==========

====== Mac Cracking- A series on deprotection methods on the Macintosh =======

Part 1

Here it is. First in the series of the Infamous Atom's mac crack series. Some of you may have macs, others just wonder how you crack on the mac. In this series I'll attempt to show you the basics of cracking on a mac and hopefully give you an idea of the difficultly and difference between Apple][and Mac cracking.

1) Things that make Mac cracking easier than Apple][:

A) All code segments must be stored on the disk in normal format. No abnormal headers or anything that cannot be read with the normal ROM read routine.(data can be stored otherwise though)

B) Protection on the mac has used fairly simple techniques since the programmers don't know all the tricks that they do on the II

C) All disk I/O has to pass through the IWM chip and thus you can't have half and spiral tracking.

2) Things that make Mac cracking harder than Apple][:

A) No debugger(or monitor) in ROM.. Macsbug is a software debugger only and therefore can be destroyed by some nasty programs(EA does it on all of theirs)

B) Programs have much more memory to play with. Instead of 0-$C000, you have $0-$400000 (of course most of the top is ROM.)

C) Virtually no documentation on non-standard read routines. Basically have to figure it out yourself.

The first thing you need to do before attempting a crack on the Mac is learn 68000 assembly (duuh...) WELL!! don't just look at it and assume you know it!! You must really understand the addressing modes and especially how the stack is handled. EVERYTHING on the Mac uses the stack. The code you are following is constantly calculating addresses and placing them on the stack to RTS to. Also, pick up a copy of Inside Mac so you can get a basic Idea of the ROM traps (there are over 300).

If you want to look for books, I suggest 68000 assembly by Leventhal and the Inside Mac phonebook or vol.1-3 from Apple.

Next time, I'll start showing you some traps, and a little 68000, then we'll jump right in to the debugger and cracking a ware Wizardry!

====== Mac Cracking- A series on deprotection methods on the Macintosh =======

Welcome to part 2 of The Atom's guide to Mac cracking-

Today's Topic- 68000 assembly and Mac Traps

By now, I assume you have looked at the 68000 assembly language somewhat and can at least understand small sections of code. Just to clarify things, and teach you some machine-dependent ideas (for the mac that is), I'll devote this part of the cracking series to the 68000 and traps/interrupts.

First of all, the 68000 is a two instruction machine, unlike the 6502. This means that most commands have 2 arguments rather than one (as the 6502 has). IE> MOVE D0,#$1000 instead of STA #$1000. This makes life much easier, especially with the use of multiple registers. (68000 has 17, compared to the 4 on the 6502). These registers are labelled D0-7, A0-7, and the processor status reg. The registers starting with D are data registers which are 32 bits long. You can store any kind of 32 bi number in them. The address registers, (denoted by A), are also 32 bits long but can only refer to even numbered addresses and are not valid for all modes. (Don't worry if it doesn't make much sense, you'll get the hang of it.)

Sooo... Now we know about registers. How about operands? Unlike the 6502, which has a different command for each register the 68000 has a standard set of commands which can work with all the regs.

Instead of: STA $1000 or STY $1000,

You would have: MOVE D0,$1000 or MOVE D1,$1000

The MOVE command is the basic operand to move data from one place to another. Be it from reg to reg (MOVE D0,D1) or memory to memory (MOVE $1000,$2000), or whatever.

Most of the other commands are similar to 6502, and work pretty much the same (JSR, RTS, CMP, BEQ, BNE, etc.). There is one other addition to the syntax you should know: the .B, .W, and .L suffixes. These refer to byte, word, and long data commands. By adding any of these to the end of an operand, you limit the command to only that size of data. For instance, a MOVE.B D0,D1 would move the lowest 8 bits of D0 into the first 8 bits of D1. The word command uses 16 bits, and long word uses all 32. These can be added to most commands that manipulate data, like CMP, CLR, MOV, ROL, etc. If you ignore the syntax, it defaults to .W.

Now we get to Addressing modes: On the 6502, you had different syntax for addressing, and its about the same on the 68000. An indirect jump (JMP ($1000)) would become JMP ($1000)... really hard huh? You can use inderict addressing in MOVE commands also: MOVE (A0),D1. This would move the contents of whatever address was in A0 into D1. (IF $1000 was in A0, then the word at $1000 would go to D1). One note, the indirect mode can only be used with the address registers, not the data registers.

Finally, there are the auto-increment and auto-decrement indirect modes. If you did a MOVE (A0)+,D1 (and A0 was $1000), it would the contents of $1000 into D1, and then automatically increment A0 so it now points to $1002. (It increments by 2 since it moved a word (2 bytes) and each address points to a byte (in effect, its now pointing to the next WORD)). Auto-decrement works basically the same way, a MOVE -(A0),D1, would decrement the address in A0 by 2, then move the contents of $FFE into D1. The placement of the - or + is the way its set up, so you can't do a MOVE +(A0),D1 or MOV (A0)-,D1.

And a couple more sytnax things- there are no stack commands (PHA,etc.), they use the auto-inc and auto-dec modes to implement a stack with the A7 register. It's kind of complicated exactly how it works, so I won't go into it here, but just assume that MOVE D0,-(A7) pushes D0 onto the stack and MOVE (A7)+,D0 pops the value off the stack into D0. (the A7 is sometimes replaced by SP as in MOVE D0,-(SP)).

So now we know everything there is to know about 68000 assembly, right?

Wrong... but we know enough to crack something!

But before I start talking about the debugger, I'll mention something about the traps on the 68000. Since all ROM routines are called through these, it pays to know what they are.

First of all, when the 68000 finds an opcode it doesnt know (some code that doesn't translate into an executable instruction), it will look in a trap table to see if there is a replacement code for it. This way, you can implement your own 68000 commands by putting the address of your routine into the trap table and simply issuing the command. On the Mac, the trap tables are in low memory and point to ROM routines. Since the routines are always in the same place with the same ROMs, the debugger keeps a table of these traps and will actually name them for you in the code. So while listing a section of memory, you may see something like this:

```
_InitGraf

_InitFonts

_InitWindows

MOVE    #14,D0

MOVE    D0,-(SP)

_Read
```

What it is doing is calling three rom routines to initialize different sections of the window management, executing a couple of 68000 instructions and then calling the Read ROM routine to read from the disk. Fairly simple, right? It does make cracking quite a bit easier, as long as you know what most of the traps do. So be sure and have a copy of Inside Mac at hand when you start to debug/crack something.(and lots of paper).

Well, next time, we'll look at the debugger and start trying to crack a few warez...

====== Mac Cracking- A series on deprotection methods on the Macintosh =======

Part 3

Here we are again, with the 3rd installment of "Mac Cracking- man or myth?"

(or something like that).

The topic of discussion in this section will be the DEBUGGER. Otherwise known as MacsBug. If you end up doing much cracking at all, you'll begin to love (and hate) some of MacsBug's commands, and you should get fairly familiar with reading other people's 68000 code.(or compiler code).

First, a couple notes about MacsBug: To run it, you must have the MacsBug file on the disk you are booting up, and it must be named MacsBug exactly, (well, case doesnt matter, but otherwise, exactly like that). If you are using the HFS system, it must be in the system folder along with the system and finder files. Also, don't forget to install the little programmer's switch in the side of your mac. If you don't have it in, you can't even start up MacsBug!

Ok, well, I'll start by talking about how MacsBug is loaded in, set up, and then list some commands and show you some examples.

Booting up-

When the mac boots up, it reads a bunch of system related stuff from the boot disk, initializes the Font, QuickDraw, Resource and other managers, and then throws up a Dialog saying "Welcome to Macintosh". It then looks for a file name MacsBug on the disk, and if it finds it, it allocates some extra memory for it and then loads it in and sets it all up. Macsbug takes up about 40k so for large programs, running on a 128k, you may not be able to load it.(get a 512k!)

Basically what MacsBug does when it sets up is change a few pointers in low RAM that used to point to error routines to point to it. Like the error when you hit the interrupt button on the side of the computer (plus a few more). So now, when you hit the interrupt button, instead of getting a bomb dialog, a new window will pop up, covering most of the screen, with a dump of all the registers (D0-D7 and A0-A7 as well as the PC and some other info). You are now in MacsBug! You have stolen control of the 68000 from the executing program and can now debug (or crack) to your hearts content but first, you need to know the MacsBug commands!

Commands

MacsBug has a lot of commands. At least 40. It basically works the same as the monitor on a II, but with different syntax, and a bunch of nice tracing functions. They are set up in a general format of a one or two word command, followed by a few numeric parameters.

What follows is a partial list of the basic commands we will be using to crack Wizardry, and some explanation for each. To get a list of all the commands, look in your Inside Mac manual under the section called "INSIDE MACSBUG" (only in newer versions).

Oh yea, forgot to mention, I'm using MacsBug V5.1. These commands work with all versions 5.0 and higher. (You can check your version of MacsBug by typing DV <cr> at the prompt.)

(aaaa = address, nnnn = number)

Memory Commands:

DM aaaa nnnn Display memory- gives you hex dump of the bytes starting at aaaa and going up to address aaaa+nnnn
Ex. DM 0F00 10 would dump out hex from $F00 to $F10. If you leave out the nnnn, it lists the next 16 bytes. If you
leave out aaaa, it starts at the current address.

SM aaaa nn,nn1,nn2... Set memory- changes value in address aaaa to nn, then address aaaa+1 to nn1, etc.

TD Total Display- dumps out all the registers, PC and disassembles the current line.

Break Commands:

BR aaaa Sets a break point at address aaaa. When the program executes the line at aaaa, it will be interrupted and the
MacsBug window will pop up.

G aaaa Just like the Apple][, starts execution at aaaa. If you leave off aaaa, it starts where the PC last was.

GT aaaa Very nice command, starts executing at the PC, and then stops and returns to MacsBug when it gets to address
aaaa (easier than setting breakpoints).

T Trace, executes one instruction, then dumps out the registers and disassembles the next line.

MR Magic Return. If you are tracing along, and suddenly encounter a JSR, you can type MR and it will execute the
subroutine and then return you to trace mode right after it gets a RTS.

A Trap Commands:

(these are probably the most important, be sure you understand them)

AB TRAPNAME Causes the computer to halt and return to MacsBug when it sees a TRAP command that is referred to by TRAPNAME. Ex. AB READ would stop the next time the program does a READ call.

AT TRAPNAME Traces and displays the address of each call to the trap TRAPNAME. Doesnt halt though. Ex. AT EJECT would show the address of each line that the program executed that called the EJECT trap, and then continue executing the program.

AX Clear all trap commands. (so it won't stop everytime it does a READ anymore)

Disassembler commands:

IL aaaa nnnn List out disassembled code starting at address aaaa and going until address aaaa+nnnn. Just like the L command on the][. If you just enter IL <CR> it will list out the next 10 or so instructions.

So those are all the commands you need to know! there are a bunch more, but they aren't as powerful or as easy to understand as these, so you can learn them on your own.

Just to give you and example of using the debugger, I'll show the steps you might use to find the starting address of a program:

1) put MacsBug on the disk with the program you are trying to find the starting address for. (Well, call it PROGRAM.)

2) Boot up the disk, and go to the desktop.

3) Press the interrupt button- you should see a big window pop up with a dump of the registers in it.

4) Type AB INITGRAF This tells MacsBug to stop the next time it encounters an _InitGraf call. (_InitGraf is usually one of the first instructions applications run, so it will close to the starting address.)

5) Hit G to start the finder running again.

6) Double click on PROGRAM and hope for the best!

7) If all goes ok, MacsBug should pop up in a little while, displaying the address of the instruction that called _InitGraf.

8) Type IL aaaa, where aaaa is the address that MacsBug said the _InitGraf instruction was at.

9) Thats it! the beginning of the code for our application! from there, we could trace on using the T command and watch the execution of the program. Or hit G to give the program back full control of the 68000.

Don't forget to do an AX command when you are done, otherwise you will be jumping into macsbug everytime you run an application that calls _InitGraf.

Finally, I'd like to say something about MacsBug alternatives: I know of one great debugger called MCBUG. it works a lot like MacsBug, but has a few extra features that help a lot, like a built in mini-assembler, some nice launch funtions, and a few other helpful commands. It's a shareware/public domain program, so if you look around you should be able to find a copy of it on CompuServe or from a user group. It comes with docs, and installs just like MacsBug; simply rename it and boot!

====== Mac Cracking- A series on deprotection methods on the Macintosh =======

Welcome to Part 4 of Mac Cracking- your guide to fame.

In this, the fourth, and hopefully final part, we will look at Wizardry and actually remove its protection. Of course, as we all know, this is only for backup purposes, right?

So first we need to set up a copy of the disk to work on. Wizardry is an unusual protection, in that you can copy all the files off the disk, but it asks you to put the master disk back in upon boot, and then reads some bad blocks off of the master disk. The nice thing about this method is that it does not crash the machine if it can't find the master, it simply continues with a semi-demo game of Wiz. This saves a lot of time when you are constantly backtracking and reloading the files to find the protection.

Here we go!

First, sector copy (or finder copy) the files from the Wizardry disk onto a blank. Then trash the Imagewriter file (we need more space for MacsBug). Copy MacsBug onto the Wizardry disk, and then select the Wizardry file, and install a MiniFinder with only Wizardry in the selection. This way, when the disk boots up, we can set up some breakpoints while the MiniFinder is running and then execute Wizardry. If we let it boot straight into Wizardry, we would have to guess when to hit the interrupt switch and hope that we didn't miss something.

Now we have a disk to crack. Boot up the disk, and when you see the MiniFinder, hit the interrupt switch. MacsBug should come up. Type AB INITGRAF <cr>. This will find the starting address of the program by halting when it starts initializing the managers. Now type G. You are back in the MiniFinder, so double click on the Wizardry file and wait for MacsBug to regain control.

At this point, MacsBug should appear saying it halted on an _InitGraf call at location F200 (this address will be different depending on your memory size. F200 is on a 512 or plus). You can now type IL F200 to start listing the code. As we look at the code (hit return to see another 20 lines after you are done with a section) we see that the program contains no branching until address F222. The protection check is going to involve reading a sector from the disk and then branching on a result. So all we have to look for is a branch after some disk access.

Type GT F222. Wizardry loads in some resources and sets up its menus and windows. If you booted up your copy of wizardry normally, this is right before it puts up a dialog and asks for the key disk. Now we have to narrow down the search to a specific JSR. If you try GT F322, you see that it goes through the check and comes back with a message that you did not insert the master disk. This means that the JSR to the protection routine is somewhere between F222 and F322. So now we look some more!

(If you did try the GT F322, you can type EA to exit to the application, re-running Wizardry. It will abort again at the _InitGraf and then you can type GT F222 to get back to where you belong.)

By continuing this process (trying locations closer and closer to F222 in the GT command) you will eventually find that the JSR to the protection is at F31E. The program does not throw up a dialog saying you inserted the wrong disk before this JSR, but does draw a dialog at F322, the next statement. So we have tracked it down to a single JSR. Now we can have fun.

If we trace, using the T command at this subroutine, we find that it immediately executes a _LoadSeg trap. And then for some mysterious reason, MacsBug never regains control. This is the tricky part- After the JSR to EFB7A (the _LoadSeg), the program loads the protection routine from disk AND loads code into EFB7A. Since the T command replaces the code at the next instruction with a break command in order to regain control after one step, this code is loaded on top of the break command replacing it. This is why your MacsBug never comes back. There is not a break point to interrupt the program any more!

What we can do though, is interrupt with the interrupt switch after it brings up a dialog saying we inserted the wrong disk, and disassemble the code that was loaded into EFB7A. When we look there, we see it did a JMP to 13828. The code at 13828 was also loaded in with this loadseg trap, so we didnt see it before. This is the main protection routine.

But now we have a problem- how do we stop the execution of the program at 13828 so we can trace the protection and find the correct branch? We can't set a breakpoint at 13828 with GT, since it would

get replaced with the code during the _LoadSeg. And we can't stop it after the _LoadSeg since it replaces itself and any breakpoints we set after it! What do we do?? Alas, MacsBug comes through with yet another amazing command. ST. This works like the GT, but does not set a breakpoint to stop the program. (I'm not sure what it does to do this, but it uses the 68000 step flag.)

So we get back to F31E (using the EA command as before, then the GT). And type ST 13828. The reason we don't issue a ST right after F200 is that the S commands slows execution of the program noticeably. You will have to wait a couple of minutes for the ST 13828 to return to MacsBug. (The drive will make some strange noises, but don't worry, just be patient.)

After this hard work, we are now in trace mode at 13828. Hurrah! Almost there. By repeating the method we used to find the first JSR, and by reading the code, you find that the last branch that seperates a key disk dialog from the bad disk dialog is a 139D0. The BEQ +90 is executed if the protection check comes out bad. If you search the code, you see that the good code continues 3 instructions from the address the BEQ branches to. So now we simply replace the BEQ with BRA +98, and no matter what the protection check returns, everything continues fine.

Now to test it to be sure our patch works. Boot the disk from scratch and get to

the trace mode at 13828 using the commands we used before. Now type SM 139D0 60 00 00 98. This is the code for a BRA +98, which we are replacing at 139D0. Hit G, and there it is! Your copy should continue to load, and no matter what disk you put in for the master, it will thank you for inserting your "master" and continue along its merry way.

But we don't want to have to use MacsBug to do this EVERY time we boot up, so we'll change the program on the disk. First dump the memory from the instructions before and after the BEQ +90 and write them down. Then run Fedit (its a sector edito program) and open the file Wizardry on your cracking disk. Do a hex search for the values- 0A 00 00 01 67 00 00 90 30 2E FF F2 (These are the bytes surrounding the instruction that you wrote down earlier. By searching for the whole string, we are sure we have the correct BEQ +92 in the program, in case there is more than one). Go in to hex edit mode and replace the 67 00 00 90 with 60 00 00 98 and write thesector back out.

Congratulations, you have successfully cracked Wizardry. You can copy the Wizardry file you patched onto your master disk, o make the patch to a sector copy of the original to get the disk back looking like it normally did. (Auto boot into Wizardry, no MacsBug or MiniFinder, and with an ImageWriter file.)

So, concluding this discussion, I'll say that these are the methods that work for me, but you are welcome to try anything else Mac Nosy is a good program for disassembling code you are trying to unprotect. And other protection schemes are very different from Wizardry's. So practice on some other programs and with any luck, you'll be cracking everything you can get your hands on.

Also, a few problems with the above crack I would like to note. Although it does work fine (I've killed Werdna on a cracked version), it is annoying to see the 3 dialogs at the beginning and also have it eject the disk twice. For further study, you might consider taking out the Ejects, so you don't have to re-insert the disk.

(Hint: you will not only have to take out the _Eject's, but also the routine that waits for another disk to be inserted. Since it doesnt eject any more, there isn't any way to insert a disk!)

Just to help you along, I'll give the patches to remove the ejects and wait for insert disk routines:

Search for -> change to

20 5F A0 17 3E 80 4E D1 4E 56 -> 20 5F 3E BC 00 00

21 6E 00 0A 00 12 A0 17 3D 40 -> 21 6E 00 0A 00 12 4E 71

2F 2E 00 08 4E BA FF 66 A8 5E -> 2F 2E 00 08 4E 71 4E 71

FF F4 66 20 48 7A 06 7E 48 7A -> FF F4 4E 71

(Just search for the first part, then change the bytes that are different in the second part. 4 patches in all plus the main protection patch.)

This introductory file is by no means the last word on assembly, cracking, etc. Some of the ideas of the Mac were simplified in order to bring you up to a good level of proficiency in a short time. Some of the functions do not work exactly as I outlined, but the ideas I presented are close enough for those who are beginners to the Mac world. If you are interested in the real inner workings of the Mac, I suggest getting the Macintosh Revealed books from Hayden. They explain the traps and ROM routines in greater detail.

==========

Copy Info

There are several programs currently that don't seem to be fully crackable. By bit copying the one or two protected tracks and making a couple of patches, you can make them easily copyable, even though the original can't be copied at all. These programs all use protection from the same company, and it works like this:

A nibble read of the protected track is done, then a search is made for the string ABCDEFEF where a data marker should be. They also write over low memory pointers that the debugger uses, so that the debugger will crash. Programs like this are:

HARRIER, ROUGE, GRID WARS, WINTER GAMES, ETC.

All have the following strings, which if you NOP them, will not destroy the debugger

Search for:2489 51C8 FFF2 46C3 Change to:4E71

Search for:12D8 51C8 FFFC 46C3 Change to:4E71

Once you find the code that must be changed to unprotect, it is found to be encoded on the disk, and decoded just before i is used. On the HIPPO ALMANAC, the data was not only encoded, but the block of memory that it was in was reversed enc for end.

==========

Eve Protection Scheme

There is a way to get past the EVE protection scheme, but it's a bitch.From what I understand, you need to decompile the routine that checks for the dongle, and them re-assign it to check that the machine has a simple serial port as opposed to the info on the dongle itself. That info is virtually unreadable as it's encrypted, and pretty much useless to anything except the app that's looking for it because it's mainly using for app reference.

==========

Faces

Search for:4240 4840 80FC 0030 4840 3D40 Change to:4280 etc.

It will ask you who beat Napoleon at Waterloo. The answer is Welling.

or

This program was pretty tough to crack until I first did Welltris. Why? Because it takes your password, stores it in memory then compares it to the correct answer (only the first 4 digits, making the password a nice simple Long). However, if you do a simple crack it will then say bring you up to the screen with the start game button. If then checks the password again - i it's wrong, then the program corrupts itself (thanks to SAM for telling me that!). This is very similar to Welltris, so I was armed for the job.So, after the _GetIText the program pushes the return address (A5-142A) onto the stack, then you password (A5-113). A JSR then stores your password with only the first 4 bytes, in lower-case at A5-142A. My simple crack simply pushes the correct password (A5-12C), instead of your password. Then, when ever the program compares what should be *your* password, it's actually comparing it to itself! Har har.

KRAK PROCEDURE

The protection in Faces is fairly typical. The password dialog does not affect your game, it just compares your passwords and has a local variable to say whether to quit or not, and it also stores your password elsewhere to be checked later.

The GetIText was also really close after the ModalDialog to make cracking quite simple.

This double password thing had me confused for a while because occasionaly it would corrupt on me, so I had to open the damn archive again! I knew where to go in MacsBug, yet I just could figure how it knew to kill itself!

This crack was not too advanced, yet not too simple. I thank it's creators for making the PEA so simple to find. Although this program had no definate crack point, such as an _ExitToShell, or a BNE/BEQ/TST, etc. it was fairly obvious with the BLT that it's only concerned with the first 4 chars, therefore making it easy to find where it is actually moving and comparing memory.

I couldn't find the exact place to not show the Dialog, so I merely jumped over the ModalDialog routine, so you will see it flash onto the screen then disappear. That's OK.

Now, since I don't have a color Mac, I had to absolutely guess at cracking that one. It looks almost exactly the same in the copy protection routine, except it's merely a few bytes down in CODE 3. If the crack doesn't work, don't blame me, just call me at christmas, and maybe I'll have a CQD machine.

Here is the complete crack:

CRACK PATCH

Open Faces 1.0 with ResEdit

Open CODE 3

Change CODE 3+$652 (just a few characters over from $650)

from: 486D 01CA

to: 603A 4E71

AND

Change CODE 3+$694 (just a few characters over from $690)

from: FEED

to: FED4

Open Color Faces 1.0 with ResEdit

Open CODE 3

Change CODE 3+$7C2 (just a few characters over from $7C0)

from: 486D 017A

to: 603A 4E71

AND

Change CODE 3+$804 (just a few characters over from $800)

from: FEED

to: FED4

==========

Fileguard

How to beat Fileguard!

Step one: Make a system disk with norton util, AND HDT

Restart machine holding down command-option-shift-delete

Run HDT, select "Install HDT Driver" (on the protected, but unmmounted volume)

Restart, File guard wont bother you anymore.

==========

Hard Disk Ejects

It has come to our attention that many games are obnoxious when run (in cracked form) on a hard disk. These games cause a warm reboot and bring down the hard disk. The solution -> use FEdit to scan the games for OS Trap A017 (_Eject) and replace it with ADF4 (_ReturnToFinder).

For example...

