another example: Let nnn be the address of a Window, then the expression:

```
"*Region@(windowRecord(nnn).updateRgn^)"
```

will display the contents of the region as it shifts around in memory.

The **OS Traps** command brings up a display of the Operating System traps. Each trap entry gives the trap number, its name, and a Pascal–style description of any parameters and/or results. The notation used for register–based trap calls is detailed in **Program Conventions**.

The **TB Traps** command brings up a display of the Toolbox traps. Each trap entry gives the number of the trap, the name of the trap, and a Pascal–style description of any parameters and/or results. The notation used for register–based trap calls is detailed in **Program Conventions**.

The **Sys Syms/WITH Values** command (**Cmd–]**) brings up a display of the system global variables that ***The Debugger*** uses for value–to–symbol substitution. Each symbol entry has the following format:

hexAddress name type {*currentValue*} *; comment*

currentValue is shown if **Shift** is pressed when the command is invoked.

The **Sys Errs** command brings up a display of system errors. Each entry has the format::

errorCode errorName ; comment

If their is a selection in the front window (hopefully an error number), then it will be searched for in the '-Sys Errs-' Window

The **Constants** command brings up a display of constants from the Pascal "include" files.

The **Ascii** command brings up a display of decimal, ASCII, and hex equivalents.

The **Calculate/Execute** command evaluates Pascal–like expressions.

Select an **expression**, then invoke this command from the menu or by pressing **Cmd–[**.

Hold the **shift key down** to **execute** one or more **statements** in the expression language.

The expression can contain references to PP globals, System globals, IM data types, and a set of intrinsic functions described on CrossRef.
Note that the **default radix of constants is hexadecimal**, unless the constant is preceded by a #.
For function calls, the default radix of the second argument to **Bsl** , **Bsr** , and **Btst** is decimal.
The characters $ and # define the **current and ALL subsequent constants** to be hex or decimal until the next $ or # appears outside of the second arg of the *func_calls*.

Here are some examples :

expression	result of **Calculate**
`17- #13 - 10 .....`	`0 ( $17 - $0C - $0A = 0 )`
`theZone^.Moremast .....`	`40`
`WindowRecord(RA0).dataHandle^ ....`	`addr of Data handle of the WindowRecord in A0`
`Bsl($ff,4) .....`	`0ff0`

The **Convert Hex to Dec** command (**Cmd–minus**) converts a hexadecimal value into its decimal, ASCII, and system symbol equivalent., with the result placed in the "-Notes-" window.
Here is an example. Convert was invoked twice: the current selections were 258 and 124.

```
$258 =        600 '...X' DiskVars+JSetSpeed+2
$124 =        292 '...$' DskRtnAdr
```

Shift-Cmd–minus will convert a Decimal number (**nnn**) to Hex.

Add/Zap User Defs - Hold down the shift key to ZAP (remove) all User type defs, otherwise SELECT a range of Pascal like TYPE definitions to be added to ***The Debugger***

The syntax of the type definition statements is:

a) type = typex; equivalence type, typex must be defined.
b) type = ^typex; equivalence type, typex may be a forward reference
c) type = ARRAY[n..m] OF typex; equivalence type, typex must be defined.
d) typename = RECORD fieldLists END;
   All types in a fieldList must be previously defined.
e) You may reference any Type from the "Tables:Record/All Names" list.
   The "base types" are described on page 31

To display the results of Type *@address* in a "Write" statement use the function "?REC(typename,expression)" .

User defined Type names should be 3 or more chars long so that there is no conflict with the 2 char type abbreviations.

Because of ambiguities in the syntax of the **Fields of** command, the type definition of a user Type name that conflicts with valid Hexadecimal numbers can not be displayed.

Example: The type def "abc = integer;" is ambiguous to Cmd-space.

• You may want to display a record defined like this:

```
MyDptr = ^MyData;                    { forward reference of a Ptr Equiv}
MyData = RECORD
    Terecord:TeHandle;               { an Inside Mac type }
    File_Vol,windstuff:integer;      {list of field names separated by commas}
    arrx:ARRAY[1..5] of integer;     { Array with integer subscripts }
    ch_ar:ARRAY[1..5] of CHAR;       { NOTE: array is not PACKED, same for Boolean }
    ftrunc,changed,third:Boolean;    {list of Booleans}
    more:Longint;                    { this should be aligned on a Word boundary }
    s1:String[6];                    { a Pascal String,max 6 chars, occupies 8 bytes }
    cs:Cstring[8];                   { a C string, max 8 chars long, occupies 8 bytes }
    more2:Word;                      { an integer that will print in Hex format }
END;
```

If Register A0 holds the address of the record, you can display the above record like this:

write(?Rec(MyData, RA0)); OR write(RA0:MyDptr);

NOTE: write(expr:type) is setup to print the value of expr according to a type that is <= 4 bytes long. while **?Rec(type, a_expr)** will correctly display the value of an instance of any type that starts at the value of a_expr.

To clarify, assume DragRect is a global var of type Rect. Then **?Rec(Rect,@DragRect)** is the correct way to display the value of the record which is 8 bytes long. The @ sign preceding DragRect is so a_expr is the address of DragRect.

Add/Zap ArgList Defs - Hold down the shift key to ZAP (remove) all ArgList defs, otherwise SELECT a range of Pascal Proc/Function definitions. The syntax of the ArgList defs line is the same as a Pascal Procedure/Function definition. Look at the **Tables:TB Traps** window for examples.

MPW C users should note that Booleans are passed as Longwords by C.

Pascal Users note that if BB is an inner proc then the environment is passed as an argument and must be added to the list given to ***The Debugger***. For example:

```
PROCEDURE outer;
   PROCEDURE BB(anArg:integer);
      begin
      end;
 end;  {outer}
```

The declaration given to ***The Debugger*** is:

```
PROCEDURE BB(anArg:integer; Environ:Longint);
```

Execute selection (opt-Cmd-F) - execute the selected lines that are a subset of a ".dsi" file.

Delete Task … - Deletes a resource task that is no longer active from ***The Debugger***'s table of tasks (Ω-T). If the front window has a number selected, it is taken as the argument, else it brings up a dialog box.

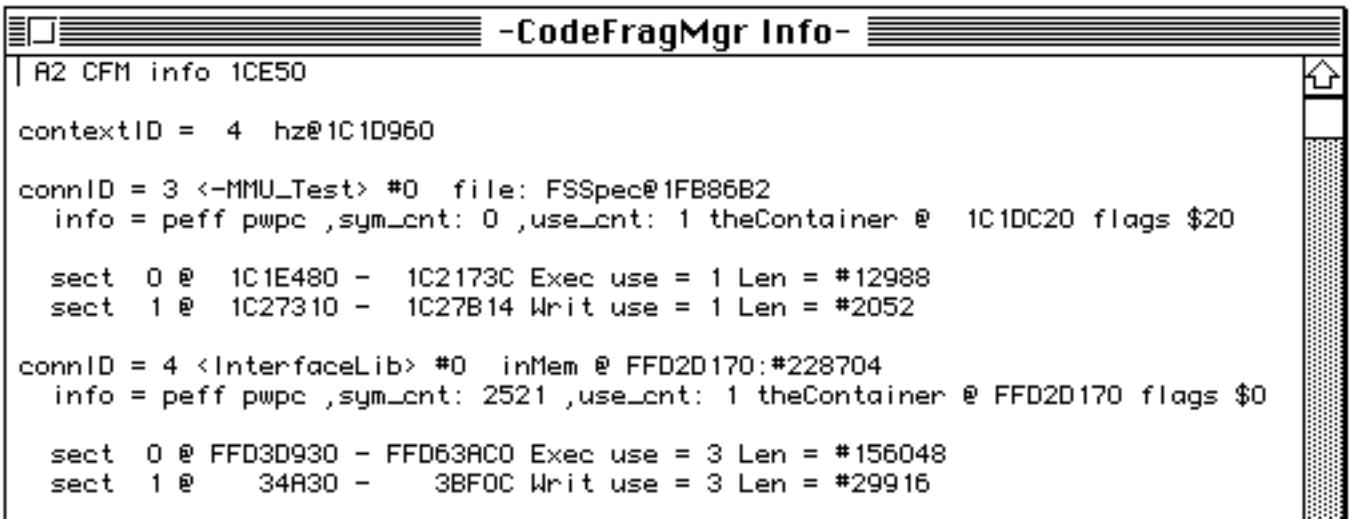
Code @ addr (**Cmd-=**) - Open up an unstructured Assembly Window of **68K** code at **addr**. Just select an address. Leading $, @ , = and blanks are ignored.