Game Fedit	File Block	Byte# Was	Replace With
MacAttack	00C	00BE	A017 ADF4
Frogger	00C	005C	A017 ADF4
Frogger	00D	01B4	A017 ADF4

==========

Image Express

Background

~~~~~~~~~~

Image Express uses the Eve Protection Scheme. Basically the protection scheme consists of 3 parts. 1) The Hardware device, 2) The Eve Init, and 3) the code inside each application which compares the values.

The hardware device which hooks up to the ADB port houses a chip that I assume has a value or a resource on it. I say assume because I do not have one in my possession to check out.

The Eve init is basically a driver that gets installed onto the Eve at bootup. And remains there until powerdown.

The Apps which are protected with Eve will read from the Eve and compare
~~~~~~~~~~

values, if they match it will continue on with the program as normal if they

don't then a dialog will surface telling you that you either hooked up Eve

wrong or the init is not installed. If there is no Eve hooked up the same

dialog will surface telling you there is no Hardware key hooked up. etc.

Image Express is made up of 9 files in all. They are Camera, Demo Projector, Image Express, Transporter, Image Express Launcher, Camera Launcher, Projector Launcher, Transporter Launcher and Eve Init. All of these Apps except Demo Launcher are protected and have checks for Eve in there Apps at least once.

The launchers all use the same exact protection scheme but I have not really worked on them too much and so I have not cracked any of them. The Eve Init is basically only values and some code in the crack I am working on the Eve Init will no longer be needed. The Camera application is somewhat tougher than the other applications; I have yet to crack it but I am slowly making progress. I have cracked the Image Express App, the Projector App and the Transporter app. Here are the hex changes:

Image Express

Search:3B5F F33A 4A6D F33A 6622

Change: 4E71

Search:4EAD 0B62 4EAD 166A 4EAD 16DA

Change:4E71 4E71 4E71 4E71 4E71 4E71

Search:4EAD 257A A9F4 4E5E 4E75

Change: 4E71

Projector:

Search:4A6D FDCE 664C 4EAD 056A

Change: 4E71

Transporter:

Search:3B5F FBB8 4A6D FBB8 6600 01E6

Change: 4E71 4E71

Search:57C0 4A00 6700 010E

Change: 4E71 4E71

==========

Infini-D 1.0 01-1400-6350 31-9326-1679

Infini-D 1.02/1.1.1 31-9326-1679

Infini-D 1.02

Protection Scheme: Serial Number

Supplier: Far Side

Problem : Ok, a cool user on my board, Far Side, logged on new and since I was around spying on him, I decided to validated him before he got on. I should do, Infini-D1.02. I unpacked it and booted it sometime thereafter and noticed that the first thing that happens is a Serial Number Registration box comes up and asks me for my Name, Organization and Serial Number. Ug. Playing around with it, like I usually do, I noticed that if you try to hack out a Serial Number, it will just stay there like nothing happened. This is useful in determining how to go about cracking it because now I know I can not just search for an _ExitToShell Trap (Hex-A9F4). By the way, I tried putting in some earlier version's serial numbers but they do not work on the new version (1.02) at all.

Solution : So I cancelled out of the registration thing and got back to the Finder where I jumped into MacsBug. Then I set a trap for _InitGraf (Hex-A86E) which is usually the first Trap executed by every program to init the graphics screen and what not. From here I double-clicked the App (Infini-D 1.02) and the cursor changed a little and found the trap. I scanned through a little, going nice and slow as to be careful noticing which JSR or JMP or branch would be the one to go to the Registration dialog. I went a little further, then it happened, so I went into MacsBug and found this as the last instruction before the dialog box appeared:

Addr	Instruction	Hex Bytes
Who Cares	JSR $0798	4EBA 0796

Ahh! How easy? It's a JSR, so all we have to do is change those bytes (4EBA 0796) to two No Operations (4E71 4E71). This crack was extremely easy, to tell you the truth. But it was fun none the less. I hope you enjoy it.

Hex Changes:

Search for: 4EBA 0796

(You should find this 2 times, at Sectors 392 & 3FE)

Change to : 4E71 4E71

==========

MINIFINDER ZAP

For 3390 byte MiniFinder created Feb. 19, 1904 4:40 AM by CCOM. (If you don't have this version don't even think of trying the following.) Important note! Use Fedit or an equivalent to zap the file. Block 1, positions 88 and 89 contain the number of file types that MiniFinder looks for on a disk. Change Block 1 Position 89 from $01 to $02. Block 5, positions 408 through 416 contain the type list and other apparently useless information (I zeroed almost everything out and it still worked). Change Block 5 Positions 408 to 416

From: $00 $00 $6D $46 $69 $6E $64 $65 $72

To: $46 $4E $44 $52 $00 $00 $00 $00 $00

And that's it!

Now The Finder of other disks will appear in the scoll window of MiniFinder when it runs. One problem though, make sure you have two drives. When the Finder of another disk is chosen to be opened, MiniFinder begins to execute it and then asks for the disk MiniFinder is on to be inserted into a drive if it is not online. If you do this by having to eject the disk with the Finder that you wnat to boot, MiniFinder gets confused and crashes.

==========

MultiDisk Partition Cracks

Here’s how to do a MultiDisk Partion Crack:

The way you "crack" Performer is with MultiDisk. If you don't know what I

mean by that, he is the routine. Install The MultiDisk INIT, and DA. Reboot. Go to the DA. Create a partition that is a little bit large than Pedrformer (and it's few extra files) needs. Mount it. Insert the Perfromer disk. Make sure it is NOT write protected. Launch off the floppy. It will take you first to the HARD DISK INSTALL screen. Install onto the partition. Quit the installer. Unmount the FLOPPY. Check out the partition, see if the install works (it will...). Drag the partition to the trash (unmount it.). Open MacTools, etc., or whatever program you have that will, 1- let you see invisible files, and, 2- let you "uncheck" tthe invisible box, so as to make them visible on the desktop. Find the MultiDisk partition. It will be called "MultiDisk Partition000000xxxxx" (lots of numbers, no matter...). Make it visible. Quit MacTools. Back to Finder. Finder the partition FILE. Drag it into a folder. VERY IMPORTANT, you must get this file off the desktop, by dragging it into a folder. If you don't, it will not copy properly! Open the folder you put it in. Slect the file. Hit command-D, and enjoy watching it copy. Do it again, even. The last few numbers on the file will get replaced with the word copy. No worry, just delete the letters c-o-p-y, and all will be ok. Now, get your floppy install back, so you can return it to whoever let you borrow it. Drag one of these copies out of the folder, open MultiDisk DA, and mount it. Insert the orig floppy. Launch off the floppy, as before. This time, REMOVE the install. Quit installer. Take out the floppy, write protect it, and give back to friend. It is now as it was before. Drag the partition to the trash (unmount). Also, Drag the MultiDisk FILE (the one on with all those numbers) to the trash. After all, you just removed the install from it, it is now worthless. Drag another one of the copies out of the folder, use the DA to mount it, and you now have Performer (or Vision, or anything else you desire). Copy protection bites the dust!!!! NOW, the files here on the Nest, like Performer, Vision, etc., are already partitions, you only need to dl MultiDisk. Make sure you keep a backup of MultiDisk, it likes to corrupt itself once in a while, and it will freak out if you are one a network.

==========

Network Protection Scheme

Cracking a Network protection scheme is pretty easy. Registration on any LocalTalk network has to be done by a single ROM call NBPRegister. The passed parameters to NBPRegister consist of a pointer to what is called an ABusRecord and a Boolean. Ignore the Boolean and look in the ABusRecord for another pointer called the nbpEntityPtr. (The newer versions of Nosy should decode these data structures for you.) Now find the data that is being passed in with the nbpEntityPtr (look for a PEA instruction), go to that data which will consist of a packed set of bytes corresponding to name:object:zone. Change the name, leaving object and zone alone. Presto Jerry and the thing now cannot recognize itself as self.

==========

Painter 2.0 0011187QBO (The last 3 are letters)

It's a bit of a tough one for you to try as a first crack, anyways i looked atit briefly the other day and this seemed to work, later i realized that saving and printing didn't work and so it must have gone to demo - i made no attempt to comprehend or decompile what the hash routine was, rather i simply tried to disable ever single check, so that wrong means right type thing. I'm sure this is very close to the real thing that is needed.

Painter 2.0

GWIL

GWIL main CHANGES

+21A +FE 6718->6018

+286 +16A 6FB8->4E71

+2E2 +1C6 6FAC->4E71

+322 +206 6600 FEF8 -> 4E71 4E71

+3DA +2BE 6EB6->4E71

+40A +2EE 670C->4E71

CODE 2

+12EA 6600 FE46 -> 4E71 4E71

+131C 6FDB -> 4E71

+134C 6FD8 -> 4E71

oh and a tip , whenever using this type of crack info use resorcer instead of resedit as it has a better faster interface. Also whenever you encouter code in a non-"CODE" resource, simply go to the preferences in resorcerer and tell it that , in this case "GWIL" is a pseudonym for "CODE". ie add "GWIL" to the list, then you can view it as a code resource.

==========

Panorama II

Crack

I jumped into MacsBug and set a trap for _InitGraf, which is normally how ever program starts off. It went fine and I traced the code, but unfortunately, after tracing through, it killed me back to the Finder. They obviously protected against tracing. No prob, I clear the trap table, and then rung up an _ExitToShell trap. Then I ran the program and hit return and MacsBug caught the _ExitToShell call being executed. It looked something like this:

Addr	Instruction	Hex Bytes
004A55A4	_ExitToShell	A9F4
004A55A6	CMPI.L #$00000001,-$6F52 (A4)	0CAC 0000 0001

004A55AE BNE.S +$001C 661A

So, I wrote down the hex bytes for a backup and then went to DisAsm to scan for _ExitToShell traps (A9F4 in Hex). I found 2 right off the bat but they weren't the ones I was interested in because they didn't have the same bytes following them as the code above. So I kept on searching until DisAsm bombed into oblivion with an "Out of Memory xxxx" error or something or other. I hate that! In any case, I booted up my FEdit+ 3.21 and searched for the bytes I needed and found them on secto 494 of the Panorama Application. So I changed the ExitToShell Trap (hex A9F4) to a NOP (hex 4E71) and booted the program. The configuration dialog came up again and when I hit return, the program didn't quit but went on to the rest o the program the right way. The APP was CRACKED! Yes! But like every good cracker knows this crack was half ass because the dialog should not even come up in a well cracked ware. So I went back to FEdit and looked around at where I made the changes. And found out a couple of bytes before where I made changes was where the program put the dialog up. So here are the complete changes to the COMPLETE Panorama II Crack:

Hex Changes

Search for: 6730 4EBA 02E8 4AAC 90AE 6602 A9F4 (Found on sector 494)

Change to : 6730 4E71 4E71 4E71 4E71 4E71 4E71

And the sn# is not the only thing you need to run panorama II. Your Name, Company, Phone #, and SN# are coded into the application so you need to use the crack.

But if you want your name SN# etc. in the program just look for the Panorama II prefs in the Prefs folder and if it's there delete it. Then run a virgin copy of Panorama II and type in anything you want and say (OK or Cancel) then patch the Program and the patched Program uses the Pref file you created in the Prefs folder and Voila

==========

QuickFormat 7.0

Protection Scheme: Key Code Required to use special features

Scenario

Ok, I was calling around a few boards and as usual I make my rounds on the Buzzard's Nest(s). When calling BN Central noticed someone posted asking me to remove the protection scheme from Quick Format 7.0. So I got it, and I have finally gotten around to cracking it today. Oh well, sue me!

Problem

When launching the program an annoying dialog box comes up asking you to register the program with a KEY CODE to use the advanced features of the package. If you typed in an INVLALID key code it will let you use the program with the few less sophisticated functions.

Solution

So I quit out of the program and then jumped into MacsBug. I set a trap for the _INITGRAF (Hex-A86E) and I double-clicked the Quick Format 7.0 Application.

I traced through to a very suspicious part of code that looked something like this:

Addr Instruction Hex Bytes

583834	JSR SETUPMEN	4EBA FE14
583838	JSR INITIALI	4EAD 02E2
58383C	JSR INITGLOB	4EBA FEBE
583840	JSR VIRALCHE	4EBA FF22
583844	JSR CHECKMOD	4EBA F82A

Now, its pretty obviouse from just looking at the labels that they used that you can determine what is going on. In mos cases people would not use LABELS like the ones used above, but since it is shareware and not a $500 commercial package I can see why the author opted the easier route for programming ease. The first JSR would probably be him initializing his menus and stuff. The second JSR would be to initialize the screen and the fonts or whatever, the third JSR would be Initializing the global variables he would need and the fourth would be to check for any virus, persay. The fifth however is the routine he uses to check if the program has been Registered and brings up the dialog asking you to enter a key code. I it hasn't be registered with the correct keycode the program turns off some options. But, that is not necessary, as by omitting this JSR CHECKMOD you will remove the check and the program will run with all options available. Neat, eh?

Byte Changes (You should find the SEARCH string only ONE TIME!)

Search: 4EBA FE14 4EAD 02E2 4EBA FEBE 4EBA FF22 4EBA F82A

Change: 4E71 4E71

QuickFormat 7.1

CODE 3 +$3B6 From $6606 to $4E71

==========

Railroad Tychoon

Find Hex string 0684 6646 486E and replace it with 0684 4E71 486E. When you hit the choose the locomotive screen, any choice you make will be valid. I've since heard that the krak listed above allows the game to continue to limit your play to two trains at a time. A bad krak in other words.

Protection Background

I sat down here a few hours ago with the intention of blasting a quick hole in the protection. It's now five in the morning This is a questionable krak. The first anonymous attempt reportedly didn't work. I'm not sure if this second attempt (mine) will work either. Understanding why means taking a much closer look at the logic of the program.

Krak Procedure

The protection in Railroad Tycoon is not typical. The screen that comes up asking for you to identify the type of locomotive is crucial to the operation of the protection. It's also very difficult to remove from the flow of execution - at least I haven't been able to sidestep it. Your choice sets up certain values in certain global variables for the program to use later.

There are two separate response handling routines. One routine, called BCONTENT (accessed by a JSR CA(A5)), handles the response if you click the mouse. This routines highlights the name choices of the locomotives as you click them. It will also handle a click on the OK button. The other routine, called BKEY (accessed by JSR E2(A5)), handles keyboard input. I suppose it's possible, though I haven't tried, to choose the locomotive name under the direction of the arrow keys. This second routine also handles a return (hit OK).

This double routine thing had me confused for at least an hour because I would randomly (and unconsciously) choose the OK button, sometimes with the mouse and sometimes with the keyboard. I kept setting breakpoints where I knew execution would occur (I knew this because I had painstakingly traced through the whole code block, picking out good breakpoints) but I had only done so for the mouse routine. I kept confusing myself by randomly selecting the OK button with the keyboard (sometimes TMON would break in, sometimes it wouldn't, on a seemingly random basis)!

I began looking at the protection with the idea that the protection was a subroutine. It is not. The whole initialization of the game is all in one routine, with the protection at the leading edge of the code segment. This pre-misconception (at midnight) proved time consuming. (I'm telling you about all these traps I fell into so you can use the experience in your kraks.) The fact that the protection is built into the main initializing code segment makes the whole approach to kraking it much more involved. As I said above, I was looking for a "quickrak" [hmmm, a new word], but this just was not going to be the case.

I stubbornly kept looking for a definitive krak point - one of those luscious, helpful, welcome conditional branches that, when satisfied, completely jumps around an undesirable bunch of code (the code bunch that says, "Nope. You're either blind or a pirate with no manual, so you will have only two trains"). At about four o'clock, it finally dawned on me (pardon the pun, the sun is just coming up now as I type this) that the protection was hard-wired into the logic of the program. When I say this, I mean that there is not any clear-cut result from your choice.

In particular, this protection scheme (as I said above, or at least think I said above) takes your choice and puts a bunch of corresponding values into a bunch of global variables. Later on, the protection code then goes on and does a bunch of math operations on those values to come up with, not a condition, but data addresses and pointers for a dialog box.

The protection always draws a dialog box (regardless of whether or not you make the right choice). The only thing that is different when you make the right choice is the contents of that dialog box, as computed from the numerical values assigned to your choice.

Are you confused? The protection is not saying yes and it's not saying no. It's using your choice to figure out what to say in its dialog box (the gray one that pops up right after you make your choice). Because figuring out all that crumby math logic is way beyond my effort limits (especially for a game like this), the program always fills its dialog box with the message corresponding to an incorrect choice. The protection always tells you that you will be limited to two trains and there is nothing I am going to do about it.

A bit farther down in the code from the _ModalDialog (the trap call which waits for you to acknowledge reading the dialog box's message with a return) is one of those luscious, helpful, welcome conditional branch statements I'm always looking for. Need I say that I was very happy to see it?

I changed the BNE at "BSWITCH"+AC4 to an NOP. Here's where the ambiguity comes in. I *think* this modification forces the program to always act as if you've made the correct choice for the locomotive's name. (If this modification does in fact do this, then it constitutes a "krak"! If it does not, well, then it's your turn to take a shot at this thing.) In more words, despite the fact that the program's dialog box says you've made the wrong choice and will be limited to only two trains, this little "krak" makes the program a liar. You should be able to play without any software protection limitations to the number of trains.

To be honest, I do not have the patience right now to figure out the game, to play it, or to test the krak. I'm way too tired. Additionally, I feel fairly certain that I will not have the slightest interest in figuring out the game, to play it, to test the krak tomorrow, or the next day, or even the next day. (I dunno. I'm hedonistic when it comes to computer games. People tell me Robot Wars and SSI games are great. I tell them they're nuts - I'm action arcade all the way. Besides, nothing turns me of faster to a game than a pathetically slow opening graphics presentation, and Railroad Tycoon definitely takes the cake (and whole damn cooking pan and spatula) when it comes to pathetically slow opening graphics presentations! Have you eve seen anything slower?!?)

In all fairness (actually curiosity), I went back and looked through the game for the krak patch that Grimm posted. It's there all right, but I believe it is in the wrong place. If you search for what he says, but search for it twice, the second occurrence is remarkably close to where my krak patch is.

Krak Patch

Hex search for:

28 00 02 B0 69 06 84 **66 08** 08 AD

Hex change the bold values to 4E 71

This should appear on sector 301, byte 25848 of the file.

or

Find Hex string 0684 6646 486E

Replace it with 0684 4E71 486E

When you hit the choose the locomotive screen, any choice you make will be valid The krak listed above allows the game to continue to limit your play to two trains at a time.

==========

Claris Resolve 1.0

Protection: Date Expiration

Supplied by: Zelig

Problem: Resolve will expire August 10, 1991 unless a valid serial number is entered.

Solution: I was having problems finding out where the check was taking place so I went the easy route and set a trap in MacsBug for an _ExitToShell (A9F4) and then back traced using "IL" to list the code before the _ExitToShell. And this is what was revealed:

```
PEA $FFFC                    ;486E FFFC

JSR Timebomb                 ;4EBA FCF6
```

Now, this is extremely suspicious. All you have to do is change the JSR Timebomb to NOPs and a branch right afterwards that will make Resolve work forever.

Hex Changes:

Search : 486E FFFC 4EBA FCF6 4A00 588F 670E

Change : 4E71 4E71 600E

==========

Shutdown Code

For you people with hard disks and are running MacBugs or apple debugger and when you crash and are not able to exit to shell (Finder) here is the code to type to unmounted all volume and then do a shutdown this will help to recover faster by not having to rebuilding the desktop after your crash:

SM A78 3F3C 0002 A895 (THEN A RETURN)(YOU HAVE TO PUT THE SPACES IN)

G A78 (THEN ANOTHER RETURN) (YOU JUST DID A SHUTDOWN NOW)

==========

StuffIt Lite 3.0

If you have a registerred copy of Stuffit Lite 3.0 and you want to change the registration info, change the serial number that is in the data fork of the Stuffit Lite 3.0 application at the offset 836 ($344) from the start of the data fork (which is at the end). The registration name is in the middle of the data fork.

If your copy of Stuffit Lite 3.0 has been patched by the Stuffit Lite 3.0 crack program, you can change it back to normal by changing CODE 13+$1208 from $4E71 to $6622. Unpatched Stuffit Lite programs won't have the same CODE 13 resource (see next paragraph). I tried this patch before I got the Stuffit Lite 3.0 crack and found that it didn't enable encryption or multiple open files so it was useless. I threw away the Stuffit Lite 3.0 crack because of this and it did the same thing I already tried.

Stuffit Lite 3.0 contains some compressed resources that include most CODE resources (including CODE 13) and some DITL resources and probably others. (This is why trying to compress the file only gives about 3% saved.) This compression effectively scrambles them so they appear virtually meaningless. Stuffit uncompresses them when you run the program. The CODE 13 resource that was used in the patch was an expanded version of the CODE 13 resource in unpatched programs. The person who made the patch got the uncompressed resources from memory when the program was running and worked with them. (I made an FKEY that will retrieve resources from memory). You can't patch your unpatched program without the expanded CODE resource or the crack program. Surprisingly, Stuffit doesn't care if it's resources are expanded even though they should be compressed.

Here are the serial numbers. They were created using numbers from 0 to 199.

(most should work):

L297000000	L347000001	L397000002	L447000003	L497000004	L547000005
L597000006	L647000007	L697000008	L747000009	L267000010	L317000011
L367000012	L417000013	L467000014	L517000015	L567000016	L617000017
L667000018	L717000019	L237000020	L287000021	L337000022	L387000023
L437000024	L487000025	L537000026	L587000027	L637000028	L687000029

L207000030 L257000031 L307000032 L357000033 L407000034 L457000035

L507000036 L557000037 L607000038 L657000039 L177000040 L227000041

L277000042 L327000043 L377000044 L427000045 L477000046 L527000047

L577000048 L627000049 L147000050 L197000051 L247000052 L297000053

L347000054 L397000055 L447000056 L497000057 L547000058 L597000059

==========

Assembly for Cracking

by THE SHEPHERD

This document is a short tutorial designed to prepare the reader to use TMON and MacNosy to render protection schemes inoperative. It will not prepare the reader to begin programming in assembly language, in fact, I am not a programmer myself. Hopefully this will allow someone with a minimum programming background to learn how to quickly read assembly listings, and then quickly locate a give protection scheme. Actual cracking will not be covered in detail in this document.

The following topics will be discussed in detail:

Number Systems and Memory

Basic Architecture and Addressing Schemes

Instruction operands and parameters

The Flags Register

The Stack

Traps

Assembly Mnemonics

How To: MacNosy

Example Code

How To: TMON 2.8.x

How to Crack Sorcerer: A Test Cruise.

THE BASICS

Number Systems

We will be dealing with three different number systems. The difference between the number systems is simply at which number one decides to carry into the next column. In Decimal (the first system), we carry at the 10th number. That is, any given digit can only hold 10 values, namely, the numbers 0 - 9. Once we get to the carry value, we carry a one into the next column and reset the previous column to zero which is precisely what happens when you go from 9 to 10 (or 99 to 100 in which you carry twice, etc).

The second number system is called binary. In this system, the carry value is 2. This means that a given digit (called a bit in binary) can hold 2 values: 0 and 1. To add one to a number in binary, you use the same principle as in decimal, except that the carry is a different value. To add 1 to 8 in decimal, you just add 1 and there is no carry (because the ones column hasn't reached the carry value (10) yet). To add 1 to 9 in decimal, you have to carry the one to the next column (because you have passed the carry value) and reset the ones column to 0. So, counting in binary looks like this:

0	
1	Add one to zero: we haven't reached the carry value (2) yet.
10	Add one to one: now we have 2 so we have to carry one to the next column and reset the first column.
11	Add one to zero (in the first column) and you just get one.
100	Add one to the first column and you get 2 so carry 1 to the second column and zero the first column. Add the carried one from the first column to the second column and you are adding 1 + 1 which is 2 - carry again. So, carry the one to the third column and zero the second column.
101	And so on...
110	
111	
1000	
1001	
1010	
1011	
1100	
1101	
1110	
1111	And here we are at 15 decimal.

OK, we refer to binary because it is the native numbering system of the computer and also because in some of the instructions, the individual bits represent different information. Unfortunately, binary is hell for us humans. That brings

us to the third major numbering which is hell for the computer AND hell for us! But both sides can deal so it's not too bad.

Hexadecimal is the third system and its carry value is, of all things, 16. Now, we don't have 15 digits so hexadecimal uses the letters A-F for its last values. Here is how to count in hexadecimal;

Hex	Decimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C	12	1100
D	13	1101
E	14	1110
F	15	1111
10	16	10000

And so on. You may be wondering what the hell is so great about hex numbering. Well, it turns out that one hex digit can account for 4 binary digits (whereas decimal cannot hold a whole number of binary digits). This makes it extremely easy to convert binary to hex and back. To convert to hex from binary, just take the right-most 4 digits and convert it to its equivalent hex digit with the above table. Then do the same for the next 4 binary digits and keep going until you are out of binary digits. For example: 1010111000110001101. Break it up as follows: 101 0111 0001 1000 1101 and convert each group of 4 into a hex digit: 6 7 1 8 D so the hex number is 6718D. Easy right?

To go back to binary, take each individual hex digit and convert it to its equivalent binary code.

Signed Numbers and 2's Complement.

The basic binary system has no way of representing negative numbers. To accomodate this, we use what is called a sign bit. The sign bit is simply the leftmost bit we a talking about (meaning that often we have a 32 bit piece of data, but only care about 8 or 16 of the bits - so the sign bit is the 8th or 16th bit respectively), and is set to one for negative numbers. This means that if you want an 8 bit number to be negative, then it's eighth bit must be 1 (and 16th bit must be one for 16 bit numbers, etc.).
Two's Complement is an operation (yes there is an assembly instruction to perform it) that converts a positive integer to its negative equivalent (e.g. 1 to -1, 5 to -5, etc). To perform it, simply invert every bit in the number, then add a binary 1 to it. Take the number 00000001 (the eight bit integer 1). To make this -1, invert every bit (11111110) and add

binary 1 to it -> 11111111. This then is -1 (or FF hex) as an eight bit integer. What happens if we want to treat this as a 16 bit integer? Big trouble, because now the sign bit is bit 16 and god only knows what is in bit 16. So, assembly has an instruction called Extend that extends a number out any number of binary places to make sure that any bits to the left of the original number don't affect its value.
All of this is relatively unimportant, since the assembly program you are trying to crack has already taken care of all these details and I have yet to see this type of information be critical to the cracking process. I simply wanted to get this out in the open so that you will have a better understanding of some of the instructions that will come up in the assembly instruction listings.

Now let us start by talking about memory. You probably already know that there are two kinds: ROM - Read Only Memory - and RAM - Random Access Memory. As crackers, we don't care about ROM since we can't change it. Memory is one of the two things that we can move information into and out of (the other being CPU registers explained below). Each individual piece of memory has its own address which is simply one number from a sequential list of all available memory (i.e. it starts at zero, and goes up to the end of memory). The address is the means of telling the processor which piece of memory we are talking about. For example, if we want to execute a piece of code, we need to tell the processor the address of the memory that the code starts at.

Basic Architecture and Addressing Schemes

CPU Registers

The 680X0 processors contain 8 data registers and 8 address registers. You can think of a register as a variable if you like; basically it is a storage unit that can hold up to 32 bits (binary digits) of information - or 4 bytes. Note that the programmer is not required to use all 32 bits; in fact most assembly operators can be used on 8, 16 or all 32 of the bits.

The Data registers are labeled D0 through D7 and are used to hold data that will be operated upon. For example, mathematical operators (e.g ADD, SUB[tract] ,etc.) operate on data registers.

The Address registers are labeled A0 through A7 and are used to hold memory addresses. This is how assembly language treats pointers. Pointers are simply a tool for easily dealing with a particular section of memory. If an address register contains an address, then that register can be used to move things into and out of the memory address that it contains (i.e. the memory that it points to).

It is important to remember that ANY register simply contains 32 bits of information. There is actually no difference between what is contained in a data register and what is contained in an address register. In fact, information can be moved between the two directly. The reason we call D0-D7 data registers, is because there are no commands to deal with their contents as addresses. And we call A0-A7 address registers because all the address commands apply to them.

Addressing Schemes:

The idea here is to understand some of the ways that information can be moved into and out of registers and memory itself. I will give some very short programming examples to illustrate both the syntax and the use of a given scheme. I will be using the MOVE instruction which simply moves the first argument into the second argument:

for example: MOVE 100,D1

moves the number 100 into the data register D1. You might be wondering whether 100 is binary, decimal, or hexidecimal. Well, right now we don't care, but as a general rule, we will assume that a number is decimal, unless it is

prefixed by a dollar sign $. TMON and Nosy will be very explicit about telling you what type of number the command is using - but more on that when we talk about TMON and Nosy.

BTW, this list is not the offical set of addressing schemes. I have grouped similar schemes into larger groups. For example, there is immediate addressing which means that you are moving a value (not a memory address or register). I have grouped immediate addressing with direct addressing since it does the same thing.

Direct Addressing: This is simply the moving of information directly into a register or memory address.

Examples:

```
MOVE    100,D1    ;100 is in decimal
MOVE    D1,D2
MOVE    D0,100    ;A little different here: since 100 is the receiving address (the second one) it will be treated as a memory address. So this instruction moves the contents of D0 into memory address 100.
MOVE    $55,D5    ;$ indicates 55 is in hexadecimal
MOVE    $97BA54,A1          ;moves the hex address 97BA54 into A1.
```

Remember here that the last two instructions are essentially the same. They both move some number into a register. However, the last instruction - since it moves the number into an address register - is setting up a pointer and a whole host of new instructions become available to it that are not available to the D registers.

Later we will note that there are several parameters that can be attached to the MOVE instruction (and many other instructions, for that matter). These will be covered later. This section is simply to show you how various kinds of information is manipulated. Note that in Direct Addressing, you see exactly what it is that is being moved: in the first example, you can see directly that the decimal number 100 is being moved into register D1. Any subsequent operations on D1 will involve the number 100.

Indirect Addressing: (extremely important)

This scheme involves moving some address into an address register and then operating not on the number in the address register, but rather on the address that is contained in the address register.

Example:

```
MOVE    100,(A1) ;moves the decimal number 100 into the address pointed to (or contained in) by A1.
```

Re-examine the last example of Direct Addressing. The command moved the number $97BA54 into address register A1. Since it is an address register, we can think of $97BA54 as an address rather than just a number. It may well be just a number, but odds are it will eventually be used as an address. The instruction above moves the decimal number 100 into the address $97BA54. It does not move the number 100 into address register A1. The parentheses mean that whatever is in A1 is actually an address and that this memory address will now contain the number 100.

Example:

```
MOVE    (A1),$1000
```

This instruction looks at the contents of A1, treats the contents as a memory address, and gets whatever is contained in that address and moves into hex address 1000.

Example:

```
MOVE    (A1),(A2)
```

This instruction looks at the contents of A1, grabs the contents of the address it contains, and places this value into the address pointed to by A2.

Lets look at a simple program and examine the memory that it deals with:

```
MOVE    100,D0    ;move 100 into D0
MOVE    $5000,A1  ;move address $5000 into A1
MOVE    D0,(A1)   ;move D0 into address in A1
MOVE    D0,A1     ;move D0 into register A1
```

Ok, let's analyze this sucker. First off, we move the decimal number 100 into data register D0. Any further references to D0 will also be references to the number 100. The second instruction moves the hexadecimal number 5000 into address register A1. Since we are dealing with an address register, we can think of $5000 as the memory address $5000. The third instruction says to move the contents of D0 (which is the number 100) into the address contained in A1 (which is the address $5000). So after this instruction, if you looked at memory address $5000, you would see the number 100. The last instruction serves to illustrate the difference between direct and indirect addressing. This instruction move the contents of D0 (still 100) directly into register A1 (and not into memory address $5000, as the previous instruction did). After this instruction, if you looked at the A1 register, you would see the number (or address since it is an address register) 100. After this last instruction, if you repeated the third instruction, the number 100 would be moved into memory address 100 (since we just changed the address contained in register A1).

Consider an assembly program that needs to fill a block of memory - let's say from address 100 to 200 - with the number 10. To do this with direct addressing would require the following:

```
MOVE    10,D0             ;D0 now contains the fill number.
MOVE    D0,100    ;put the number 10 into address 100.
MOVE    D0,101
MOVE    D0,102
```

and 97 more move instructions to directly move the number 10 into the appropriate memory addresses. Now consider the same program using indirect addressing (here I will use some psuedo-code to fill the loop structure):

```
MOVE    100,A0    ;put first address into A0.

While A0 not equal to 200 do the following:

        MOVE    10,(A0)

        Increment A0 to next address

End While Loop.
```

Note that this program is much simpler. Once the address register is set to the correct address, we can move the number 10 into this address then just increment the value in A0 which effectively makes A0 point to the next address.

Note also that we could have MOVEd the number 10 into D0 and then inside the loop MOVEd D0,(A0) which would have had the same result but with one more instruction.

Auto Increment Addressing:

This is not actually a distinct scheme, rather it is a slight modification of the indirect scheme. The idea is to automatically update a pointer simply by referencing it. There are two flavors of this: auto pre-decrement, and auto post-increment. Pre-decrement first decrements the register in question, while post-increment increments the register after the instruction is finished. It looks like this:

```
MOVE    D0,-(A1) ;decrement A1 to the previous address and put the contents of D0 into this new address.
MOVE    D0,(A0)+;move D0 into address pointed to by A0 and then increment A0 to point to the next address.
MOVE    (A0)+,(A1)+         ;move the contents of memory pointed to by A0 into the memory address pointed to by A1 and then increment both registers.
```

Now lets look at the previous program to fill a block of memory:

```
MOVE    100,A0
While A0 not equal to 200 do:
MOVE    10,(A0)+ ;fill the address and increment to next address.
end while loop.
```

In this program, we use the auto post-increment to automatically increment register A0 to the next address that we will be using. This type of program structure is often used to move and compare passwords around in memory. Let's say the password is residing at memory address $A000 and that we need to move it to address $B000 before we call a routine that checks to see if is the correct one. Here is a program we might use:

```
MOVE    $A000,A0        ;put source address in A0.
MOVE    $B000,A1        ;put destination into A1.
MOVE    (A0)+,(A1)+     ;move one piece of password to destination and increment both pointers.
MOVE    (A0)+,(A1)+     ;move next piece of password to destination.
```

The third line moves the first half of the information from $A000 to $B000. After both registers are incremented, the registers contain $A002 and $B002 respectively and are ready for the next piece of the password to be moved (assuming the password was 4 bytes long). Now why, you are asking, did the auto-increment add two to the two addresses instead of just one? Well, check out the next section on data size parameters to find out.

This about wraps up addressing schemes and register introduction. Next I want to look at one instruction - MOVE - and consider all the parameters one might use with it.

The first thing to consider is that there are several types of MOVE instruction. There is the basic MOVE that we have used up until now. This is used to move data around.

MOVEA is used to move addresses. Example: MOVEA $5000,A0.

> Yes - we should have been using this in the above examples when moving addresses into address registers, but I wanted to show addressing types, not instruction types. The Move Address is used just like the Move command, but lets you know that it is an address that is being moved (which means simply that the destination is an Address register).

MOVEQ Move Quick: A shortcut instruction that moves an eight bit signed integer into a data register.

Two things to note: 1) a eight bit integer translates to -128 to +127 in decimal (the 8th bit is the sign so we only get to use 7 bits as actual data), and 2) all 32 bits of the destination register are affected. This means that even though only 8 bits are used to represent the integer, these four bits will be sign extended into a 32 bit integer (remember - sign extension means that the sign of the number will be preserved as we use all 32 bits of the register). Don't get too confused here. The MOVEQ instruction simply takes an 8 bit integer and turns it into a 32 bit integer before putting it into a register. We could certainly think of the eight bit integer as unsigned (always positive) even though the instruction says that it is signed. Signing the integer becomes important only when we remember that the sign (or 8th bit) will be extended across 32 bits - so if you use MOVEQ to put the unsigned number 255 (11111111 binary) into D0, the instruction says OK, here is the signed eight bit number -1 (in binary, -1 and 255 are the same), and it needs to be turned into a 32 bit signed number. *Now* we have problems with the 255 because -1 in 32 bits is 32 binary ones, but 255 in 32 bits is still only 8 binary ones. This will make more sense when we look at data sizes.

This command is often used to load loop counters into D registers. A standard MOVE instruction could be used, but the MOVEQ is a shorter command and therefore takes up less memory and fewer machine cycles.

Example: MOVEQ $50,D1 ;treat this instruction as a normal direct address MOVE.

MOVEM Move Multiple: used to quickly move several registers to or from memory.

Example: MOVEM D4-D7/A0-A5,$5000.

Moves data registers D4,D5,D6 and D7, and address registers A0,A1,A2,A3,A4, and A5 into memory starting at $5000. This command is used primarily at the start and end of subroutines to save the contents of registers. Note that by reversing the arguments (so that $5000 comes first), the registers are restored to their original values which were saved in the above instruction.

There are a couple of other forms of the MOVE instruction, but they are rare and unimportant for cracking. If you see one, you should be able to figure out what it is doing. Now, we look at modifying the operands of the MOVE instruction.

Up until now, we have worked under the assumption that registers (and memory) contain 32 bits of information. This is not quite true. First of all, a memory address can hold 8 bits of information. Luckily, the Mac is smart enough to know that if we are moving a 32 bit register into memory, it needs to use 4 consecutive memory addresses. Secondly, we aren't limited to just 32 bit instructions. Consider:

```
MOVE.L D0,(A0)
MOVE.W D0,(A0)
MOVE.B D0,(A0)
```

These demonstrate the methods for referring to Long-words (all 32 bits), Words (16 bits) and Bytes (8 bits). The first instruction moves all 32 bits of D0 into the address pointed to by A0. Since the address in A0 can hold only 8 bits of information, the processor will put the remaining 24 bits of information into the three address following A0. The second instruction says to move the low 16 bits (I'll illustrate low bits in a second) into the address pointed to by A0 and the address following A0. The last instruction moves the low 8 bits of D0 into just the address pointed to by A0.

OK: here is what all that really means. Consider:

Instruction	Memory Address Contents->	$5000	$5001	$5002	$5003		
MOVE	$5000,A0	??	??	??	??		
MOVE	$12345678,D0	??	??	??	??		
MOVE.B	D0,(A0)	$78	??	??	??		
MOVE.W	D0,(A0)	$56	$78	??	??		
MOVE.L	D0,(A0)	$12	$34	$56	$78		

Question marks indicate that the instruction did not affect that memory address. Note that 1) when the information to be moved is longer than 8 bits it is automatically moved into successive memory addresses, and 2) the information is stored from most significant to least significant. The terms most and least significant (or high and low) are used to designate the higher vs lower portions of the number. In the number $1FF hex, the most significant byte is 01 and the least significant byte is FF. In the number $12345678, the MSB (most significant byte) is 12 and the LSB is 78. In that same number, the most significant word (2 bytes) is 1234 and the least significant word is 5678. I will often make references to both most/least significant bytes and most/least significant bits.

One last thing before we move on is to note that when using the auto increment/decrment addressing modes, the amount of increment or decrement is dependent upon the size of the data being moved (which makes sense). If you say MOVE.W D0,(A0)+ then A0 will be incremented 2 bytes so that it then points one address past the data just moved into it. Likewise, if the instruction was MOVE.L D0,(A0)+, then A0 would be incremented by 4 bytes and would again point one address past the data just moved.

Also, often the size identifier is left off the instruction (like in MOVE D0,D2). When this is the case, it means the instruction is using a word size operand or MOVE.W. If the instruction is referring to byte or long-word size operands, it will explicitly say so in the command - MOVE.B or MOVE.L.

Special Registers:

Program Counter, denoted PC. This register always points to the instruction to be executed. You won't usually care what is contained in the PC, but you will want to do your assembly listings from wherever it currently is. TMON makes it very simple to start dis-assembling from the current PC so that you can see on-screen the instructions that are going to be executed.

The Status Register: very important.

This guy is how the processor keeps track of what just happened. For example, anytime you compare two values, you need to know if they were equal, not equal, one was bigger, etc. All this type of information is contained in the Status register. Basically, the status register is a 16 bit register in which certain bits contain information that you will want to access. Don't worry about which bits mean what because assembly language has operators that refer to the bits with nice, easy to remember mnemonics. Here are the bits that you will care about:

Z the zero flag. This flag is set if the result of an operation is zero, or if two compared values are the same - it is cleared otherwise. For example, ADD.B $FF,1 would result in the number $100. But since we specified a byte size operation, the byte result is 0 and the flag would be set.

C the carry flag. This contains the carry from an arithmetic operation. If you add two 8 bit (.B) numbers, the carry flag contains the 9th bit. Say you add $FF and 1 again. The result is a byte valye of 0 with a carry into the next bit. This carry would show up in the c flag. This bit also receives bits that are shifted out of a number during shift or rotate instructions. (See commands list).

N the negative flag. Set if the high bit (meaning the 8th bit when using the .B specifier, the 16th bit for the .W, etc) of an operation gets set. Also gets set if the result of an operation is negative.

V the overflow flag. Set whenever an operation yields a result that cannot be properly represented. For example, when adding the bytes 7F and 01, the result - 80 - cannot be represented in 8 bits. In eight bits, the eighth bit is the sign bit (telling whether the number is posative or negative). Note that this only happens if you are adding bytes - if the command added words, then the result CAN be represented in 16 bits. This flag won't be used too much.

X the extended flag. This is basically a copy of the carry bit, but not all operations affect it. The X flag is used to enable multi-precision instructions, that is, instructions can be intermixed without always affecting the X flag (in this case , the multi-precision carry bit). Once again, not used to much.

This probably doesn't make too much sense. That's OK, because you will get the hang of it when we look at a batch of code listings. The only reason I am listing them here is because TMON can display these flags and their current values. This allows you to predict where the program is going when it decides to branch somewhere. These flags are used to control program flow and, as such, are the single most important element to cracking. This is how you tell a program that the password you just typed was equal (and not unequal) to the password the program is looking for. We will look at the branch instructions later on. These instructions almost all use the Status Register Flags.

The final special register is actually just the A7 address register. The reason it is special is because it is used as the stack pointer on the Mac. The Stack is basically a chunk of memory that is used for special situations such as jumping to a subroutine and having to remember where the program jumped from so it can return when the subroutine is finished. The stack is also an excellent way to pass values to a subroutine. This will be illustrated later. All you need to understand is that the Stack is a piece of memory and can be manipulated as such. To refer to the stack, refer to the A7 register. Also, the stack moves backwards as it is used. Therefore, when a program wants to put a number on the stack it uses the pre-decrement indirect addressing mode:

```
MOVE   D0,-(A7) ;puts the value in D0 onto the stack and moves the stack
                pointer back one address.
MOVE   (A7)+,D0;puts the value on the stack into D0 and increments the
                stack pointer to the next stack value.
```

And of course, to get the value back off the stack, you would use Post-Increment. These are not always used, but when they are used, it moves the stack pointer to the next available piece of stack space. When we begin working with Traps, you get a good workout with the stack so don't worry if this doesn't make complete (or any) sense yet.

Traps

Traps are a quick and easy method of accessing the 9 jillion built-in subroutines found in the Macintosh ROM. Traps do everything under the sun and are probably the main reason that all the Mac programs look alike. When a program wants do anything from drawing text to bringing up dialog boxes to putting up menus, traps are used. Why not just call the subroutines directly? Well, the problem is that every time Apple comes out with new system software, they change the addresses of one or more of these subroutines that almost all programs need. This would create chaos for applications, so Apple uses the idea of a trap table. The trap table is a means of associating the trap name (actually it's machine language code) with the proper address of the subroutine. So, no matter what the system version (within reason), an application can use the trap table to correctly call the subroutine it wants. These traps are easy to spot: they all start with an underscore and then the name of the trap, e.g. _GetNewDialog.

A quick note about traps and viruses / anti-virus programs. If you were ever wondering how a virus program works, consider that a virus needs to be able to write portions of itself onto a disk. To do this, it needs to have access to an operating system that can do the actually writing. It could either pack an operating system around with itself (unwieldy and difficult to change when apple modifies the system) or use the trap table to call the traps that write resources. Now, the trap table can be patched by a program...i.e. a programmer can substitute his own subroutine into the trap table so that any program that calls the trap to do something, actually calls the new subroutine. Knowing this, an application could be written that patches the trap table and monitors the activity of any trap that writes resources. (I haven't de-compiled the newer virus programs, but I know that's how vaccine worked). The anti-viral program then just sits back and intercepts any of these traps, takes a look to see just what it is that is being written and where. If it looks suspicious (like writing an nVIR resource to the system!) then it lets you know. WDEF was a really great virus because the programmer figured a way to bypass this method. The first thing WDEF does is try to determine exactly which system it is operating under, and, if it is one that it recognizes (the 6.0x series I believe) it will re-patch the trap table with the original system values so that it can write to the disk without being monitored! The key is that this only works if WDEF knows the original values of the trap table and, since they often change, this means that WDEF is only effective on certain system versions. (Note that if it cannot re-patch the trap table, it will attempt to run and hope that there is no anti-virus program running).

Well, back to assembly. Almost every trap needs some parameters to operate. For example, GetNewDialog needs several parameters, including the ID # of the dialog to load, and several other things; and it returns a pointer to the dialog. Here is where the stack becomes important. Most traps use the stack to pass parameters and return values. Consider the following code (which will probably be incomprehensible)

```
CLR     -(A7)       ;put 0 (word length since there is no size specifier ) on the
stack
MOVE    $2FF,-(A7)          ;Put $2FF (word size again) on the stack.
```

```
CLR.L    -(A7)      ;put a nil-pointer on stack
_StopAlert
MOVE    (A7)+,D0
```

This little subprogram brings up an alert dialog. If we were to look in Inside Mac vol 1under StopAlert we would find that it requires 2 arguments and returns one result. If we were programming, we would care what types of information these parameters are (integer, pointer, etc.) but since we are cracking, we can assume that the program to be cracked has already figured all this out.

Anytime a trap returns a value the calling code must allocate space on the stack before it puts the parameters on the stack. That is precisely what the CLR instruction does. (CLR or clear, puts zero into its operand so CLR.L D0 would put 32 zero bits in D0) This is a fast way to move the stack pointer back one byte...we don't actually care what get puts on the stack (zero in this case) because the trap is going to replace that number with its return result. Since Inside Mac says StopAlert returns an integer and that an integer is 2 bytes or 1 word, we first clear an integer's worth of space on the stack.

Next we start putting arguments on the stack in the same order as Inside Mac says. The first thing is the alertID which is an integer. This is simply the number of the alert - i.e. the number you would see if you looked at the alerts in Resedit. So, this number ($2FF in my example) is moved onto the stack. The second argument is filterproc and is a procpointer (nothing more than a pointer). This argument is used only if the built-in dialog handlers don't quite cut it for you're application (maybe you have special command keys to watch for or something). If this is the case, you would pass a pointer to you're filtering procedure in this argument. Since I don't care about this, I will pass a nil pointer (one that points to nothing - this is defined as $00000000 [a long word] in assembly).

Once I have put the proper information on the stack, I can call the trap. The final instruction moves the trap result from the stack into register D0. At this point I can test the result and branch accordingly.

Finally, let's take at the branching structures. Branching is how a program makes decisions based upon tested values. For example, you type in a password. The program must compare what you typed in with what the true password is. Once it compares the two, it has to be able to go one place if you typed in the correct value, and someplace else if you typed in the wrong password. There are several ways to compare values, and I will cover all of them in a listing of the assembly commands. The most common is the CMP (compare) command. This compares its two operands, and sets the Z flag if the two are equal and clears the Z flag if the the two are different. Don't worry if the Z-flag doesn't quite register - it was one of the bits of the status register and you won't care too much about it...just note that the various branching instructions will be testing the status flags and jumping to a new chunk of the program accordingly. How about an example?

```
MOVE.B 1,D0
MOVE.B 2,D1
CMP.B  D0,D1
BEQ    Code Section 1
BNE    Code Section 2
```

OK, first we move two numbers into D0 and D1. The CMP instructiion compares the two values (actually it subtracts the second from the first - thus you can test for more things than just the two being equal) and sets the status register accordingly. From this example, we can see the the two are not equal and so the BEQ (branch if equal) will not be executed. However the BNE (branch if not equal) will be executed since the values are indeed not equal. The branch instructions cause program execution to actually jump to a new spot in memory. From this example, you can see that what flags in the status register actually get set is not of primary concern. All you have to know is that two values are being compared, and the program wants to know if they are equal - as opposed to wanting to see which was bigger...consider:

```
MOVE.B 1,D0
MOVE.B 2,D1
CMP.B  D0,D1
BGT    Code Section 1
BLE    Code section
```

Here, the program wants to know which value is bigger. In this example, if D1 is bigger than D0, the the BGT (branch if greater than) will execute. The BLE (branch is less than or equal) will not execute. This is really easy to pick out in programs - as long as one of the various CMP instructions is used...note I say of the various; remember that most commands have several modes: consider CMP (compare), CMPA (compare address), CMPI (compare immediate), CMPM (compare memory). Once again, you don't care which of these is being used, you just care what the hell is being compared, and how they are being compared (are they equal?, is one bigger?, etc)

Let me quickly mention that BEQ is not technically branch if equal (although functionally it certainly is). BEQ means branch if equal to zero (referring to the Z bit in the status register) and BNE means branch if not equal to zero. This is not critical, but it will help you to correlate the zero bit in the status register with the BEQ and BNE instructions.