Shift-Cmd-= opens up an unstructured assembly window of **PowerPC** code at **addr**.

CFM… - Displays various Code Fragment Manager info as follows:
If nothing is hilited in the front window then it bring up a dialog box with the following options:

‘ return for All, "<LibNname" for exports, OR SymName to Find’

Pressing return brings up a window displaying ALL the CFM ‘contexts’ and connections.
This is a good starting point for most mucking around with CFM.



In the above, you see a partial display of the Link info for context 4 and the frist 2 of it’s ‘connections’ (loaded CFM Libraries).
The Library `MMU_Test` has a minus sign in front of it as it has NO exported symbols (`sym_cnt = 0`).

The ‘pwpc’ indicates that it is a PowerPC fragment.
‘theContainer @ nnnnn’ is the starting address of the CFM fragment which usually is 16 byte aligned and starts with the 4 byte id `'Joy!'` to identify it.
The code is in the `'Exec'` utable section and the Data Section is the `'Writ'` able one.
NOTE: the TOC pointer(s) for a fragment usually point into the middle of the Data Section.

Hiliting `'<InterfaceLib'` and selecting ‘CFM…’ will bring up a window with all it’s exports.

Hiliting an arbitrary symbol and selecting ‘CFM…’ will display the value of the symbol (a Tvector) and the name of the library it is in.

Target Lib … - Brings up the ‘CodeFragMgr Info-’ window so you can select a library to be debugged. After selecting the library and re-executing the comand, it will display a Standard File dialog so you associate a ‘sym’ file with the library.

Special & Command-Key Summary

Check out the reference section for a particular menu in this manual for more detail on each command. Relevant menus and commands are indicated in square brackets, like this: [Edit:Cut]

escape	Clear any hilighting in the front window
~ (tilde)	In Swap screen mode, toggle between the Debugger's and User's screens
Cmd-Click	Toggle a source code window between source only and source and assembly
Option-Click	Pop-up menu of the list of windows
Command	Pause step continuous, etc **(you must release the command key to exit The Debugger)**
Cmd-Space	Depending on the current selection, bring up a display of data [Tables:Fields of]
Shift-Cmd-Space	Bring up a Hex/ASCII display of data [Tables:Fields of]
Cmd-Comma	**Transfer control** to the PP, execute one instruction, treating a subroutine or trap call as one instruction, and return to ***The Debugger*** [Go:Step Thru]
Cmd-Period	Terminate current command, **continuous-step** mode, window generation, dialog box, etc
Cmd-?	Execute one instruction, and return to ***The Debugger*** [Go:Step Into]
Cmd-;	Bring up a dialog box for a **continuous-step** mode [Go:Step Continuous]
Shift-Cmd-;	Go directly into **continuous-step** mode with current settings [Go:Step Continuous]
Cmd-[	Evaluate the current selection (a Pascal–like expression) [Tables:Calculate]
Cmd-minus	Convert a hex number to its decimal, **ASCII**, and system symbol equiv [Tables:Convert]
Cmd-A	Select all the contents of the topmost window [Edit:Select All]
Cmd-B	Depending on the current selection, toggle or set a **breakpoint** [Stops:Toggle/Set Bkpt]
Shift-Cmd-B	Set a **conditional breakpoint** [Stops:Toggle/Set Bkpt]
Cmd-C	Copy the selection to the **clipboard** [Edit:Copy]
Cmd-D	Display a code or data block [Display:Code|Data Blk]
Shift-Cmd-D	Display a data block with the preceding code block [Display:Code|Data Blk]
Cmd-E	**Transfer control** to the PP, i.e., Exit from Debugger [Go:Exit to Program]
Cmd-F	Find a string [Edit:Find]
Shift-Cmd-F	Find a string, restrict search to label field [Edit:Find]
Cmd-G	Find the string that is in the **Clipboard,** case-insensitive [Edit:Grab Clip & Find]
Shift-Cmd-G	Find the string that is in the **Clipboard**, case sensitive [Edit:Grab Clip & Find]
Cmd-H	Bring up a window showing the **application** (PP) **heap** [Misc:Heap Dump]
Shift-Cmd-H	Bring up a window showing the **system heap** [Misc:Heap Dump]
Cmd-I	**Scroll** a window until the insertion point is visible [Edit:Show insert pt]
Cmd-J	Set a **trap intercept** via current selection or type in dialog box [Stops:Set Intercept]
Shift-Cmd-J	Set a **trap intercept** via the big dialog box [Stops:Trap Intercept]
Cmd-K	Close the **file** associated with the topmost window [File:Close]
Cmd-M	Revive the **mouse** (Mac+ only) [-D-:Fix Mouse]
Cmd-N	Copy the current selection to the **-Notes-** window [Edit:sel to Notes]
Shift-Cmd-N	Copy the current selection to **-Notes-** window and close its window [Edit:sel to Notes]