OK, now let's take this one step further. You know that a program can use the CMP instruction to test two values and you know that something happens to the status register - but you really don't care what - and you also know that you can jump to a new section of code based upon the result of the CMP. Consider for a moment the fact that the branch instructions depend entirely upon a bit in the status register. By this I mean that BEQ only executes if the Z (zero) bit is set, BCC (branch carry clear) only executes if the carry flag is clear, etc. From this it should be evident that ANY operation that changes the status register bits, could potentially be a reference for a branch. Consider the seemingly harmless enough CLR instruction. It serves to put the value zero into its operand. But, by its very definition, the CLR instruction sets the Z flag to 1 since it is setting something equal to zero. There are a slew of commands that set and clear the various bits in the status register. Refer to the command listing to see which commands affect which status flags.

There are also several ways to change which section of code is currently executing. As you have seen, the branch instructions all cause the program to jump to another piece of code. Similarly, the BRA (branch with no test of status flags), JMP (jump), BSR (branch to subroutine) and JSR (jump to subroutine) all cause the program to jump to another location and begin executing. The BSR and JSR will cause the program to execute at its new loaction until an RTS (return from subroutine) is encountered at which point the program jumps back to the instruction following the original JSR or BSR.

Finally, I want to quickly discuss two important instructions: PEA and LEA, which stand for Push effective address and Load effective address. Basically, LEA takes the first argument, computes the address at which that argument resides, and puts that address into the second argument. PEA computes the address of the argument and puts that address onto the stack. Many programs use PEA as a shortcut to putting trap arguments onto the stack. For example:

```
LEA     var1,A0   ;put the address of variable 1 into A0
MOVEA.L           A0,-(A7) ;put address on stack.

PEA     var1      ;put address of variable1 on the stack.
```

These two code listings do essentially the same thing. The first computes the address where variable 1 resides in memory and places that address in A0. At this point, we could use A0 to move information into an out of the variable var1 using indirect addressing (MOVE 1,(A0)). Then, the address in A0 is placed on the stack. The last line directly moves the address of variable onto the stack accomplishing the same thing as the previous instructions.

A word about pointers and handles. You should be familiar with pointers by now. A pointer is simply an address which is used to access the memory that it points to. A handle is nothing more than a pointer to a pointer. That is, a handle is an address that points to some piece of memory, just like a pointer. The difference is that the memory the handle points to contains the address of yet another piece of memory. Many traps return handles to data rather than pointers. The reason is so that if the Mac's memory manager needs to move memory around, pointers can be moved without loosing the handle to the pointer. This isn't too important to cracking since, once again, the program knows how to handle its pointers. You will often find a section of code that looks like this:

```
_GetNewDialog     ;this trap returns a handle (according to IM) to the dialog in
question.
MOVE.L (A7)+,A0;Move the handle from the stack into A0.
MOVE.L (A0),A0   ;A0 now contains the pointer.
```

Basically, this turns a handle into a pointer. First, the handle is moved from the stack into A0. (Remember, traps pass return values via the stack). Next, using indirect addressing, the handle is turned into a pointer. The last line first looks at the value in A0 and treats it as an address. Then it looks at the contents of this address. This 32 bit value (which is actually the pointer that the handle points to) is then moved back into A0. Lets say A0 contains memory address 1000. At memory address 1000 is the value 2000. Now, 2000 is where the data we care about is actually

located. So, we take the value in 1000 (which is 2000) and place that value back into A0. After this line, A0 contains the value (or address) 2000 and so A0 points to the data in question. I illustrate this because it is an often used technique.

Following is a detailed description of all the 68000 instructions. Some day I will buy a book on 68030/68882 instructions and update this, but it should serve for now.

COMMAND LISTING

ABCD — Add Binary Coded Decimal. Add two operands using BCD, result is in the second operand. Binary coded decimal is basically hexidecimal without the letter codes for the numbers 10-15. Using this, we get the flexibility of hexidecimal but the convience of decimal. I have yet to see this used. Flags affected:

N:	Undefined.
Z	Cleared if the result is not zero, otherwise unchanged.
C	Set by carry out of the most significant BCD digit.
X	Same as C.
V	Undefined.

ADD — Add two operands, result in the second operand. Flags affected:

N	Set if high-order bit of result was 1, otherwise cleared.
Z	Set if result was zero, cleared otherwise.
C	Set by the carry out of the most significant bit, cleared otherwise.
X	Same as C.
V	Set if operation results in an overflow (see definition of this bit).

ADDA — Add Address: add the contents of address registers, result in second operand. Flags affected: None

ADDI — Add Immediate: Add a constant to an effective address, result in second operand. Flags affected:

N	Set if high bit of result is set.
Z	Set result is zero.
C	Set on carry out of most significant bit.
X	Same as C.
V	Set on overflow.

ADDQ — Add Quick: Add a three bit value to the second argument, result in second argument. Flags affected:

N	Set if high bit of result is set.
Z	Set if result is zero.
C	Set of carry out of high bit.
X	Same as C.

V Set on overflow.

ADDX Add Extended: add two values but allowing for values that require more than 32 bits of information. Flags affected:

N Set result was negative.
Z Cleared if result is not zero. Else unchanged.
C Set on carry out of high bit.
X Same as C.
V Set on overflow.

AND — Performs bit-wise and upon the two operands with the result in the second operand. This means that the two values are compared bit by bit. For every binary digit, if both operands contain a one, the result will contain a 1, otherwise the result will contain a zero. For example, consider 101 AND 110. The result would be 100 (only the third bit is set in both numbers. Flags affected:

N Set if high bit of result is set.
Z Set if result is zero, cleared otherwise.
C Always cleared.
V Always cleared.

ANDI — And Immediate: Performs bitwise and with a constand and an operand, result in second operand. Flags affected: same as AND instruction

ASL — Arithmetic Shift Left: Performs a bitwise shift left. If there are two arguments, then the first determines how may times to shift the bits to the left. The lowest bit is set to zero.

X Set according to the last bit shifted out of the operand (that is, the most significant bit before the shift was executed.
N Set according to the most significant bit in the result.
Z Set if the result is equal to zero (all bits zeroed), cleared otherwise.
C Same as the X bit.
V Set if the most significant bit is changed at any time during the operation. (That is, if the ASL involves shifting more than one time, then if during any of the shifts, the msb is changed, V is set). NOTE - the msb does NOT mean the leftmost bit as I described way back when. It DOES mean the leftmost bit within the range of the operation. In other words, if it is a byte level shift, then the 8th bit is the msb, if the operation is at the word level, the the 16th bit is the msb, etc.

A quick note about bit operations is probably in order. Basically, any register contains 32 bits, each of which is either a one or a zero. Assembly language contains several commands for directly manipulating the individual bits in a register - as opposed to manipulating the entire value contained in the register. For example, consider the ASL above. Basically, this command moves each bit in the register in question over one slot. Now, knowing how binary numbers work, you should be able to see that this operation serves to effectively multiply the value of the entire register by 2. Similarly, the ASR (shift right) will effectively divide the value in the register by 2. There are also commands to set and clear individual bits, as well as test to see if individual bits are set. More on these commands later in the listing...

ASR

Arithmetic Shift Right: Performs a bitwise shift right. If there are two arguments, then the first determines how may times to shift the bits to the right. The most significant bit is unchanged (and not zeroed as in the ASL); this is so that the sign bit remains unchanged.

X	Set according to the last bit shifted out of the operand (that is, the lowest bit before the shift was executed.
N	Set according to the most significant bit in the result.
Z	Set if the result is equal to zero (all bits zeroed), cleared otherwise.
C	Same as the X bit.
V	Always cleared.

OK, next are the infamous branch instructions. Basically, all these operations will examine one or more of the flags and jump to a new section of code based on the result. None of these affect the status flags. Since these are the instructions that usually need to be altered to crack a program, I will list the actual hex codes associated with the instructions. This way you can go into Resedit and apply the patch. All the branches translate to 6X AA in hex where X is the status flag to check, and AA is the address to branch to. To modify the type of branch, just change the X, e.g. to change BEQ (67 hex) into BNE (66 hex) just go into Resedit, find the 67 in question, and replace it with 66. To change the address that the branch jumps to, you need to find the address you want the branch to jump to. Then start counting instructon bytes starting with the byte immediately following the branch instruction. Call that byte zero and count upwards to the spot to jump to. This number (the difference between the two addresses is the AA parameter. Note that you can start counting backwards also if you need to branch backwards. More on all of this in the actual cracking manual. Here they are:

BCC	Branch Carry Clear. Branch if the C flag is clear. 64 hex.
BCS	Branch Carry Set. Branch if the C flag is set. 65 hex.
BEQ	Branch if Equal. Branch if the Z flag is set. 67 hex.
BNE	Branch if Not-Equal. Branch if the Z flag is clear. 66 hex.
BGE	Branch if Greater Than or Equal. Branch if the N and V flags are either both set or both cleared. Basically, when dealing with these multi-flag branches (yes, there are several more coming up), look at the instruction that set the flags (usually a CMP) and ask yourself whether the relationship between the 2nd and 1st operands (the order is critical!) is true. So, for BGE, look at the CMP and say - is the 2nd operand greater than or equal to the first? If so, the branch will go. Or you can just step through this stupid command with TMON and see whether or not it branches. 6C hex.
BGT	Branch if Greater Than. Branch if 1) N and V are set and Z is clear, or 2) N, V, and Z are all clear. Basically the same as above but don't branch if the two are equal. 6E hex.
BLE	Branch if Less Than or Equal. Branch if 1) the Z bit is clear, 2) N is set and V is clear, or 3) N is clear and V is set. 6F hex.
BLT	Branch if Less Than. Branch if 1) N is set and V is clear, or 2) N is clear and V is set. 6D hex.
BHI	Branch if Higher Than. Branch if C and Z are both clear. Treat this as the same as BGT. 62 hex.

BLS — Branch if Lower or Same. Branch if either C or Z are set. Treat this as BLE. 63 hex.

BMI — Branch Minus. Branch if the N bit is set. 6B hex.

BPL — Branch Plus. Branch if the N bit is clear. 6A hex.

BVC — Branch V Clear. Branch if V is clear. 68 hex.

BVS — Branch V Set. Branch if V is set. 69 hex.

BRA — Branch. Branch regardless of what the hell is in the flags. This one is important...Imagine a program checking for an original disk, and then saying BEQ to the rest of the program. If Z is clear, the program continues and bombs. Now imagine changing that BEQ to BRA. All of a sudden, the dumb thing jumps to itself correctly no matter what happens! 60 hex.

BCHG — Bit test and Change. Inverts the nth bit (determined by the first operand) in the 2nd operand. Z is set according to the state of the bit BEFORE the inversion (by this I mean that if the bit was 0, Z is set and vice versa). No other flags are changed.

BCLR — Bit test and Clear. Same as above but clears the nth bit instead of inverting it. Flags are set the same.

BSET — Bit test and Set. Same as BCLR but sets the nth bit instead of clearing it. Flags are set the same.

BSR — Branch to Subroutine. This instruction first places the instruction following the BSR onto the stack. Next, operation is continued at the address specified by the BSR - called a subroutine. At the end of the subroutine will be a return instruction - covered later - at which point the original address is popped off the stack and execution continues from the instruction following the BSR. BSR is the same as JSR for all intents and purposes, except that BSR can first check any of the status flags the same way that the branch instructions did.

BTST — Bit Test. Test the nth bit of an operand and set the Z flag accordingly.

CLR — Clear. Sets its operand to zero.

N	Always cleared.
Z	Always set.

CMP — Compare. Compares two values. Actually, this command sets the status flags as if the second operand were subtracted from the first (but neither operand is actually changed). See the SUB command for more details.

N	Set if the result is negative. Cleared otherwise.
Z	Set if the result is zero - or if the operands are equal. Cleared otherwise.
C	Set if the result generates a borrow. Cleared otherwise.
V	Set on overflow in the subtract. Cleared otherwise.

CMPA — Compare Address. Same as above but this command will be used to compare address registers.

CMPI — Compare Immediate. Same as CMP, but this command will be used if the first operand is an actual number (instead of a register).

CMPM — Compare Memory. Once again, same as CMP, but this command always uses post-increment addressing and compares two memory addresses.

Decrement and Branch instructions

These commands make up part of assembly language's looping structures. Essentially, these commands decrement a loop counter (a specified data register) and branch back to the start of the loop. There are two ways that the loop may be terminated. First, if the condition is met, the loop will end, and second, if the loop counter reaches -1 then the loop will end. I am not going to list all the conditions for each command - for these, refer to the corresponding branch instruction.

DBRA — Decrement and Branch. No conditions are checked - terminate loop only when the loop counter reaches -1.

DBCC — Decrement and Branch unless Carry Clear.
DBCS — Decrement and Branch unless Carry Set.

DBEQ — Decrement and Branch unless Zero.

DBNE — Decrement and Branch unless Not Zero.

DBGE — Decrement and Branch unless Greater Than or Equal.

DBGT — Decrement and Branch unless Greater Than.

DBHI — Decrement and Branch unless Higher Than.

DBLE — Decrement and Branch unless Less Than or Equal.

DBLS — Decrement and Branch unless Less Than or Same.

DBLT — Decrement and Branch unless Less Than,

DBMI — Decrement and Branch unless Minus.

DBPL — Decrement and Branch unless Plus.

DBVC — Decrement and Branch unless V Clear.

DBVS — Decrement and Branch unless V Set.

DIVS — Divide Signed. Divides a 32 bit quantity (second operand) by a 16 bit quantity (first operand). The low order word of the 2nd operand gets the quotient and the upper word gets the remainder.

N	Set if quotient is negative cleared otherwise. Undefined if overflow.
Z	Set if result is zero, cleared otherwise. Undefined if overflow.
C	Always cleared.
V	Set on overflow.

DIVU — Divide Unsigned. Treat this as identical to the above instruction except that the result is treated as an unsigned integer. Flags are set the same.

EOR — Exclusive Or. Performs an exclusive or (which means that each bits are compared, if both are 1, the resultant bit is 0, if one of the two is 1, the result is 1, and if both are 0, the result is 0) on two operands. The result is in the 2nd operand. Example: EOR 1010,0011 (both binary) would yield 1001. This is not a valid instruction, but does show how exclusive or works.

N	Set if the most significant bit of the result is 1, cleared otherwise.
Z	Set if the entire result is zero (all bits zero), cleared otherwise.
C	Always cleared.
V	Always cleared.

EORI	Exclusive Or Immediate. Same as above, but the first operand will be an actual number.
EXG	Exchange. Exchanges all 32 bits of any two registers. No flags are affected.
EXT	Extend. Extends the sign bit into either a word or long word data size. N Set if result is negative, cleared otherwise. Z Set if result is zero, cleared otherwise. C Always cleared. V Always cleared.
JMP	Jump. Transfers control to another section of code. The address supplied (using any of the addressing modes) is put into the program counter and execution commences from that address. No flags are affected.
JSR	Jump Subroutine. Places the address of the next instruction on the stack, places the supplied address in the PC, and commences execution at the supplied address. When the subroutine executes a return instruction (below) the address on the stack is popped off and placed in the PC and execution commences at the address following the JSR. No flags are affected.
LEA	Load Effective Address. Computes the address of the first operand and places that address in the 2nd operand. No flags are affected.
LINK	Link. This command is a bitch to understand, but it is used a lot and is the method most compilers use to handle local variables for subroutines. Basically, Link creates what is called a Stack Frame on the Stack. Link takes two operands, an address register (A6 is almost always used), and the size of the stack frame to create. First, the address register is pushed on the stack and the resulting stack pointer is placed in the address register making it a new temporary stack pointer. Then the 2nd argument is added (note that this number is usually negative) to the original stack pointer. The memory between the stack pointer and the address register is then treated as a buffer to contain any local variables the subroutine may need. Don't worry to much about the dynamics of this command - just remember that usually, a subroutine will start with a LINK command, and end with an Unlink command and then return to the calling procedure.
LSL	Logical Shift Left. Performs a bitwise left shift on the second operand (or the first if there is only one). The first operand (if there are two) tells how many times to shift. The first bit is set to zero.

X Set according to the most significant bit before the shift is executed.
N Set a one is shifted into the most significant bit (indicating a negative result for signed numbers), cleared otherwise.
Z Set if the entire result is zero, cleared otherwise.
C Same as X.
V Always cleared.

LSR Logical Shift Right. Same as above, but shifts right. The most significant bit is set to zero (meaning the sign is lost. If this important, the program would use ASR instead).

X Set according to the first bit before the shift was executed.
N Always cleared (since zero is shifted into the sign bit).
Z Set if the result is zero, cleared otherwise.
C Same as X.
V Always cleared.

MOVE — Move. Moves the first operand into the second operand.

N	Set if the most significant bit of the result is set, cleared otherwise.
Z	Set if the result is zero, cleared otherwise.
V	Always cleared.
C	Always cleared.

MOVEA — Move Address. Same as Move except that address regesters are being used. No flags are affected.

MOVEM — Move Multiple. Moves the specified register(s) onto or out of the stack to facilitate temporary storing of the registers.

MOVEQ — Move Quick. Moves an 8 bit signed integer into a register. The 8 bit integer is sign extended to 32 bits and then all 32 bits are placed into the destination register.

N	Set if result is negative, cleared otherwise.
Z	Set if result is zero, cleared otherwise.
C	Always cleared.
V	Always cleared.

MULS — Multiply Signed. Multiplys the first argument by the second with the result in the second operand.

N	Set if the result is negative, cleared otherwise.
Z	Set if the result is zero, cleared otherwise.
C	Always cleared.
V	Always cleared.

MULU — Multiply Signed. Same as above. I am not sure as to the exact difference between the two multiply command nor the two divide commands. I wouldn't worry about it.

NBCD — Negate Binary Coded Decimal. Converts a BCD number into its corresponding negative value, much the same as the NEG instruction (below).

NEG — Negative. Performs two's complement on the supplied operand converting it to its negative counterpart.

X	Cleared if the result is zero. Set otherwise.
N	Set if the result is negative. Cleared otherwise.

	Z	Set if the result is zero. Cleared otherwise.
	V	Set on overflow. cleared otherwise.
	C	Same as X.

NEGX — Negative Extended. Same as NEG but used for multi-precision numbers.

	X	Set on borrow. Cleared otherwise.
	N	Set if result is negative. Cleared otherwise.
	Z	Cleared if result is not zero. Otherwise unchanged.
	V	Set on overflow. Cleared otherwise.
	C	Same as X.

NOP — No Operation. A two byte instruction that does nothing. This is supposedly to allow programmers room for future expansion or something, but I suspect it is to allow crackers to remove instructions without fouling up the program. No flags are affected. The hex code is 4E71 and we will definately be using this to effectively remove offensive instructions from applications - note that it is a 2 byte instruction, the same size as a branch instruction...

NOT — One's Complement. Inverts every bit in the operand.

N	Set if result is negative. Cleared otherwise.
Z	Set if zero. Cleared otherwise.
V	Always cleared.
C	Always cleared.

OR — Binary OR. Compares bits one at a time from the two operands. Result bit is one unless both bits are zero. Result bits into second operand. Example: OR 0101,1100 yields 1101.

N	Set if most significant bit of the result is set, cleared otherwise.
Z	Set if result is zero, cleared otherwise.
V	Always cleared.
C	Always cleared.

ORI — OR Immediate. Same as OR but used when the first operand is a numeric constant. Same flags set as OR.

PEA — Push Effective Address. Pushes the address of the supplied operand onto the stack using auto post-decrement. This command is often used to pass pointers (VAR variables in Inside Mac) to traps and subroutines. No flags affected.

ROL — Rotate Left. Similar to the Left Shifts, except that not only is the leftmost bit shifted into the C flag, but it is also rotated back into the first bit of the operand (instead of a zero being shifted there).

N	Set if one is rotated into the most significant bit, cleared otherwise.
Z	Set if result is zero, cleared otherwise.
C	Set according to the last bit shifted out of the operand.
V	Always cleared.

ROR

Rotate Right. Same as ROL, but shift to the right. The bit shifted out of the lowest position is placed into the C flag, and also rotated back into the most significant bit.

N	Set if one is rotated into the msb, cleared otherwise.
Z	Set if the result is zero, cleared otherwise.
C	Set according to the last bit shifted out of the operand.
V	Always cleared.

ROXL

Rotate Left with Extend. Same as ROL, but the bit that gets shifted into the C flag is also shifted into the X flag. Flags identical to ROL except that the X flag will be the same as the C flag.

ROXR

Rotate Right with Extend. Same as ROR, but the bit that gets shifted into the C flag is also shifted into the X flag. flags identical to ROR except the the X flag will be the same as the C flag.

RTS — Return from Subroutine. Places the long word from the top of the stack into the program counter and resumes execution. This has the effect of returning execution at the end of a subroutine called with either BSR or JSR. No flags are affected.

Set Instructions. This is a group of instructions that use the condition flags in an identical manner to the Branch instructions and therefore will not be listed out in full detail. Essentially, if the condition that the command is testing (for example, not equal) then the operand's low byte is set to all ones (hex FF), otherwise the byte is cleared to zero. Example:

SEQ D0 — This would set the low byte of D0 to hex FF if the Z flag was set.

There are two special forms: SF will always clear the byte, and ST will always set the byte.

SUB — Subtract. Subtracts the first operand from the second - result in the second operand.

- X — Set on borrow, cleared otherwise.
- N — Set if msb is one, cleared otherwise.
- Z — Set if result is zero, cleared otherwise.
- V — Set on overflow, cleared otherwise.
- C — Same as X.

SUBA — Subtract Address. Same as SUB, but used for address registers. No flags are affected.

SUBI — Subtract Immediate. Same as SUB, but used when the first operand is a numeric constant. Same flags set as SUB.

SUBQ — Subtract Quick. Same as SUBI except that the constant is limited to 3 bits. Same flags as SUB.

SUBX — Subtract Extended. Subtract for multi-precision numbers. Same flags set as SUB, except that Z will only be cleared by a non-zero result - it will not be set by zero.

SWAP — Swap. Swaps the words in a single operand. That is, bits 0-15 are swapped with bits 16-31.

- N — Set if bit 31 of result is set, cleared otherwise.
- Z — Set if entire result is zero, cleared otherwise.

V Always cleared.
C Always cleared.

TAS Test and Set. Tests a byte specified by the operand and sets the high order bit of that byte to 1. Apparently this is used to prevent two processors from grabbing the same resource - but I have not seen it.

N Set according to the high order bit of the specified byte before the TAS command is executed.
Z Set if the byte is zero before the TAS is executed.
V Always cleared.
C Always cleared.

TRAP — Trap. All traps will be presented in their Inside Macintosh equivalent so you should never see this command.

TRAPV — Trap on Overflow. If V is clear, do nothing. If V is set, then the flags and the program counter are pushed on the stack, and the a new program counter is loaded from absolute location 1C hex. I have seen this instruction, but have ignored. Apparently some high-level languages use this to process overflow errors.

TST — Test. Tests an operand for negative or zero values.

N	Set if the msb is set, cleared otherwise.
Z	Set if zero, cleared otherwise.
V	Always cleared.
C	Always cleared.

UNLK — Unlink. Undoes a LINK command. The specified addres register is placed in the stack pointer (restoring it) and a long word is popped off the stack and placed in the address register (restoring it). No flags are affected.

Using MacNosy

Before looking at an actual assembly program listing, we need to look at MacNosy. The version I am using is 2.95 so if you have an older version, bear with me.

What the Hell is it??

MacNosy is an incredible dissassembler. Instead of simply converting all the hex information in a program straight into assembler syntax, Nosy analyzes the program recursively, attempting to determine exactly where data is located, what types of information is being used and passed to and from procedures, etc. Once Nosy has attacked a program, it expects you to give it some hints about what you think is going on, then Nosy examines it again and so on until you like what you see. The two main types of information Nosy deals with are Code Blocks, and Data Blocks. Code blocks are what Nosy thinks can actually be executed while data is simply referred to by the code - but never actually executed. Often Nosy will be tricked into thinking that a code block is a data block. You will find out later how to show Nosy what is really going on.

Starting Out.

The first thing Nosy presents you with is an open dialog requesting the program to disassemble. All resource files will be available, but only something with executable code would make sense to decompile - such as applications, DAs, Inits, Cdevs, etc. Once the file to decompile has been selected, Nosy asks if you want to view the resources. Pressing y <Return> will list all resources and information pertaining to each. Pressing n <Return> or just <Return> will skip to the next question. Next Nosy wants to know what type of resource to decompile. Press return to decompile CODE resources (for applications, and any inits, cdevs, or DAs that use CODE resources). If CODE is not what you want, type in the resource type - INIT for inits, DRVR for DAs, and >cdev for Cdevs (the > is necessary). Finally, a dialog will come up asking how you want to decompile. Just leave all options as is, and hit return. Nosy will go through what it terms a TreeWalk which means that it will recursively analyze the program and generate it's decompiled code.

Working with Nosy.

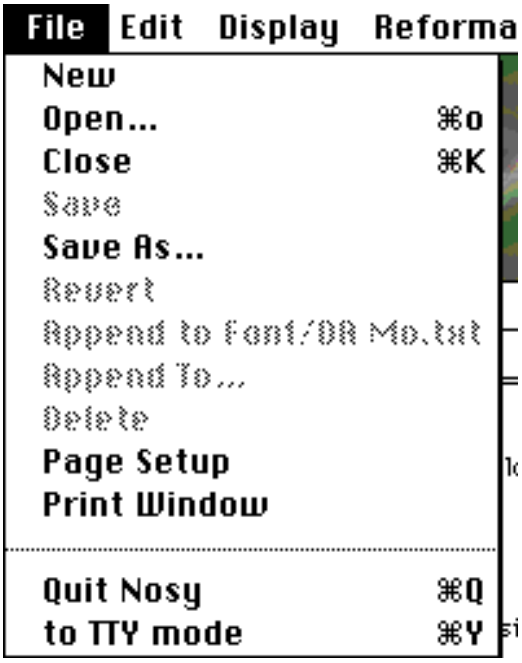
Since I don't want to re-type the entire Nosy manual, I am going to list just the basics. There are some great features that I never use and don't even know how to initiate without referring to the manual and these will be omitted. Everything I use to crack software will be covered in detail.
At this point, Nosy will present you with several windows: a Code Blks window, listing all procedures in the decompiled file; a Notes window which Nosy will use to display information; and a Mystery window, listing things that Nosy had trouble with during decompilation. Nosy can also display a list of all Data Blocks which are chunks of code that either did not make sense as executable code, or were referenced as data (Nosy looks for PEA and LEA to determine this and looks for JMP, JSR, and BSR to find individual procedures). Notes cannot be closed, so ignore it, and Mystery has things that - to date at least - don't matter that much to the cracker. When working with Nosy, you can at any time select the name of a code or data block and hit CMD-d to display it in a new window. Before examining the menu commands, lets look at a basic Nosy listing and see what Nosy tells us.

This is the procedure called Eject from the file Font/DA Mover. This is the file I will describe in detail later.

```
BD4:                                    QUAL    Eject ; b# =79  s#1  =proc47

                              vbt_1      VEQU  -64
                              param2     VEQU  8
                              param1     VEQU  10
                              funRslt    VEQU  14
BD4:                                    VEND

                              ;-refs -  2/CLOSEMYF

BD4: 4E56 FFC0       'NV..'  Eject     LINK    A6,#-$40
BD8: 41EE FFC0       200FFC0           LEA     vbt_1(A6),A0
BDC: 316E 0008 0016  2000008           MOVE    param2(A6),ioVRefNum(A0)
BE2: 216E 000A 0012  200000A           MOVE.L  param1(A6),ioNamePtr(A0)
BE8: A017            '..'              _Eject  ; (A0|IOPB:ParamBlockRec):D0\OSErr
BEA: 3D40 000E       200000E           MOVE    D0,funRslt(A6)
BEE: 4E5E            'N^'              UNLK    A6
BF0: 225F            '"_'              POP.L   A1
BF2: 5C8F            '\.'              ADDQ.L  #6,A7
BF4: 4ED1            'N.'              JMP     (A1)
```

OK, The first column contains the code resource-relative address of the instructions. To the right of this is the hex listing of the instruction, followed by an ascii display, followed by the actual assembly instruction.
The first line tells you the following: The name of the procedure (either a meaningful name Nosy found somewhere, or a generic procN where N tells where the procedure falls sequentially in the file), the block number (similar to proc number except this takes into account data blocks as well as procedure blocks), the segment or CODE resource ID #, and the actual procedure number. So in the above example, we are looking at Eject, it is the 79th block in the file (counting data blocks), it is in code resource ID 01, and it is the 47th procedure block in the file. So we could open CODE 01 in Resedit, skip down to BD8 and see the hex codes that Nosy lists. Whe BD8 and not BD4 like it says above? Well, on disk, a CODE resource has 4 header bytes (whose meaning escapes me at the moment) so we have to add 4 to the Nosy address to find the correct Resedit address.
Below this information will be listed any local variables used (they will always contain an underscore) along with their relative offsets from the procedure, then any parameters passed along with their offsets. If there is a result that will be passed back to the calling procedure, it will be listed as funRslt (as it is here). Don't worry about all the offset information as Nosy will refer to parms and local variables by their symbolic names. VEND denotes the end of the variable list. Next comes any references to this procedure - any procedures that call this procedure. Finally comes the actual listing. Occasionally, Nosy will stick more than one procedure in a window. If this happens, each procedure will have the above header. Nosy also might include data blocks in a procedure's window and it will have the word dataXX to the left of it.

Menu Commands



This is pretty straight forward and should require no explanation. Save As... will allow you to save as text a procedure or data window. This way you can type in your own comments and save them. I never use this feature, however. TTY mode allows some of Nosy's extra features. In previous version, TTY was the only mode and I imagine was hell to use. With the newer version, 2.0 or higher I believe, you can stay in the window mode that you are currently in and never use TTY mode.

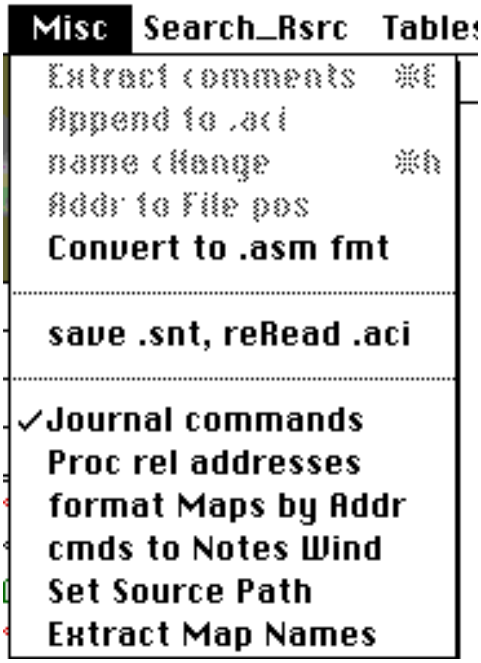
OK, the first two sections are standard. Find will find the next occurence of whatever you have selected. For example, you can select a local variable name and hit cmd-f to find the next time it occurs in the current window. If nothing is selected, Nosy presents a dialog allowing you to enter a search string.

Change	brings up a standard search/replace dialog - similar to Word.
Goto Line	allows the user to goto a specified line number in the front window.
Grab Clip & Find	operates like Find except that the clipboard is used as the search string.
Show Insert pt	scrolls the window to display the cursor (if you had scrolled the cursor off the screen).
insert pt to Top	places the cursor at the top of the screen (line 1).
sel to Notes	copies the current selection into the Notes window.

Code|Data blk	displays the currently selected code or data block in a new window. For example if you select a procedure from the code blocks window and hit cmd-d, the procedure will be displayed in a new window. If there is no current selection then Nosy will request a proc name via a dialog.
Refs to	Active only if a procedure name is selected. Displays all procedures that call the selected procedure. Using this, you can see any part of the program that is calling a particular procedure.
Call chain	Similar to Refs to. Any procedure that calls the selected procedure is also treated with Refs to. For example, you select a procedure called proc5. Doing a Refs to shows all procs that call proc5 - let us say for example, pro10 and proc 15. If you had selected proc5 and done a Call chain instead of Refs to, then first proc10 would be displayed along with any proc that called and so on backwards until the chain ends. Then proc15 would be listed along with any procs that call it, and so on until the chain ends. This is an excellent way of tracing a procedure that draws an error dialog back to the procedure that actually generated the error.
Sys syms map	Displays all system global variables along with any procedures that reference them. An example might be the system global MemErr which contains any OS errors. Nosy would display any procedures that reference MemErr - note that this command displays ALL system globals and their referencing procedures.
Trap refs map	This is a beauty. This will list all traps called by the program and the procedures that call them. If a program is asking for a key disk, use this command to search for procedures that call either ModalDialog or one of the Alert traps.
Globals map	Displays all program global variables and the procedures that use them.
Rsrc map	Lists all program resources, their lengths, and names if any.

Strings	Lists all strings and the procedures that reference them. I am not sure what Nosy defines as a string, but try it on Font/DA Mover to see.
Data Blks	Displays a window listing all data blocks.
Case jumps	Displays any procs containing a structure that resembles a case statement.
Mystery procs	Opens the Mystery Procs window showing any procedures that Nosy was unsure how to handle.
ROM Patch Info	Unknown. My outdated docs don't even mention this command.
Bad Blks	Displays information about any blocks that Nosy thought were code but contained illegal instructions so Nosy converted them to data blocks. Encrypted code would fit into this category.
Blk tbl	Displays the following for all blocks: name, segment number (resource ID #), start address, and length.
Code Blks	Displays the Code Blks window listing all code blocks.

Review...	Lets you review data blocks, optionally converting them to code blocks. More on this later.
Link Jmp to Tbl	Defines the link between a mystery jump and and a data block. To use it, select the address of a mystery JMP and choose the command. More on Jump Tables later.
Code to Data	Converts a selected code block to a data block. The blocks name won't change until the next Explore (see below).
Is Proc	Converts a selected data block to a code block. The block's name won't change until the next Explore.
JSR is JMP	Tells Nosy that a JSR is really a JMP. Sometimes a JSR is followed by data - which will look like jiberish code. The called procedure then pops the return address off the stack and uses that address as a pointer to a data block with no intention of returning to the calling procedure. To use this command, select the destination of the JSR (e.g. for JSR proc100, select proc100) and choose this command.
Explore	Initiates a TreeWalk. This allows Nosy to re-examine the program using any changes you might have made (i.e. converting data blocks to code blocks, etc).

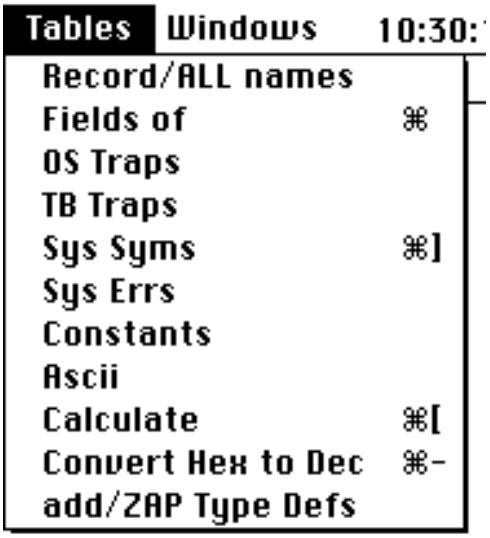


Extract Comments	Extracts comments from the selection (usually an entire procedure window) into a window entitled Comments. See section on commenting below.
Append to .aci	Appends selected comments (created using the above) to an aci (additional comment information) file. Later, this file can be re-merged into the dissassembly.
name cHange	Changes the selected name to whatever you type in. Use this to rename procedures from the generic procN to a name that tells you what the procedure basically does. The change will be immediate and global.
Addr to File pos	Converts a selected address (the leftmost information for a procedure display) into file-relative information, displaying a file-relative offset (in hex), block or sector number, and the block offset to the start of the address. This might be handy if you were using a file editor (instead of Resedit which allows you to simply open the proper code resource) to edit the file.
Convert to .asm fmt	Converts the current procedure window to asm format by removing the addresses and hex and ascii data. Cannot be undone - you must close the window, not save changes, and re-open it.
save .snt, reRead .aci	Saves a .snt file for the dissassembled program and re-reads the comment file if it exists. .snt files (saved Nosy tables) are a means of saving your dissassembly mid-session. All changes are remembered so when you re-open the file later, you begin right where you left off instead of doing a TreeWalk all over again, etc.
Journal commands	A checkmark next to this indicates that all your commands are being saved to a text file. The file will not actually be saved unless you specify so when you Quit - see quitting later.
Proc rel addresses	When checked displays the address of each instruction as procedure relative (starting at zero for each procedure instead of starting at zero for each code resource).
format Maps by Addr	When checked, Nosy will change the way it displays its various maps, i.e. Sys syms, Trap refs, etc. under the Display menu. Instead of proc names, Nosy will display the segment number, and the segment offset. But - if Proc rel addresses is also checked, then Nosy will replace the

proc names with a proc name followed by a + followed by the procedure relative offset. Try it out if this doesn't make sense.

cmds to Notes Wind	Not yet implementated - like a lot of great features (see below).
Set Source Path	Not in my manual - you're on your own.
Extract Map Names	Not in my manual - on you're own again. This doesn't seem to do anything, though, when I try it.

Unfortunately, the Search_Rsrc menu is totally disabled. Maybe the next version...



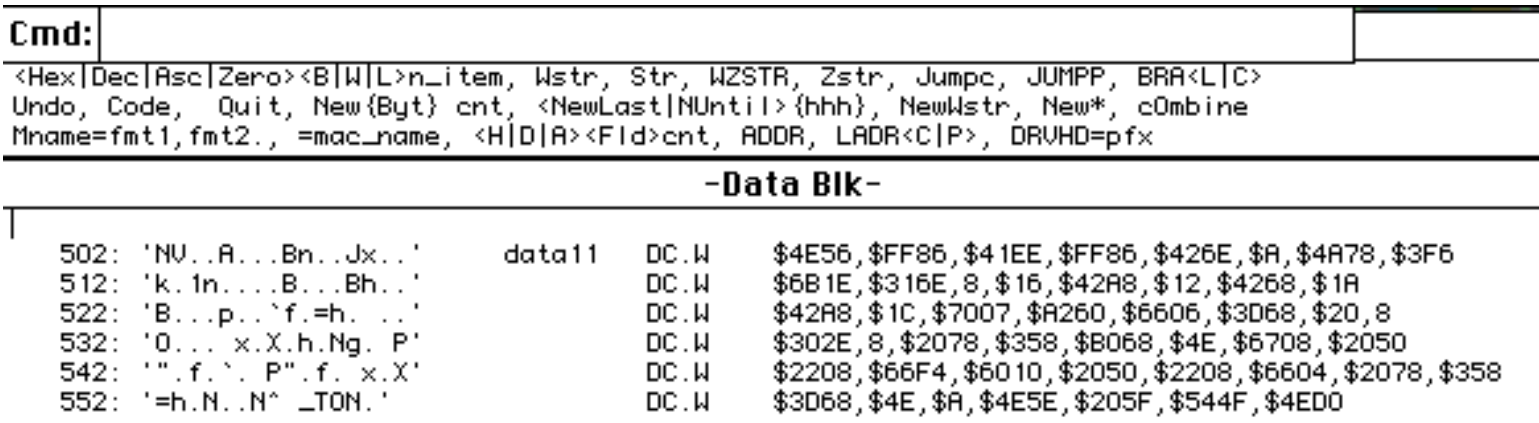
Record/ALL names	Lists all Macintosh data structures that Nosy currently knows.
Fields of	Depends on the selection: 1) a datatype (e.g. Dialog Record - or anything listed by the previous command) - describes all fields for the datatype. 2) datatype@address - displays the current values for the datatype if one exists at the specified address. 3) @address - displays hex/ascii dump at specified address.
OS Traps	Lists all known Operating System Traps and their parameters.
TB Traps	Lists all known ToolBox traps and their parameters.
Sys Syms	Lists all known System Symbols.
Sys Errs	Lists all known System Errors and their codes.
Constants	Lists all known Macintosh Constants. Note that my copy of Nosy contains an error in this window - it specifies that constants in brackets can be selected and viewed by pressing cmd-?. Use cmd-<space> (or select Fields of from the Tables menu).
Ascii	Decimal/Hex/Ascii lookup table.
Calculate	Evaluates selected expressions which can contain mathematical operators, system globals, program globals, IM datatypes, registers etc. Use # to force decimal (#10) and $ for hex ($10).
Convert Hex to Dec	Converts a selected number to decimal, ascii, and system symbol equivalent if there is one.
add/ZAP Type /Defs	No idea.

Note on commands requiring a selection (Calculate, etc.) You may have noticed that often there is no place to enter the text you want to select. In cases like these, type your text into the Notes window, select it, and choose the command you wish.

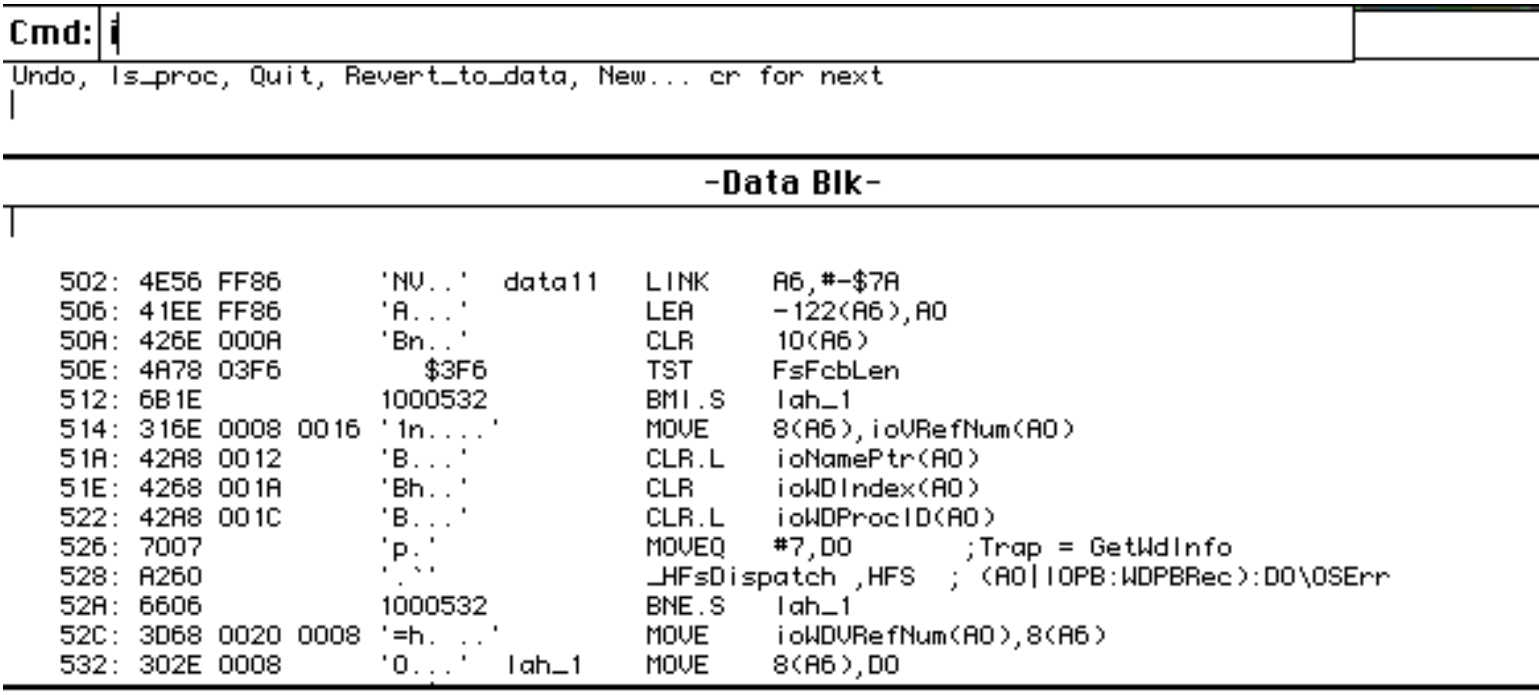
Reviewing Data Blocks

This is the process by which you tell Nosy that it has mistakenly made a piece of code a data block. Once you initiate the Review... command, Nosy will show you each data block in sequence and give you a chance to work on it. Note that you may never need to do this to crack a program - god knows I never use it unless I am really having problems. This section is to provide you with some idea of what Nosy can do.

Here is a typical display after selecting Review... from the Reformat menu. The Data Blk window displays the data block, the window directly underneath this will display the section of code that references that data block (if their is one) and the Cmd window awaits your input. There are about a zillion things you can do, but the most important one is the c command.



If your press c and return, Nosy shows you what the data block looks like in assembly:



.

Notice that this looks like pretty good code! Also note that Nosy has placed i in the Cmd window anticipating that you will want to change this to code. Here are all the commands available:

H	takes 2 parameters: 1) either L,W, or B for Longword, Word, or Byte and 2) the number of entries per line. Formats the block as Hex bytes. Example: HL2 would format the block as hex longwords, 2 per line.
D	same as H (above) but formats block as decimal entries.
A	same as H (above) but formats the block as ascii entries.
Z	same as H (above) but formats the block as zero entries.
W	Formats the block as a word-aligned Pascal string.
S	Formats the block as a Pascal string.
WZSTR	Formats the block as a word-aligned zero-terminated string.
Z	Formats the block as a zero-terminated string.
J	Formats the block as a set of Jump Table entries - each word is taken as an offset from the beginning of the data block to a common procedure and these jumped to spots are marked as common blocks (a common block - denoted com_nn - is any procedure executed via JMP instead of JSR or BSR). If you do this, you need to use the Link Jmp to Tbl command to link the jump table to its jump command. See Jump Tables below.
JUMPP	Same as J except that the entry points are marked as procedures instead of common blocks.
BRAL	Formats the block as code. Any instructions that are BRAnched to are marked with local markers (as in a standard Nosy listing).

BRAC	Same as above except that instructions reached via BRA are marked as common blocks.
U	Undoes any formatting changes.
C	Changes the block to a code listing and brings up the code menu - discussed below.
Q	Exits Review mode.
N	takes an integer parameter X. Splits the block into two blocks, the first block getting X words (remember a word is two bytes).
NB	same as above except the parameter specifies bytes instead of words.
NL	same as above except the parameter is a segment-relative address specifying the end of the the first block.
NU	takes an optional search string as parameter. Splits the block with the first block ending upon finding the search string. If no string is supplied, Nosy searches for a logical procedure end (RTS, JMP(AX)). The block is formatted as code and the code menu is displayed.
NW	Splits the block in two, the first block being made a word-aligned Pascal string.
N*	Splits the block in two. Uses the first longword to determine the length of the first block.
O	If the previous block is a data block, then combine it with the current one.
ADDR	Formats the block as a list of word-length procedure block addresses.
LADRC	Formats the block as a list of longword-length common block addresses.
LADRP	Formats the block as a list of longword-length procedure block addresses.
<Return>	Saves changes, and takes you to the next block.

The Code Menu Commands: these come up if you use c or nu to change the block to a code listing.

U	Undo any changes to size or format and takes you back to the Review menu.
I	Tells Nosy to keep the block as code and return to the Review menu.
Q	Exits Review mode.
R	Changes block back to data, but retains any size changes you may have made.
N	Same as the N commands above.
<Return>	Same as above - returns to the Review menu.

Once you have finished Reviewing data blocks, you must select Explore from the Reformat menu to have Nosy incorporate any changes into its lists.