Cmd-O	Open a **file** via the standard selection dialog [File:Open...]
Shift-Cmd-O	Open the **file** whose name is selected in topmost window [File:Open...]
Cmd-P	Set the PC to the highlighted value in the frontmost (Assembly) window [Go:Set PC]
Cmd-R	Show the references to the block whose name is selected [Display:Refs to]
Cmd-S	Bring up a window listing the **code block names** [Display:Code Blks]
Cmd-T	**Scroll** the topmost window until the insertion point is at the top[Edit:insert pt to Top]
Shift-Cmd-T	**Move** to the very top of the window (**home window**) [Edit:insert pt to Top]
Cmd-U	Update the **–Values–** window [Misc:Update Values]
Shift-Cmd-U	Update the **–Local Vars–** window [Misc:Update Locals]
Cmd-V	Paste the **Clipboard** into the topmost window[Edit:Paste]
Cmd-W	Add the current selection to the **–Values–** window [Misc:add to Values]
Shift-Cmd-W	Delete the current selection from the **–Values–** window [Misc:DEL to Values]
Cmd-X	Cut the selection to the **Clipboard** [Edit:Cut]
Cmd-Z	Undo [Edit:Undo]
Cmd-5	Ignore Discipline or MMU Protection Bus Error At PC [Bondage:Ignore Error at]
Cmd-8	Toggle Dynamic Window Updating [Misc:Flip Dynamic Wind Upd]
Ω-2	Transfer from Debugger to Shell/ObjectMaster™ [-D-:Side Door to Shell]
Ω-B	set or Clear an action clause or condNum [Stops:set/CLR Bkpt condition]
Ω-D	Change current task to selected [Misc:Use Task]
Ω-F	Execute dsi commands from selected lines [Tables:Execute Selection]
Ω-G	Find the Heap & block containing the selected address [Misc:Find Block containing…]
Ω-J	Clear trap intercept for selected name [Stops:Clear Intercept]
Ω-L	Repeat last memory search from current address [Misc:Repeat Search]
Ω-P	Continue execution until selected PC is reached [Go:Until PC =]
Ω-Q	Bring up the Shutdown Debugger dialog [Files:Shutdown Debugger…]
Ω-R	Continue execution, break to *The Debugger* on exit from ROM [Go:Until Exit ROM]
Ω-S	Display window with breakpoints and action clauses [Stops:Show Bkpts]
Ω-T	Display Task and File information [Misc:Task & File info]
Ω-U	Update the **Local Vars in current proc** window [Misc:Locals in Current Proc]
Ω-V	Set default mode for Source/ASM Windows [Misc:Source only Windows]
Ω-W	Set Watchpoint [Stops:Set Watch Pt]
Ω-Z	Display Debuggers flags and some misc values [Misc:Sts Dump]