Working with Jump Tables

A jump table is a means of efficiently transferring control to a procedure. An example of a jump table would be a program that receives an event (as most mac programs do) and then has to execute a procedure depending on what the event was. Font/DA Mover has an extermely simple jump table - actually it is not a true jump table - in which the button the user clicks is returned to its main event loop as an integer. The program then repeatedly subtracts one from the integer and branches to an appropriate procedure when the integer has been reduced to zero. A more common (and true) jump table consists of a list of offsets. The program then takes an integer which tells it which entry in the table to use, multiplies it by 2 (assuming each entry is two bytes in length) and then indirectly jumps to the correct procedure. Here is an example taken from the Nosy manual (this is from the System File's .MPP driver):

```
                LEA         data4,A3
                ADDA        D3,A3
                ADDA        D3,A3
                MOVEA       (A3),A3
                PEA         .MPP
                ADDA.L      (A7)+,A3
                JMP         (A3)

data4           DC.W        $82,$280,$26C,$3C, etc
```

As gross as this looks, lets see what it is doing. At the start, D3 contains the selector that determines which entry in the table to use. A3 is loaded with the address of the jump table. D3 is added to it twice (we could have doubled D3, then added it) so that now A3 contains the address of the proper jump table entry. The instruction MOVEA (A3),A3 grabs the jump table entry (which is simply an offset from the start of the program to the correct procedure) and puts that entry back into A3. Next the address of the program start (.MPP) is pushed on the stack, and this value is added to A3 to produce the actual address of the procedure (the address is the start of the program plus the offset). Now A3 is setup, so the program Jumps to the address in A3. If you don't mind looking at this type of listing (and I don't since it probably is not the copy protection - although it might be jumping to the copy protection) then you need go no farther. But Nosy can set this up to look much nicer.

To fix this up, select the address (the far left column) of the Jump instruction - in this case, the JMP (A3). Now choose Link Jmp to Tbl from the Reformat menu and a dialog box appears requesting the name of the jump table's block - in our case that would be data4. Click continue and a new dialog appears. The first thing we need is the table format. There are three choices: JUMPP - tells nosy to label the jumped to procedures as procedure blocks; JUMPC - tells Nosy to label the jumped to procedures as common blocs; JUMPL - tells Nosy to label the jumped to procedures with local labels in the same block as the jump table. To figure out which one to use (and it is really a matter of preference), decide if you want to break the whole thing up into many procedures, or keep it as one large procedure with tons of local labels. If the procedure is a massive one, you may want to break it up (and I would recommend JUMPP - but then, I like proc labels better than com labels), otherwise, use JUMPL.

Next we need the number of jumps. Just count the number of entries - but be careful: you need to decide the size of each entry in the table. Note the DC.W next to data4. This means that Nosy is showing you individual words so you can just count the number of entries. But if Nosy is using DC.B, then it is showing you bytes, so you would have half as many word length entries.

Finally, we need the Table Bias. Bias is a parameter that Nosy uses to determine the actual procedure address of a non-standard jump table. To calculate this, use this formula: Bias = Address of JumpTable + Offset - TargetAddress. The tricky thing is to deterine the TargetAddress. In the above example, it is easy, since the code clearly refers to the start of itself (it refers to the address of .MPP). JumpTable is the address (leftmost column in the listing) of the start of the jump table, and Offset is the first word-length offset in the table. Note that your calculations will result in a hex bias - Nosy needs you to change it to decimal.

Click Accept, do another Explore, and that is it! Now the listing looks like:

```
                JMP         (A3)

                JBIAS       92
data4           JUMP        procA
                JUMP        procB
                JUMP        procC
                etc.
```

Notice that Nosy uses JUMP to distinguish it from the instruction JMP. Once this is set up, it is a cinch to see where the jump table is jumping - provided you can deduce the selector. Most of the time Nosy works wonders with jump tables, and the few times it has problems (it will list these problems in the Mystery window) I have found it not worth the work to convert them to the above format.

Commenting Your Listings

There are a couple of cool features that I am not going to explain regarding commenting simply because I have never used them and the manual I have isn't the most verbose. Basically, you can put comments on any line that Nosy hasn't already commented. All comments must start with a semi-colon. Once you have all the comments you want, do a cmd-a to select all, and choose Extract Comments from the Misc menu. Nosy will extract all your comments into a comments window. Now hit cmd-a again, and choose Append to .aci from the Misc menu. This will save your comments. Now close the procedure window and don't save changes. Select Save .snt, reRead .aci from the Misc menu. Nosy may ask you if you want to delete something or other which it claims saves space in the Debugger. Since we are not using the Debugger, choose No. Now when you open the procedure again, you comments appear.
There is one feature I will attempt to explain, because it could be a serious boon. There a several slash (/) commands Nosy understands. One in particular, /w, works like this: Anytime a register is setup to contain a pointer to a Mac structure, you can have Nosy automatically show the structure whenever the register is referenced. Here is an example:

```
PEA         data24              ; len = 206
_OpenPort  ;  (port:GrafPtr)
LEA         data24,A2    ; len = 206
MOVE        #4,68(A2)
MOVE        #9,74(A2)
```

Note in the 3rd line that A2 is given the GrafPtr. Since GrafPtr is a pointer to a valid Mac structure (GrafPort), we could use the /w command as follows: click at the end of the 3rd line and hit return (so we are not commenting on a line that Nosy has already commented). Enter /w<space>GrafPort. Now save the comments as illustrated above. When the proc is re-opened it shows the following:

```
PEA         data24              ; len = 206
_OpenPort  ;  (port:GrafPtr)
LEA         data24,A2
/w GrafPort
```

```
                    MOVE        #4,txFont(A2)
                    MOVE        #9,txSize(A2)
```

Using this technique, Nosy will use the fields of the structure instead of the actual offsets from the register. This will work for any valid Mac structure. I haven't used this feature (I just noticed it when compiling this manual) so that is all I will say - feel free to experiment.

Well, I guess the next thing to do is start looking at some serius code listings pulled directly from Nosy (and not stuff I made up on the fly). Nosy has shorthand notations for certain operations, most commonly for stack operations. The two to watch out for are PUSH and POP (btw, TMON does not use this notation, but rather uses the standard notation found in any assembly book). PUSH is the equivalent of putting the operand onto the stack using auto pre-decrement. POP is the same as grabbing the operand off the stack using auto post-increment.

Let's take a look at some code listings from Font/DA Mover 3.8. I selected this program so that you can pull up the same listings that I will refer to in Nosy. First take a look at the initial procedure: DA Mover. There will always be a procedure whose name is the same as the program you are de-compiling. This procedure (DA Mover in this case) is the first procedure the program executes when launched.

```
430:                                QUAL    DA Mover ; b# =12  s#1  =proc4
```

OK, I will stick my notes right in the listing (below or to the right of what I am refering to), so bear with me. Note the first line (above). We are looking at block 12, segment (or CODE resource #) 1, and it is the 4th procedure in the program.
The first few lines will do some startup stuff that I do not fully understand and so I will skip it.

```
430: 4EBA 0C48      100107A DA Mover JSR     proc70        Proc 70 is just an RTS
434: 4E56 0000      'NV..'           LINK    A6,#0
438: 2C5F           ',_'             POP.L   A6
43A: 4EBA 0C40      100107C          JSR     proc71        Proc 71 calls _RTINIT which seems to be a common
                                                           step for MPW compiled programs. It then initializes
                                                           some global variables. Let's skip all this.
43E: 486D 024A      3000000          PEA     proc232(A5)
442: A9F1             '..'              _UnLoadSeg ; (Proc:ProcPtr)   Here we are dumping an un-needed
                                                           procedure.
```

Now, looking at the listing from here, notice the procedures that get called - two without labels, then SetUp, MakeAWin and FinderSE and finally MainEven which sounds suspicially like an event loop. Let's check out these procedures.

```
444: 4EBA 05E2      1000A28          JSR     proc28        If you look at this procedure, you see several
                                                           references to memory stuff like ApplHeap, ApplZone,
                                                           Rom85, etc. Looks like this is checking for enough
                                                           memory or something. Not too interesting.
448: 4EBA 021E      1000668          JSR     proc10        This proc looks to be changing a few traps. Note that
                                                           it allocates a new pointer (NewPtr trap) and then calls
                                                           GetTrapAddress, and then SetTrapAddress.
```

```
 44C: 4EAD 01F2      2005092          JSR     SETUP(A5)      Take a look at this listing below the current one:
 450: 4EAD 01FA      2005852          JSR     MAKEAWIN(A5)   Draw the main dialog.
 454: 4EBA FCC2      1000118          JSR     FINDERSE       Checks if user launched a suitcase and if so, opens it.
 458: 4AAD F4F0        -$B10          TST.L   glob26(A5)     Verify the main memory handle.
 45C: 6706           1000464          BEQ.S   lae_1          Branch if it is empty.
 45E: 4EBA FBA0      1000000          JSR     MAINEVEN       Do the actual program until user Quits.
 462: 6008           100046C          BRA.S   lae_2          Exit without error.
 464: 3F3C 0031      '?<.1'  lae_1    PUSH    #49            Out of Memory Error.
 468: 4EAD 01CA      200023C          JSR     DOALERT(A5)    Do an Out of Memory alert
 46C: 4EBA FF86      10003F4 lae_2    JSR     DOCLEANU       From here, the program exits.
 470: 4EAD 01D2      20002B2          JSR     MYEXITTO(A5)
 474: 4EBA 0C2A      10010A0          JSR     %INITHEA
 478: 4EBA 0C2C      10010A6          JSR     proc73
 47C: 4E75           'Nu'             RTS

 47E: 4E5E 4E75 C64F 4E54    data9    DNAME   FONT_DA_,4

 48A: '..'                   data10   DC.W    0
```

Here is the SetUp Procedure:

```
 5092:                                QUAL    SETUP ; b# =467  s#2  =proc199

                              vho_1    VEQU  -12              Only one local variable, no parameters.
 5092:                                VEND

                              ;-refs -  1/DA Mover            Only called via DA Mover.

 5092: 4E56 FEE6      'NV..'  SETUP   LINK    A6,#-$11A      Setup a Stack frame for local variables
 5096: 48E7 0118      'H...'          MOVEM.L D7/A3-A4,-(A7)       Save D7,A3 and A4 on stack
 509A: 7E01           '~.'            MOVEQ   #1,D7          Loop coming up, this inits the loop counter.
 509C: 6006           20050A4         BRA.S   lho_2          And branch into the loop.

 509E: 4EAD 00D2      1000AB0 lho_1   JSR     MoreMasters(A5)
 50A2: 5247           'RG'            ADDQ    #1,D7          Increment loop counter.
 50A4: 700F           'p.'    lho_2   MOVEQ   #15,D0
 50A6: B047           '.G'            CMP.W   D7,D0          Test if loop done...
 50A8: 6CF4           200509E         BGE     lho_1          If not, branch back.
```

OK, take a look at the above code. First, D7 is initialized to 1 and then the program branches down to lho_2. The loop test is setup here (15 is the end of the loop). At the compare, ask yourself, is D0 greater than or equal to D7? Well, the first time, D0 is 15 and D7 is 1 so the loop branch will execute. So, MoreMasters is called, 1 is added to the loop counter, and then the loop is checked again. This will loop 15 times (until D7 has 16 in it). MoreMasters is a trap (in this case, the procedure called MoreMasters will execute the trap) that causes a block of master pointers to be allocated in the current heap zone. See Inside Mac's (here on referred to as IM) Memory Manager section for a better description.

```
 50AA: 486D F420        -$BE0          PEA     glob3(A5)      Push the address of glob3 on the stack.
```

```
50AE: A86E           '.n'          _InitGraf ; (globalPtr:Ptr)
50B0: A8FE           '..'          _InitFonts
50B2: A912           '..'          _InitWindows
50B4: 2F3C 0000 FFFF '/<....'      PUSH.L  #$FFFF
50BA: 201F           ' .'          POP.L   D0
50BC: A032           '.2'          _FlushEvents ; (whichMask,stopMask:EventMask)
50BE: A9CC           '..'          _TeInit
50C0: 42A7           'B.'          CLR.L   -(A7)
50C2: A97B           '.{'          _InitDialogs ; (resumeProc:ProcPtr)
50C4: A930           '.0'          _InitMenus
50C6: 486E FFF4      200FFF4       PEA     vho_1(A6)
50CA: A910           '..'          _GetWMgrPort ; (VAR wPort:GrafPtr)
50CC: 2F2E FFF4      200FFF4       PUSH.L  vho_1(A6)
50D0: A873           '.s'          _SetPort ; (port:GrafPtr)
50D2: 206E FFF4      200FFF4       MOVEA.L vho_1(A6),A0
50D6: 4868 0008      'Hh..'        PEA     8(A0)
50DA: A87B           '.{'          _ClipRect ; (r:Rect)
50DC: 2F3C 000E 000C '/<....'      PUSH.L  #$E000C
50E2: A893           '..'          _MoveTo ; (h,v:INTEGER)
50E4: 3F3C 0029      '?<.)'        PUSH    #41
50E8: 4EBA CE84      2001F6E       JSR     DRAWRESS
50EC: 3F3C 000C      '?<..'        PUSH    #12
50F0: A88A           '..'          _TextSize ; (size:INTEGER)
50F2: 422D F4EF        -$B11       CLR.B   glob25(A5)
50F6: 42A7           'B.'          CLR.L   -(A7)
50F8: 3F3C 0004      '?<..'        PUSH    #4
50FC: A9B9           '..'          _GetCursor ; (cursorID:INTEGER):CursHandle
```

InitGraf, we see in IM, that InitGraf must be called once near the start of a program. It requires one parameter, a pointer to the first QD global variable. This parameter is first pushed on the stack in the previous instruction.

These traps can all be found in IM.

Note that InitDialogs needs a ProcPtr (a long word). The clr command here uses auto pre-decrement to push a NIL pointer onto the stack.

OK, notice the VAR in the trap below. This means that info will be returned via the parameter we push on the stack. So, after the trap, vho_1 will be a GrafPtr (whereas before the trap, god knows what is in it).

Now, our GrafPtr is used to set the current Port.

Now A0 contains our GrafPtr.

This instruction says to add 8 to A0,and push that address on the stack.

The MoveTo trap requires two integers to be passes, but only one value is being pushed on the stack. Since the instruction says to push long, 4 bytes are being put on the stack, and an integer is only two bytes. Even though one instruction is being used, there are actually two parameters being passed to the MoveTo trap.

It turns out that DRAWRESS will draw the 41st string in the STR# resource. If you look in Resedit, you will see that this is "3.8" the version number.

Note the lack of a size specifier. Remember that this means use the word (two bytes) size. Textsize needs an integer and IM tells us that an integer is two bytes - or one word.

This is pretty easy - sets the fontsize to 12 point.

Here is the .B size specifier, meaning clear only the low byte of glob25.

OK, this is a slightly different trap, since it returns something on the stack - as evidenced by the colon and description at the end of the trap parameter list (:CursHandle). Since this trap returns a value on the stack (and not with a passed pointer as with the GWMgrPort above), the program will first clear enough stack space to hold that value. Thus the CLR.L -(A7). The trap returns a handle which is 32 bits or a long word. The trap needs an integer, so the program pushes the word 4 onto the stack. Next, the program will pop the CursHandle returned by the trap off the stack into the variable glob24.

```
50FE: 2B5F F4EA        -$B16          POP.L   glob24(A5)    This the CursorHandle.
5102: 1F3C 0002      '.<..'           PUSH.B  #2
5106: 4EBA AEF8      2000000          JSR     SETTHECU      This subroutine is setting the cursor. If you look at it,
                                                             you will see that it looks at the parameter passed (2 in
                                                             this case) as well as glob25 (0 in this case). When
                                                             called from here, it will pass down to the 2nd
                                                             SetCursor and use the CursorHandle in glob24.
510A: 42A7           'B.'           CLR.L   -(A7)         Once again, clear space on the stack for a returned
                                                             handle.
510C: 2F3A 0144      2005252        PUSH.L  data260     ; 'PACK'
5110: 3F3C 0003      '?<..'         PUSH    #3            GetResource needs the resource type and the ID# to
                                                             load.
5114: A9A0           '..'           _GetResource ; (theType:ResType; ID:INTEGER):Handle
5116: 285F           '(_'           POP.L   A4            Pop the handle (to the PACK resource) into A4.
5118: 2F0C           '/.'           PUSH.L  A4            And push it back on the stack so HNoPurge can use
                                                             it.
511A: 4EAD 00CA      1000AA6        JSR     HNoPurge(A5)  Once again we see a subroutine with the same name
                                                             as a trap. You can bet that the trap will be called
                                                             somewhere in the subroutine.
511E: 42A7           'B.'           CLR.L   -(A7)
5120: 2F3A 0130      2005252        PUSH.L  data260     ; 'PACK'
5124: 3F3C 0006      '?<..'         PUSH    #6
5128: A9A0           '..'           _GetResource ; (theType:ResType; ID:INTEGER):Handle
512A: 285F           '(_'           POP.L   A4
512C: 2F0C           '/.'           PUSH.L  A4
512E: 4EAD 00CA      1000AA6        JSR     HNoPurge(A5)
```

OK, the previous several lines have basically loaded two resources, PACK #3, and PACK #6. The handles to the two resources have been made non-purgeable meaning that the memory manager will not remove them to create free space.

```
5132: 42A7           'B.'           CLR.L   -(A7)
5134: 3F3C 0001      '?<..'         PUSH    #1
5138: 4EAD 0182      1000D8C        JSR     proc61(A5)    This little gem invokes Pack6. My understanding of
                                                             the package manager is less than it should be, but it
                                                             looks to me like this says do a Pack6 with a selector
                                                             of 1. Hell, lets just look at proc 61...

        D8C: 7406          't.'    proc61  MOVEQ   #6,D2   OK, here is the selector (and not the 1 passed from
                                                             the above procedure). So we are going to be calling
                                                             the IUGetIntl procedure (I think) with a
```

parameter of 1 (passed from the calling procedure. Look in IM for details of this trap and its parameters.

```
  D8E: 205F          ' _'          POP.L   A0
  D90: 3F02          '?.'          PUSH    D2
  D92: 2F08          '/.'          PUSH.L  A0
  D94: ADED          '..'          _Pack6  AutoPop; (selector:INTEGER)
```

This pops the parameter passed, so that the selector parameter can be put ahead of it on the stack. Now the 2nd parm can be put back on the stack and the trap called.

```
513C: 285F          '(_'          POP.L   A4
513E: 2F0C          '/.'          PUSH.L  A4
5140: 4EAD 00CA     1000AA6       JSR     HNoPurge(A5)
5144: 42A7          'B.'          CLR.L   -(A7)
5146: 2F3A 010A     2005252       PUSH.L  data260     ; 'PACK'
514A: 3F3C 0007     '?<..'        PUSH    #7
514E: A9A0          '..'          _GetResource ; (theType:ResType; ID:INTEGER):Handle
5150: 285F          '(_'          POP.L   A4
5152: 2F0C          '/.'          PUSH.L  A4
5154: 4EAD 00CA     1000AA6       JSR     HNoPurge(A5)
5158: 4EAD 0172     1000D7C       JSR     proc59(A5)
515C: 42A7          'B.'          CLR.L   -(A7)
515E: 2F3A 00EE     200524E       PUSH.L  data259     ; 'ICON'
5162: 4267          'Bg'          CLR     -(A7)
5164: A9A0          '..'          _GetResource ; (theType:ResType; ID:INTEGER):Handle
5166: 285F          '(_'          POP.L   A4
5168: 42A7          'B.'          CLR.L   -(A7)
516A: 2F3A 00E2     200524E       PUSH.L  data259     ; 'ICON'
516E: 3F3C 0001     '?<..'        PUSH    #1
5172: A9A0          '..'          _GetResource ; (theType:ResType; ID:INTEGER):Handle
5174: 285F          '(_'          POP.L   A4
5176: 4267          'Bg'          CLR     -(A7)
5178: A994          '..'          _CurResFile ; :RefNum
517A: 3B5F FFE0        -$20       POP     glob58(A5)
```

proc 61 is returning a handle to the intl resource that it loaded, so save it in A4.

This proc calles Pack2 with a selector of 2. This reads the Disk Initialization package into memory.

Clear space on stack for a returned handle.

Push the integer 0.

A4 has a handle to Icon resource ID 0.

A4 has a handle to Icon resource ID 1.

Make space for the returned RefNum.

Note - no parameters passed.

Pop off the returned RefNum.

```
517E: 486D FEDE        -$122          PEA     glob56(A5)
5182: 3F3C 000D      '?<..'           PUSH    #13
5186: 4EAD 002A      100048C          JSR     proc5(A5)
```

Here is proc5 again - the string getter. If you remember (from looking at DRAWRESS), the 1st parm is the string ptr, and the 2nd is the string # to get. This is returning a ptr to the string "The quick brown fox..." in glob56.

```
518A: 7000           'p.'             MOVEQ   #0,D0
518C: 2B40 FED4        -$12C          MOVE.L  D0,glob52(A5)
5190: 7000           'p.'             MOVEQ   #0,D0
5192: 2B40 FECC        -$134          MOVE.L  D0,glob50(A5)
5196: 7000           'p.'             MOVEQ   #0,D0
5198: 2B40 F61E        -$9E2          MOVE.L  D0,glob41(A5)
519C: 3B7C FFFF F616   -$9EA          MOVE    #$FFFF,glob38(A5)
51A2: 426D F614        -$9EC          CLR     glob37(A5)
51A6: 7000           'p.'             MOVEQ   #0,D0
51A8: 2B40 F610        -$9F0          MOVE.L  D0,glob36(A5)
51AC: 7000           'p.'             MOVEQ   #0,D0
51AE: 2B40 F61A        -$9E6          MOVE.L  D0,glob40(A5)
51B2: 7034           'p4'             MOVEQ   #52,D0
51B4: 2B40 F5FE        -$A02          MOVE.L  D0,glob31(A5)
```

The above instructions have simply initialized several global variables. We don't care what they mean at this point. If you like, you can write down what has been set to what, but I would only recommend this if later on you need to know explicitly what a global contains.

```
51B8: 42A7           'B.'             CLR.L   -(A7)
51BA: 7002           'p.'             MOVEQ   #2,D0
```

Note the MoveQ. Remember, this is the same as MOVE.L (except it executes faster).

```
51BC: 2F00           '/.'             PUSH.L  D0
51BE: 4EAD 009A      1000A5C          JSR     NewHandle(A5)
```

NewHandle is a trap that returns a handle to a block of memory whose size is in D0. It makes sense to guess that this procedure will do essentially the same thing - and after checking, it certainly does.

```
51C2: 2B5F F622        -$9DE          POP.L   glob42(A5)
51C6: 426D F626        -$9DA          CLR     glob43(A5)
```

So glob42 has a handle to a 2 byte chunk of memory.

```
51CA: 70FF           'p.'             MOVEQ   #-1,D0
```

Here is one of those cases where the sign bit is important. Remember that the -1 is sign extended to 32 bits so D0 is being set to all binary ones (-1 in binary).

```
51CC: 2B40 F602        -$9FE          MOVE.L  D0,glob32(A5)
51D0: 42A7           'B.'             CLR.L   -(A7)
51D2: 2EB8 02F0         $2F0          MOVE.L  DoubleTime,(A7)
51D6: 7002           'p.'             MOVEQ   #2,D0
51D8: 2F00           '/.'             PUSH.L  D0
51DA: 4EAD 01A2      1001120          JSR     proc76(A5)
```

This is a gross looking (i.e. no Traps anywhere) procedure so I am not going to attempt to figure it out. You will want to use the technique a lot (the "Too Gross" technique) to determine which procedures to spend time with.

```
51DE: 2B5F F5F6        -$A0A          POP.L   glob29(A5)
```

```
51E2: 207C 0000 0AD8    $AD8          MOVEA.L #SysResName,A0
```
Put a pointer to the System File's name in A0.
```
51E8: 43ED F4F6         -$B0A         LEA     glob28(A5),A1
```
Put the address of glob28 in A1.
```
51EC: 703F           'p?'             MOVEQ   #63,D0
```
Set up D0 as a loop counter.
```
51EE: 22D8           '".'    lho_3    MOVE.L  (A0)+,(A1)+
```
This moves 4 bytes from A0 to A1. Note the use of auto post increment to automatically move the pointers to the next available data each time. This moves 4 bytes of the System name into glob28. Note that glob28 will not be a pointer to the Sys Name, but will rather contain the actual string data.
```
51F0: 51C8 FFFC      20051EE          DBRA    D0,lho_3
```
This decrements D0 (the loop counter) and branches back to the start of the loop until it is finished.
```
51F4: 422D F4F5         -$B0B         CLR.B   glob27(A5)
51F8: 267C 0000 028E    $28E          MOVEA.L #Rom85,A3
```
ROM85 is another of those variables that my old IMs are missing so god only knows what is going on here. I'll guess that it is looking for the 128K roms.
```
51FE: 4A53           'JS'             TST     (A3)
5200: 6D20           2005222          BLT.S   lho_4
5202: 42A7           'B.'             CLR.L   -(A7)
5204: 3F3C 008F      '?<..'           PUSH    #143
5208: 4EAD 00E2      1000AC6          JSR     proc38(A5)
```
Well, let's see here. Proc38 uses the passed parm as a trap number and returns that traps address on the stack.
```
520C: 42A7           'B.'             CLR.L   -(A7)
```
Note that the trap address has not been popped off the stack. So when these next instructions are done, that address will still be on the stack.
```
520E: 3F3C 009F      '?<..'           PUSH    #159
5212: 4EAD 00E2      1000AC6          JSR     proc38(A5)
```
Get another trap address on the stack,
```
5216: 201F           ' .'             POP.L   D0
```
and put it in D0, leaving the first trap address on the stack.
```
5218: B09F           '..'             CMP.L   (A7)+,D0
```
Now, compare the two trap addresses,
```
521A: 56C0           'V.'             SNE     D0
```
and set the low byte of D0 to FF hex if they are not the same.
```
521C: 4400           'D.'             NEG.B   D0
```
Do 2's complement - make the low byte of D0 its own negative. Since D0's byte is either 0 or FF (from the SNE), the NEG will make it either 0 (if it was 0) or 1 (if it was FF) - (for NEG, invert the bits, then add a binary 1).
```
521E: 1B40 F4F5         -$B0B         MOVE.B  D0,glob27(A5)
```
And save this number.
```
5222: 42A7           'B.'    lho_4    CLR.L   -(A7)
5224: 2F3C 0001 0000 '/<....'         PUSH.L  #$10000
522A: 4EAD 009A      1000A5C          JSR     NewHandle(A5)
```
Get a new Handle for a block of size 10000 hex.
```
522E: 2B5F F4F0         -$B10         POP.L   glob26(A5)
```
And save the handle.
```
5232: 6708           200523C          BEQ.S   lho_5
```
Branch if a NIL pointer (meaning the memory was not available) is popped off the stack.
```
5234: 487A FE26      200505C          PEA     MYGROWZO
```
Otherwise setup a grow zone function.
```
5238: 4EAD 0092      1000A1E          JSR     SetGrowZone(A5)
```
A grow zone procedure is a custom method for handling low memory conditions and overrides

the memory managers routines. Not a great description, but we don't really care about this.

```
523C: 4CDF 1880      'L...'  lho_5    MOVEM.L (A7)+,D7/A3-A4      Restore those saved regs,
5240: 4E5E           'N^'            UNLK    A6                  Kill the stack frame,
5242: 4E75           'Nu'            RTS                         And return to the caling proc.


5244: D345 5455 5020 2020    data257  DNAME   SETUP   ,0


524C: '..'                   data258  DC.W    8

                             ;-refs -  2/SETUP

524E: 4943                   data259  DC.B    'ICON'

                             ;-refs -  2/SETUP

5252: 5041                   data260  DC.B    'PACK'
```

The DRAWRESS Procedure

```
1F6E:                                QUAL    DRAWRESS ; b# =284  s#2  =proc148

                             vfp_1    VEQU  -256            One local variable.
                             param1   VEQU  8               One parameter needed.
1F6E:                                VEND

                             ;-refs -  2/DRAWFHIN   2/SETUP      2/DRAWNUM
                             ;-        2/DRAWDHIN
```

OK, you should be able to just look at this and see what happens. First off, look at the trap, DrawString. It takes one parameter, a pointer to a string. Now, the previous line says to push the address of the local variable so this has to be the string pointer. Go back a few lines and we see that proc5 is being called with two parameters: the string pointer, and the parameter from the calling procedure. You can deduce that proc5 has to get a string from somewhere, and probably will call the GetString trap or some equivalent. In fact, if you look at proc5, you will see that it calls GetResource (resource type STR#). This returns a handle to the STR# resource. Proc5 then uses the second parameter to figure out which string the calling procedure really wants. Proc5 loops through the STR# resource until it comes to the right string, then moves a pointer to the string into the first parameter and returns. When it gets back here, vfp_1 contains a pointer to the string.

```
1F6E: 4E56 FF00      'NV..'  DRAWRESS LINK    A6,#-$100
1F72: 486E FF00      200FF00          PEA     vfp_1(A6)
1F76: 3F2E 0008      2000008          PUSH    param1(A6)
1F7A: 4EAD 002A      100048C          JSR     proc5(A5)
1F7E: 486E FF00      200FF00          PEA     vfp_1(A6)      At this point, vfp_1 has the stringptr.
1F82: A884           '..'             _DrawString ; (s:Str255)
1F84: 4E5E           'N^'             UNLK    A6
1F86: 205F           ' _'             POP.L   A0
1F88: 544F           'TO'             ADDQ    #2,A7
```

```
1F8A: 4ED0           'N.'              JMP      (A0)
```

Note that there is no RTS instruction to return. The subroutine uses a common substitute. First it pops the return address off the stack (which is actually what the RTS would have done anyways) and then does an indirect JMP (A0). This just means to jump to whatever A0 points to and A0 points to the return address.

```
1F8C: C452 4157 5245 5353    data125  DNAME    DRAWRESS,0,0
```

The MAKEAWIN Procedure

```
5852:                                   QUAL    MAKEAWIN ; b# =490  s#2  =proc209

                              vhy_1      VEQU  -12          Two local variables, no parms passed.
                              vhy_2      VEQU  -8
5852:                                   VEND

                              ;-refs -  1/DA Mover

5852: 4E56 FFF0       'NV..'  MAKEAWIN LINK    A6,#-$10
5856: 42A7            'B.'             CLR.L   -(A7)       These instructions are setting up the GetNewDialog
                                                             below. 1st, clear space for the DialogPtr.
5858: 3F3C 000A       '?<..'           PUSH    #10         Push the Dialog ID #.
585C: 42A7            'B.'             CLR.L   -(A7)       Push a NIL pointer for wStorage
585E: 70FF            'p.'             MOVEQ   #-1,D0
5860: 2F00            '/.'             PUSH.L  D0          Push a 32 bit -1 (IM says to do this to make the
                                                             dialog the frontmost window).
 5862:  A97C                 '.|'                  _GetNewDialog ; (DlgID:INTEGER; wStorage:Ptr;
                                                             behind:WindowPtr):DialogPtr
5864: 2B5F FFFA             -6          POP.L   glob67(A5)  And pop off the dialogPtr.  This will be used by proc
                                                             MAKEBOX.
5868: 486D FEC4           -$13C         PEA     glob48(A5)
586C: 3F3C 000A       '?<..'           PUSH    #10         This is the dialog item - the left list box if you check
                                                             Resedit.
5870: 4EBA FF32       20057A4          JSR     MAKEBOX     Well, after inspecting this procedure, it looks like
                                                             more can be determined by just looking at these few
                                                             instructions here.  Notice that MakeBox is being
                                                             called with two parameters: The 1st being an
                                                             unknown global variable, and the second being one
                                                             of the two list boxes in Mover's main dialog.  So it
                                                             looks like MakeBox is just performing some
                                                             housekeeping on these two list boxes.
5874: 486D FEC8           -$138         PEA     glob49(A5)
5878: 3F3C 000B       '?<..'           PUSH    #11         Now do the right list box.
587C: 4EBA FF26       20057A4          JSR     MAKEBOX
5880: 206D FEC4           -$13C        MOVEA.L glob48(A5),A0       Get the address in (not of) glob48 into A0,
5884: 2050            ' P'             MOVEA.L (A0),A0     and dereference it - or get whatever glob48 was
                                                             pointing at into A0.
5886: 216D FEC8 0004      -$138        MOVE.L  glob49(A5),4(A0)    Now move glob49 (a pointer I suspect) into
                                                             4 past A0.  So glob48 contains a pointer which points
                                                             four bytes behind the pointer in glob49.
588C: 206D FEC8           -$138        MOVEA.L glob49(A5),A0       Now do the exact opposite.  Grab the pointer
                                                             in glob49 and stick the pointer in glob48 4 bytes past
                                                             it.
5890: 2050            ' P'             MOVEA.L (A0),A0
5892: 216D FEC4 0004      -$13C        MOVE.L  glob48(A5),4(A0)
```

These last few instructions were kind of a mess because we don't no anything about how globs 48 and 49 will be used. We will come back here after looking at MainEven and particularly HandleBu. It will turn out that these two globals are pointers (or maybe handles, we don't really care) to the two list boxes on the main dialog. In addition, each pointer as a way of referring to the other list box. At this point, this does not make any sense, but later on, glob 50 will be set to either glob48 or glob 49 (or NIL) depending on which list box - if any - has a selection made in it. The reason that glob48 and glob49 need to refer to each other, is that glob50 will be used to check both list boxes to see if their associated volumes are locked. See HandleBu for details.

```
5898: 2F2D FFFA           -6          PUSH.L  glob67(A5)
589C: 3F3C 0002      '?<..'           PUSH    #2            Item is the Copy button.
58A0: 486E FFF4      200FFF4          PEA     vhy_1(A6)
58A4: 486D FFF6          -$A          PEA     glob66(A5)    This will save a handle to it.
58A8: 486E FFF8      200FFF8          PEA     vhy_2(A6)
58AC: A98D               '..'          _GetDItem ; (dlg:DialogPtr; itemNo:INTEGER; VAR
                                                  kind:INTEGER; VAR item:Handle; VAR
                                                  box:Rect)
58AE: 2F2D FFFA           -6          PUSH.L  glob67(A5)
58B2: 3F3C 0006      '?<..'           PUSH    #6            Item is the left Open button.
58B6: 486E FFF4      200FFF4          PEA     vhy_1(A6)
58BA: 486D FFEC         -$14          PEA     glob63(A5)    This will save a handle to it.
58BE: 486E FFF8      200FFF8          PEA     vhy_2(A6)
58C2: A98D               '..'          _GetDItem ; (dlg:DialogPtr; itemNo:INTEGER; VAR
                                                  kind:INTEGER; VAR item:Handle; VAR
                                                  box:Rect)
58C4: 2F2D FFFA           -6          PUSH.L  glob67(A5)
58C8: 3F3C 0007      '?<..'           PUSH    #7            Item is the right Open button.
58CC: 486E FFF4      200FFF4          PEA     vhy_1(A6)
58D0: 486D FFF0         -$10          PEA     glob64(A5)    This will save a handle to it.
58D4: 486E FFF8      200FFF8          PEA     vhy_2(A6)
58D8: A98D               '..'          _GetDItem ; (dlg:DialogPtr; itemNo:INTEGER; VAR
                                                  kind:INTEGER; VAR item:Handle; VAR
                                                  box:Rect)
```

Now the program is going to assign dialog procedures to various of its items. Items 12 and 13 - the two filename boxes are assigned the DrawName proecdure. Items 14 - the size selected box - gets DrawSize. Item 15 -the font text demo box - gets DrawHint. Items 16 through 18 - various lines in the dialog box - get DrawGray. And items 19 and 20 - the free space on disk boxes - get DrawFree. If you examine SetDProc, you will see that it simply invokes GetDItem to get a handle to the dialog item (passed from the list below) and then uses SetDItem to set the dialogProcPtr to the procedure passed from the list below.

```
58DA: 3F3C 000C      '?<..'           PUSH    #12
58DE: 487A FB2E      200540E          PEA     DRAWNAME
58E2: 4EBA FE7E      2005762          JSR     SETDPROC
58E6: 3F3C 000D      '?<..'           PUSH    #13
58EA: 487A FB22      200540E          PEA     DRAWNAME
58EE: 4EBA FE72      2005762          JSR     SETDPROC
58F2: 3F3C 000E      '?<..'           PUSH    #14
58F6: 487A FC32      200552A          PEA     DRAWSIZE
58FA: 4EBA FE66      2005762          JSR     SETDPROC
58FE: 3F3C 000F      '?<..'           PUSH    #15
5902: 487A FA3A      200533E          PEA     DRAWHINT
```

```
5906: 4EBA FE5A      2005762          JSR     SETDPROC
590A: 3F3C 0010      '?<..'           PUSH    #16
590E: 487A FE1C      200572C          PEA     DRAWGRAY
5912: 4EBA FE4E      2005762          JSR     SETDPROC
5916: 3F3C 0011      '?<..'           PUSH    #17
591A: 487A FE10      200572C          PEA     DRAWGRAY
591E: 4EBA FE42      2005762          JSR     SETDPROC
5922: 3F3C 0012      '?<..'           PUSH    #18
5926: 487A FE04      200572C          PEA     DRAWGRAY
592A: 4EBA FE36      2005762          JSR     SETDPROC
592E: 3F3C 0013      '?<..'           PUSH    #19
5932: 487A FD12      2005646          PEA     DRAWFREE
5936: 4EBA FE2A      2005762          JSR     SETDPROC
593A: 3F3C 0014      '?<..'           PUSH    #20
593E: 487A FD06      2005646          PEA     DRAWFREE
5942: 4EBA FE1E      2005762          JSR     SETDPROC
5946: 2F2D FFFA           -6          PUSH.L  glob67(A5)    Now the dialog is made the current Port
594A: A873           '.s'             _SetPort ; (port:GrafPtr)
594C: 2F2D FFFA           -6          PUSH.L  glob67(A5)    and make the dialog visible,
5950: A915           '..'             _ShowWindow ; (theWindow:WindowPtr)
5952: 2F2D FFFA           -6          PUSH.L  glob67(A5)    and make it the frontmost window.
5956: A91F           '..'             _SelectWindow ; (theWindow:WindowPtr)
5958: 3F3C 0002      '?<..'           PUSH    #2
595C: 4EBA A78A      20000E8          JSR     DIMITEM       These instructions dim the two Open buttons.
5960: 3F3C 0003      '?<..'           PUSH    #3
5964: 4EBA A782      20000E8          JSR     DIMITEM
5968: 2F2D FFFA           -6          PUSH.L  glob67(A5)
596C: A981           '..'             _DrawDialog ; (dlg:DialogPtr)  And finally, draw the damn thing.
596E: 4E5E           'N^'             UNLK    A6
5970: 4E75           'Nu'             RTS


5972: CD41 4B45 4157 494E    data270  DNAME   MAKEAWIN,0,0
```

The MAKEBOX Procedure.

```
57A4:                                 QUAL    MAKEBOX ; b# =488  s#2  =proc208

                              vhx_1    VEQU  -14
                              vhx_2    VEQU  -10
                              vhx_3    VEQU  -8
                              vhx_4    VEQU  -4
                              param2   VEQU  8              Parm 2 is the dialog item #
                              param1   VEQU  10
57A4:                                 VEND

                              ;-refs -  2/MAKEAWIN

57A4: 4E56 FFF2      'NV..'  MAKEBOX  LINK    A6,#-$E
57A8: 48E7 0018      'H...'           MOVEM.L A3-A4,-(A7)
57AC: 266E 000A      200000A          MOVEA.L param1(A6),A3        A3 gets whatever is in parm 1.
57B0: 2F2D FFFA           -6          PUSH.L  glob67(A5)    Push the DialogPtr,
57B4: 3F2E 0008      2000008          PUSH    param2(A6)    And push the item #.
57B8: 486E FFF6      200FFF6          PEA     vhx_2(A6)     This will get the Kind.
```

```
57BC: 486E FFF2      200FFF2          PEA     vhx_1(A6)      This will get the ItemHandle.
57C0: 486E FFF8      200FFF8          PEA     vhx_3(A6)      This will get the Box.
57C4: A98D           '..'             _GetDItem ; (dlg:DialogPtr; itemNo:INTEGER; VAR
                                                 kind:INTEGER; VAR item:Handle; VAR
                                                 box:Rect)
57C6: 2F2D FFFA           -6          PUSH.L  glob67(A5)     Now push the dialogPtr and item again...
57CA: 3F2E 0008      2000008          PUSH    param2(A6)
57CE: 3F2E FFF6      200FFF6          PUSH    vhx_2(A6)      Push the item Kind
57D2: 487A F662      2004E36          PEA     DRAWBOX        See IM - this is a procPtr.
57D6: 486E FFF8      200FFF8          PEA     vhx_3(A6)      And push the Box
57DA: A98E           '..'             _SetDItem ; (dlg:DialogPtr; itemNo,kind:INTEGER;
                                                 item:Handle; box:Rect)
57DC: 42A7           'B.'             CLR.L   -(A7)
57DE: 7064           'pd'             MOVEQ   #100,D0
57E0: 2F00           '/.'             PUSH.L  D0
57E2: 4EAD 009A      1000A5C          JSR     NewHandle(A5)
57E6: 269F           '&.'             POP.L   (A3)           Get a new handle - size 100 - and put it into parm1
                                                             (which A3 points to).
57E8: 2053           ' S'             MOVEA.L (A3),A0        A0 gets the handle.
57EA: 2850           '(P'             MOVEA.L (A0),A4        And A4 gets the pointer. OK, A0 is a handle meaning
                                                             it points to a pointer which in turn points to whatever
                                                             it is we care about (in this case, a free block of
                                                             memory). That means that (A0) grabs what ever A0
                                                             points to which is (by definition of a handle) the
                                                             pointer.
57EC: 28AD FFFA           -6          MOVE.L  glob67(A5),(A4)    And now we put the dialogPtr into the block
                                                             of memory gotten by NewHandle.
57F0: 426C 0060      'Bl.`'           CLR     96(A4)         Remember, A4 points (its a pointer, not a handle!) to
                                                             a block of memory, 100 bytes long. So this
                                                             instruction simply clears the 96 byte in that block.
57F4: 204C           ' L'             MOVEA.L A4,A0          Put the pointer into A0.
57F6: 5088           'P.'             ADDQ.L  #8,A0          Add 8 to A0. Previously we had stored the dialogPtr
                                                             at the beginning of this block. Since a pointer is 8
                                                             bytes long, A0 no points to the first byte after the
                                                             dialogPtr.
57F8: 43EE FFF8      200FFF8          LEA     vhx_3(A6),A1   vhx_3 is a Box which is of type Rect which is 4
                                                             integers, or 4 words, or two long words.
57FC: 20D9           ' .'             MOVE.L  (A1)+,(A0)+
57FE: 20D9           ' .'             MOVE.L  (A1)+,(A0)+    So move the Box information into the free memory
                                                             right after the dialogPtr and increment A0 to the next
                                                             free byte.
5800: 302E FFFC      200FFFC          MOVE    vhx_4(A6),D0   This is tough since we don't know what vhx_4 is to
                                                             start with.
5804: 906E FFF8      200FFF8          SUB     vhx_3(A6),D0   But whatever, subtrack vhx_3 from it, result in D0.
5808: 48C0           'H.'             EXT.L   D0             At this point, D0 is accurate to the word length (since
                                                             that was all the SUB specified). This will make it's
                                                             sign (negative or posative) accurate to all 32 bits.
580A: 81FC 0010      '....'           DIVS    #16,D0         Now, divide by 16.
```

```
580E: 3940 0062      '9@.b'           MOVE    D0,98(A4)      And put this value (whatever it is) in the last two
                                                             bytes (notice it is a word length instruction) of the
                                                             memory block.
5812: 426C 0058      'Bl.X'           CLR     88(A4)
5816: 397C FFFF 0056 '9|...V'         MOVE    #$FFFF,86(A4)
581C: 422C 0014      'B,..'           CLR.B   20(A4)         These last instructions are filling in various parts of
                                                             the memory block.
5820: 206D FFFA           -6          MOVEA.L glob67(A5),A0       Put the DialogPtr back in A0.
5824: 2153 0098      '!S..'           MOVE.L  (A3),152(A0)   A3 still points to parm1.
5828: 2F13           '/.'             PUSH.L  (A3)           So, this effectively pushes parm1
582A: 4EBA AEA4      20006D0          JSR     MAKESBAR       This is fairly complicated, but this procedure makes a
                                                             scroll bar for the dialog item.
582E: 2053           ' S'             MOVEA.L (A3),A0
5830: 2050           ' P'             MOVEA.L (A0),A0        Can't tell what these instructions are doing.
5832: 2068 0010      ' h..'           MOVEA.L 16(A0),A0
5836: 2050           ' P'             MOVEA.L (A0),A0
5838: 2153 0024      '!S.$'           MOVE.L  (A3),36(A0)
583C: 4CDF 1800      'L...'           MOVEM.L (A7)+,A3-A4
5840: 4E5E           'N^'             UNLK    A6
5842: 205F           ' _'             POP.L   A0             Pop off the return address.
5844: 5C4F           '\O'             ADDQ    #6,A7
5846: 4ED0           'N.'             JMP     (A0)           And jump back to the calling procedure.
5848: CD41 4B45 424F 5820    data269  DNAME   MAKEBOX ,0,0
```

The MAINEVEN Procedure

Basically, the main loop consists of a set of housekeeping routines, a call to ModalDialog to read dialog events that take place, and a simple jump table to handle the various events. D7 needs to be zero for the loop to keep running. If an error occurs, or the user hits Quit, D7 is changed to one and the procedure exits. First, DA Mover attempts to allocate a large block of memory (10000 hex) into glob26. If this is successful (or glob26 already has a memory handle) then the program skips down to make some more checks - otherwise a memory error is generated. Next, the procedure checks to see if there are any files open and if so, calls FlushVol to write any changes to disk.

```
   0:                                QUAL    MAINEVEN ; b# =1  s#1  =proc1

                            vab_1     VEQU  -6
   0:                               VEND

                            ;-refs -  1/DA Mover

   0: 4E56 FFF8      'NV..'  MAINEVEN LINK    A6,#-8
   4: 48E7 0308      'H...'           MOVEM.L D6-D7/A4,-(A7)
   8: 4207           'B.'             CLR.B   D7             Enable the Main Event Loop.
   A: 4AAD F4F0        -$B10 lab_1    TST.L   glob26(A5)     glob46 will (or does) contain a handle to a large
                                                             block of memory. So, if glob26 already has the
                                                             handle, branch down, otherwise try to get some
                                                             memory.
   E: 661C           100002C          BNE.S   lab_2
  10: 42A7           'B.'             CLR.L   -(A7)          Clear stack space for the returned handle.
  12: 2F3C 0001 0000 '/<....'         PUSH.L  #$10000        Size of memory block needed.
  18: 4EBA 0A42      1000A5C          JSR     NewHandle
```

```
1C: 2B5F F4F0        -$B10          POP.L   glob26(A5)      And get the handle in glob26.
20: 660A           100002C          BNE.S   lab_2           Remember, a NIL handle or pointer is all zeroes.
                                                            glob26 either has a valid handle or a NIL handle.  If it
                                                            is valid, branch.
22: 3F3C 0032      '?<.2'           PUSH    #50
26: 4EAD 01CA      200023C          JSR     DOALERT(A5)     Otherwise do some memory alert (you can check this
                                                            if you like.)
2A: 7E01           '~.'             MOVEQ   #1,D7           and disable the main event loop.
2C: 1007           '..'    lab_2    MOVE.B  D7,D0
2E: 6600 00D0      1000100          BNE     lab_15          Go if loop disabled from above.
32: 206D FEC4        -$13C          MOVEA.L glob48(A5),A0       Get reference to left list box.
36: 2850           '(P'             MOVEA.L (A0),A4
38: 4A6C 0058      'Jl.X'           TST     88(A4)          Look at the descrpition of FlushVol (next paragraph)
                                                            to see what this variable means.
3C: 670E           100004C          BEQ.S   lab_3           Seeing that 88(A4) is the VRefNum, then branch if it
                                                            is zero (no volume available - i.e. the list box has no
                                                            opened file in it).
3E: 4267           'Bg'             CLR     -(A7)           Space for function result (OSErr).
40: 42A7           'B.'             CLR.L   -(A7)           iovNamePtr parameter (NIL).
42: 3F2C 0058      '?,.X'           PUSH    88(A4)          iovRefNum parameter.
46: 4EBA 0BAE      1000BF6          JSR     FlushVol        If a volume is available, flush it.
4A: 3C1F           '<.'             POP     D6              Pop off error code.
```

```
4C: 206D FEC8        -$138 lab_3    MOVEA.L glob49(A5),A0
50: 2850            '(P'            MOVEA.L (A0),A4
52: 4A6C 0058       'Jl.X'          TST     88(A4)
56: 670E            1000066         BEQ.S   lab_4
58: 4267            'Bg'            CLR     -(A7)
5A: 42A7            'B.'            CLR.L   -(A7)
5C: 3F2C 0058       '?,.X'          PUSH    88(A4)
60: 4EBA 0B94       1000BF6         JSR     FlushVol
64: 3C1F            '<.'            POP     D6
```

Now do the same thing with the right list box.

Lets take a quick look and FlushVol and we can see a couple of things. Fist of all, we can quickly see what the parameters are: Parm1 is a pointer to the Volume Name, Parm2 is the Volume Ref Number. Looking back at MainEven, we see that the PEA 88(A4) is referring to the Volume Reference Number. FlushVol "writes the contents of the associated volume buffer and descriptive informatin about the volume (if they've changed since the last time FlushVol was called)." [IM II pg 89]. The returned result of this procedure is the OSErr.

```
                              ;-refs -  1/MAINEVEN   2/FLUSHRES   2/REMOVEST

BF6: 4E56 FFC0      'NV..'  FlushVol LINK    A6,#-$40
BFA: 41EE FFC0      200FFC0          LEA     vbu_1(A6),A0
BFE: 316E 0008 0016 2000008          MOVE    param2(A6),ioVRefNum(A0)
C04: 216E 000A 0012 200000A          MOVE.L  param1(A6),ioNamePtr(A0)
C0A: A013           '..'             _FlushVol ; (A0|IOPB:ParamBlockRec):D0\OSErr
C0C: 3D40 000E      200000E          MOVE    D0,funRslt(A6)
C10: 4E5E           'N^'             UNLK    A6
C12: 225F           '"_'             POP.L   A1
C14: 5C8F           '\.'             ADDQ.L  #6,A7
C16: 4ED1           'N.'             JMP     (A1)
```

back to MainEven

```
66: 4EAD 020A      2005DCA lab_4    JSR     HANDLEBU(A5)
6A: 486D 0212      2005EF8          PEA     MYFILTER(A5)
6E: 486E FFFA      200FFFA          PEA     vab_1(A6)
72: A991           '..'             _ModalDialog ; (filterProc:ProcPtr; VAR itemHit:INTEGER)
```

ModalDialog is the all-purpose dialog handler. It will monitor events and wait for an event involving an active dialog item. Upon returning, the dialog item number is returned in ModalDialog's 2nd parameter - in this case, vab_1. Once the trap returns, the program has to figure out what to do now that an item has been activated. Below, is a simple jump table that repeatedly subtracts integers from vab_1 until it is zero, at which point the program knows that it has the proper dialog item. It then branches to the appropriate routine.

ModalDialog also takes a parameter that specifies a special procedure that it can call whenever an event occurs. What that means, is that the line PEA MYFILTER is telling ModalDialog to execute the procedure MYFILTER anytime an event occurs. We can take a look at MYFILTER to see what it is doing (although in cracking, we probably don't care). Right now I will guess that MYFILTER is taking care of things like allowing multiple selections in the list boxes,. displaying the font string, and displaying the size of the selection.

```
74: 302E FFFA      200FFFA          MOVE    vab_1(A6),D0
78: 5540           'U@'             SUBQ    #2,D0          Copy button.
7A: 6736           10000B2          BEQ.S   lab_7
7C: 5340           'S@'             SUBQ    #1,D0          Remove Button.
7E: 672C           10000AC          BEQ.S   lab_6
80: 5340           'S@'             SUBQ    #1,D0          Help Button.
82: 6734           10000B8          BEQ.S   lab_8
84: 5340           'S@'             SUBQ    #1,D0          Quit Button.
86: 6720           10000A8          BEQ.S   lab_5
88: 5340           'S@'             SUBQ    #1,D0          Left Open/Close Button.
8A: 673C           10000C8          BEQ.S   lab_10
8C: 5340           'S@'             SUBQ    #1,D0          Right Open/Close Button.
8E: 6742           10000D2          BEQ.S   lab_11
90: 5340           'S@'             SUBQ    #1,D0          Font Radio Button.
92: 672A           10000BE          BEQ.S   lab_9
94: 5340           'S@'             SUBQ    #1,D0          DA Radio Button.
96: 6726           10000BE          BEQ.S   lab_9
98: 5340           'S@'             SUBQ    #1,D0          Left List Box.
9A: 6740           10000DC          BEQ.S   lab_12
9C: 5340           'S@'             SUBQ    #1,D0          Right List Box.
9E: 674A           10000EA          BEQ.S   lab_13
A0: 0440 0028      '.@.('           SUBI    #40,D0         We will have to check MyFilter to see what this is
                                                           doing.
A4: 6752           10000F8          BEQ.S   lab_14
A6: 6058           1000100          BRA.S   lab_15
A8: 7E01           '~.'    lab_5    MOVEQ   #1,D7          User hit Quit, so disable the loop and jump to the
                                                           loop end.
AA: 6054           1000100          BRA.S   lab_15
AC: 4EAD 01E2      2004E98 lab_6    JSR     REMOVEST(A5)   Remove Button.
B0: 604E           1000100          BRA.S   lab_15
B2: 4EAD 01EA      2004F5C lab_7    JSR     COPYSTUF(A5)   Copy Button.
B6: 6048           1000100          BRA.S   lab_15
B8: 4EAD 023A      2006B0E lab_8    JSR     DOHELP(A5)     Help Button.
BC: 6042           1000100          BRA.S   lab_15
BE: 3F2E FFFA      200FFFA lab_9    PUSH    vab_1(A6)      Push the selected item number,
C2: 4EAD 022A      20064F2          JSR     SELCLICK(A5)   and change to either Fonts or DAs.
C6: 6038           1000100          BRA.S   lab_15
C8: 2F2D FEC4        -$13C lab_10   PUSH.L  glob48(A5)
CC: 4EAD 0242      2006B50          JSR     DOCFILE(A5)    Left Open (or close) Button.
D0: 602E           1000100          BRA.S   lab_15
D2: 2F2D FEC8        -$138 lab_11   PUSH.L  glob49(A5)
D6: 4EAD 0242      2006B50          JSR     DOCFILE(A5)    Right Open (or close) Button.
DA: 6024           1000100          BRA.S   lab_15
DC: 2F2D FEC4        -$13C lab_12   PUSH.L  glob48(A5)     Remember this guy?  Refers to the left box.
E0: 2F2D FFE8         -$18          PUSH.L  glob62(A5)
E4: 4EAD 0202      2005CB8          JSR     CONTENTC(A5)   Handle a list box click.
E8: 6016           1000100          BRA.S   lab_15
EA: 2F2D FEC8        -$138 lab_13   PUSH.L  glob49(A5)     Refers to the right list box.
EE: 2F2D FFE8         -$18          PUSH.L  glob62(A5)
F2: 4EAD 0202      2005CB8          JSR     CONTENTC(A5)   List box handler.
F6: 6008           1000100          BRA.S   lab_15
F8: 3F2D FEDC        -$124 lab_14   PUSH    glob55(A5)
FC: 4EAD 0232      2006A6A          JSR     HANDLEIN(A5)
```

```
 100: 1007           '..'    lab_15   MOVE.B  D7,D0
 102: 6700 FF06      100000A          BEQ     lab_1
 106: 4CDF 10C0      'L...'           MOVEM.L (A7)+,D6-D7/A4
 10A: 4E5E           'N^'             UNLK    A6
 10C: 4E75           'Nu'             RTS

 10E: CD41 494E 4556 454E    data1    DNAME   MAINEVEN,0,0
```

Here is the end of the main loop. This checks to see if the loop should terminate. If not, branch back to the beginning of the loop.

At this point, either an error occurred or the user has hit the Quit button.

HANDLEBU Procedure

```
5DCA:                               QUAL    HANDLEBU ; b# =501  s#2  =proc214

                            vid_1    VEQU  -272
                            vid_2    VEQU  -256
5DCA:                               VEND
```

A quick observation here. After scanning the first few lines, you can notice some hereto unknown things. Look at the references to glob50. At this point, we know that globs 48 and 49 have been set up to refer (we don't know exactly how) to the two list boxes in the main dialog. A quick look down a ways reveals that D7 is used to pass an integer to our DrawString procedure (proc5). If we assume that D7 is the ID # of the STR# resource (since this is the parameter that proc5 requires), then that MOVEQ 1,D7 (line 5) must refer to STR# 1 which reads "Copy". The next two strings in the resource are "<<Copy<<" and ">>Copy>>" which are exactly the three strings that the copy button on the dialog can contain. So we might assume right now that glob50 refers to one of the list boxes, and can be used to determine whether the user has selected an item(s) in the list box. Based upon this information, the procedure will fill in the copy button with the proper string.

```
                            ;-refs -  1/MAINEVEN

5DCA: 4E56 FEEE      'NV..'  HANDLEBU LINK    A6,#-$112
5DCE: 48E7 0308      'H...'           MOVEM.L D6-D7/A4,-(A7)
5DD2: 4AAD FECC        -$134          TST.L   glob50(A5)
5DD6: 660C           2005DE4          BNE.S   lid_1
5DD8: 7E01           '~.'             MOVEQ   #1,D7
5DDA: 3F3C 0003      '?<..'           PUSH    #3
5DDE: 4EBA A308      20000E8          JSR     DIMITEM
5DE2: 607A           2005E5E          BRA.S   lid_8
```

Check the list box (it seems)

Look at what this branch is skipping.

The string is "Copy".

DIMITEM needs the item number to dim as a parm. Item 3 is the Remove button.

Take a quick look at DIMITEM and you will see that it takes an item number as a parameter, pushes the parameter, then pushes the number -1 (255 if we are talking about signed numbers) and calls HILITEIT which uses the 2nd parameter to either set or dim the desired button.

The Remove button is dimmed anytime there is no selection in one of the list boxes. How did this procedure know there was no selection? It

checked to see if glob50 was blank (or possible a NIL pointer) and if so, there is no selection.

```
5DE4: 202D FECC        -$134 lid_1    MOVE.L  glob50(A5),D0
5DE8: B0AD FEC4          -$13C        CMP.L   glob48(A5),D0
5DEC: 6604           2005DF2          BNE.S   lid_2
5DEE: 7E03           '~.'             MOVEQ   #3,D7
5DF0: 6002           2005DF4          BRA.S   lid_3
5DF2: 7E02           '~.'    lid_2    MOVEQ   #2,D7
5DF4: 206D FECC        -$134 lid_3    MOVEA.L glob50(A5),A0
5DF8: 2050           ' P'             MOVEA.L (A0),A0
5DFA: 2068 0004      ' h..'           MOVEA.L 4(A0),A0
5DFE: 2050           ' P'             MOVEA.L (A0),A0
5E00: 3C28 0058      '<(.X'           MOVE    88(A0),D6
5E04: 206D FECC          -$134        MOVEA.L glob50(A5),A0
5E08: 2050           ' P'             MOVEA.L (A0),A0
5E0A: 2068 0004      ' h..'           MOVEA.L 4(A0),A0
5E0E: 2050           ' P'             MOVEA.L (A0),A0
5E10: 4A68 0056      'Jh.V'           TST     86(A0)
5E14: 6C02           2005E18          BGE.S   lid_4
5E16: 4246           'BF'             CLR     D6
5E18: 4A46           'JF'    lid_4    TST     D6
5E1A: 57C0           'W.'             SEQ     D0
5E1C: 4A00           'J.'             TST.B   D0
5E1E: 6616           2005E36          BNE.S   lid_5
5E20: 2F00           '/.'             PUSH.L  D0
5E22: 4267           'Bg'             CLR     -(A7)
5E24: 3F06           '?.'             PUSH    D6
```

Here is the real key. glob50 is being compared to glob48. We know glob48 has something to do with the left list box, and look what happens if they are the same...D7 gets 3 which means string">>Copy>>" - the user has made a selection in the left list box.

Otherwise the user has made a selection in the right list box. A quick note: glob50 was not compared to glob49, but it was compared to glob48. We can deduce from this that glob50 had to contain either glob48 or glob49. What this means is that glob50 seems to indicate that something has been selected in one of the list boxes or is empty if there is no selection.

This is a mess. We know that glob50 is a handle to a host of information about one of the list boxes, but we didn't bother to figure which bytes mean what. The best thing to do here is to analyze all the branches in the mess, see where they go, and look at what happens as a result of each branch. So...

Look familiar? Let's guess that this is a vRefNum for the list box containing the selection.

OK, here is a branch. If it executes, D6 has something (which we guessed to be a vRefNum) in it which gets passed on to lid_4.

Otherwise, D6 is zeroed (no volume available).

D0=FF hex if there is no volume.

This branch executes if D6 was zero and will cause 1 to moved into D7 - "Copy".

Save D0 on the stack (not a parameter)

Create space on the stack for the return value.

Aha! We were right. proc6 needs a vRefNum and here is good old D6 being pushed as a parm. D6 is indeed the vRefNum.

```
5E26: 4EAD 0032      10004CA          JSR     proc6(A5)
5E2A: 121F           '..'             POP.B   D1
5E2C: 201F           ' .'             POP.L   D0
5E2E: 8001           '..'             OR.B    D1,D0
5E30: 0240 0001      '.@..'           ANDI    #1,D0
5E34: 6702           2005E38          BEQ.S   lid_6
5E36: 7E01           '~.'    lid_5    MOVEQ   #1,D7
5E38: 4267           'Bg'    lid_6    CLR     -(A7)
5E3A: 206D FECC        -$134          MOVEA.L glob50(A5),A0
5E3E: 2050           ' P'             MOVEA.L (A0),A0
5E40: 3F28 0058      '?(.X'           PUSH    88(A0)
5E44: 4EAD 0032      10004CA          JSR     proc6(A5)
5E48: 101F           '..'             POP.B   D0
5E4A: 670A           2005E56          BEQ.S   lid_7
5E4C: 3F3C 0003      '?<..'           PUSH    #3
5E50: 4EBA A296      20000E8          JSR     DIMITEM
5E54: 6008           2005E5E          BRA.S   lid_8
5E56: 3F3C 0003      '?<..'  lid_7    PUSH    #3
5E5A: 4EBA A2AE      200010A          JSR     UNDIMITE
```

Takes a vRefNum as a parm, then does a GetVolInfo, and checks the iovAttributes to see if the disk is locked. Returns a 1 if locked, 0 if unlocked.

Pop off the locked status.

Pop off the original D0.

Or them so that, in effect, the AND instruction below will be ANDing both D0 and D1 with 1.

Check to see if one of the two contains a non-zero value,

and if so, do not put a 1 in D7 (the string is not "Copy").

String is "Copy" (meaning that DA Mover will not allow the Copy to proceed) and from the above code, we might guess that this is a result of the destination volume being locked so copying is impossible.

And here is basically the same as above except that the other list box's volume is being checked

Push the vRefNum of the volume from which the selection has been made.

Locked Volume?

Go if not locked.

If the volume is locked, we cannot remove anything so dim the Remove Button.

Else activate the Remove Button (volume is not locked).

OK, let's re-cap for a minute. If you look back at MakeAWin, you will note that glob48 and glob49 are set up to refer to information about the left and right list boxes respectively. We also know that these globs contain information about the volume (and possibly the file) that is being displayed in the list boxes - since 88 bytes off the start of the pointer is the volume reference number. The above code can be broken into two pieces: from line 5DF4, to lid_5 and from lid_6 to one line past lid_7. The first piece is messy, but the end result is that the destination volume is tested to see if it is locked, and if so, the copy button text is set to "Copy". Therefore we can now assume that all that messy stuff beforehand was in essence setting a pointer to the destination list box information. Remember from MakeAWin there was a strange section of code that seemed to link the two globs to each other? Well, now we see that glob50 is set to one of these two (the one that contains a selection) but glob50 must also be able to access the other list box's volume to see if it is locked (or to see if copying to it is possible). The second section checks to see if the volume containing the selection is locked, and if so, Removing is not possible.

```
5E5E: BE6D FFF4          -$C lid_8    CMP.W   glob65(A5),D7
```
Once again, we don't know what this glob means, but we can see what gets skipped if the branch executes. Once we know what gets skipped, we have a decent idea what the global means. Keep in mind that the global is being compared to D7 - the string resource ID #.

```
5E62: 6700 0082      2005EE6         BEQ     lid_13
```
So if glob65 contains the ID # in D7, skip to the end of the procedure.

```
5E66: 42A7           'B.'            CLR.L   -(A7)
5E68: A8D8           '..'            _NewRgn ; :RgnHandle
5E6A: 285F           '(_'            POP.L   A4
5E6C: 2F0C           '/.'            PUSH.L  A4
5E6E: A87A           '.z'            _GetClip ; (rgn:RgnHandle)
5E70: 486E FEF0      200FEF0         PEA     vid_1(A6)
5E74: 42A7           'B.'            CLR.L   -(A7)
5E76: 42A7           'B.'            CLR.L   -(A7)
5E78: A8A7           '..'            _SetRect ; (VAR r:Rect; left,top,right,bottom:INTEGER)
5E7A: 486E FEF0      200FEF0         PEA     vid_1(A6)
5E7E: A87B           '.{'            _ClipRect ; (r:Rect)
5E80: 486E FF00      200FF00         PEA     vid_2(A6)
5E84: 3F07           '?.'            PUSH    D7
5E86: 4EAD 002A      100048C         JSR     proc5(A5)
```
Once again, the DrawString procedure. D7 is the string # and vid_2 returns a pointer to the string.

```
5E8A: 2F2D FFF6          -$A         PUSH.L  glob66(A5)
```
Look at the trap below. glob66 HAS to be a CtlHdl (Handle to a control object on a dialog),

```
5E8E: 486E FF00      200FF00         PEA     vid_2(A6)
```
and vid_2 we already know has the string whose ID # is in D7. Since D7's string is "Copy", ">>Copy>>", or "<<Copy<<", we can assume that the control in question is the Copy Button.

```
5E92: A95F           '._'            _SetCTitle ; (Ctl:CtlHdl; title:Str255)
```

```
5E94: 3B47 FFF4          -$C          MOVE    D7,glob65(A5)
```
Here is a clue! glob65 gets set to the string ID# - now this makes sense. Back up a few lines, glob65 was compared to D7 and if they were equal, all this stuff gets skipped. Now glob65 gets set to D7. It looks like the program is checking to see whether the Copy Button already has the correct string in it. If not, the above code changes it and updates glob65 to the new string ID# so that next time through the event loop, glob65 has the current ID # of the Copy Button's text.

```
5E98: 7001           'p.'            MOVEQ   #1,D0
5E9A: B047           '.G'            CMP.W   D7,D0
```
Remember: if D7 is 1, the string is "Copy", and no copying is allowed - either because nothing is selected, or because the destination volume is locked.

```
5E9C: 660A           2005EA8         BNE.S   lid_9
```
If copying is to be allowed, then branch.

```
5E9E: 3F3C 0002      '?<..'          PUSH    #2
```
Refers to the Copy Button:

```
5EA2: 4EBA A266      200010A         JSR     UNDIMITE
```
and - wait a second. Notice that this is backward! It is dimming the copy button if copying is allowed! I'm not sure why it does this, but look down a few lines...

```
5EA6: 6008           2005EB0         BRA.S   lid_10
5EA8: 3F3C 0002      '?<..'  lid_9   PUSH    #2
5EAC: 4EBA A23A      20000E8         JSR     DIMITEM
5EB0: 2F0C           '/.'    lid_10  PUSH.L  A4
5EB2: A879           '.y'            _SetClip ; (rgn:RgnHandle)
5EB4: 2F0C           '/.'            PUSH.L  A4
5EB6: A8D9           '..'            _DisposRgn ; (rgn:RgnHandle)
5EB8: 7001           'p.'            MOVEQ   #1,D0
```
Here we go. Now, if D7 is 1, dim the copy button, otherwise enable it.

```
5EBA: B047           '.G'            CMP.W   D7,D0
5EBC: 660A           2005EC8         BNE.S   lid_11
5EBE: 3F3C 0002      '?<..'          PUSH    #2
5EC2: 4EBA A224      20000E8         JSR     DIMITEM
5EC6: 6008           2005ED0         BRA.S   lid_12
5EC8: 3F3C 0002      '?<..'  lid_11  PUSH    #2
5ECC: 4EBA A23C      200010A         JSR     UNDIMITE
5ED0: 206D FFF6          -$A lid_12  MOVEA.L glob66(A5),A0
```
We already saw (from the SetCTitle trap above) that glob66 is a handle to the Copy button.

```
5ED4: 2050           ' P'            MOVEA.L (A0),A0
```
Convert the handle to a pointer.

```
5ED6: 43EE FEF0      200FEF0         LEA     vid_1(A6),A1
5EDA: 5088           'P.'            ADDQ.L  #8,A0
```
Well, according to IM, adding 8 bytes to a pointer to a control record makes the pointer point to the a window that the control is in.

```
5EDC: 22D8           '".'            MOVE.L  (A0)+,(A1)+
```
So, move the WindowPtr to vid_1.

```
5EDE: 22D8           '".'            MOVE.L  (A0)+,(A1)+
```
and now move the Rect (next parameter in a control record) into vid_1.

```
5EE0: 486E FEF0      200FEF0         PEA     vid_1(A6)
5EE4: A92A           '.*'            _ValidRect ; (goodRect:Rect)
```
This trap tells the Window Manager not to update the region Rect.

```
5EE6: 4CDF 10C0      'L...'  lid_13  MOVEM.L (A7)+,D6-D7/A4
```
And, now we are finished.

```
5EEA: 4E5E           'N^'             UNLK    A6
5EEC: 4E75           'Nu'             RTS
```

I am not sure exactly what is going on there when it sets the button to the opposite that it is supposed to be, then sets it properly. I might hazard a guess that this technique somehow gurrantees that the region will get redrawn properly, but I really don't know - nor do I really care, for that matter. It is pretty clear what this procedure does - it updates the text and active status of the various buttons on the main dialog. Once this is done, MainEven can let the user make a selection, act upon the selection, and then the whole thing starts over.

```
5EEE: C841 4E44 4C45 4255    data276  DNAME   HANDLEBU,0,0
```

Well, that wraps up the intensive assembly listing. Font/DA Mover has many more procedures, but the idea here was to look at an assembly listing and apply the stuff at the beginning of the tutorial to a real life situation and see if you can guess what is going on. Next I will discuss the use of TMON, and finally we will look at cracking a real application: Sorcerer. (I am choosing this because it is easy, and I recently cracked it so it is still failry fresh in my mind.)

Using TMON

TMON, unlike Nosy, is a real-time monitor / debugger. We will be using TMON in several situations: to break into active dialog windows, to break into programs that Nosy won't decompile properly, or when Nosy produces such a massive listing that we need to trace the application to see what happens where. To install TMON, just drag the application and the init into the system folder and restart. The application can be launched to configure it, but you probably won't need to do this. If you *do* configure it, make sure you save the changes in a User Area in the System Folder.

TMON can be entered several ways: System Errors, Debugger traps (this is a great technique for breaking into tough programs), user specified traps, and by pressing the interrupt button on the side of your Mac. If you lack the interrupt button, use the Programmer's Key init - this allows you to hold down command and option and press the startup key on an extended keyboard.

Once in TMON, you are presented with a Menu bar and possibly some windows. A quick note about TMON windows. They can be resized and dragged only in the vertical directions. To change values in the various windows, click the insertion bar in front of the value to change and type right over the old value. Pressing Return chops off the line at the insertion bar, pressing Enter leaves the rest of the line as is. For example, lets say you are changing the address of a dump window. If it currently reads "Dump From 00000000" and you type 1234 over the first 4 values, you have two choices. Hitting Return at this point chops off the last 4 zeroes making the effective address 1234 hex. If you were to hit Enter instead, the remaining zeroes would remain making the effective address 12340000 hex. Here are what the various menu commands do:

Dump / Cmd-d and Asmbly / Cmd-a

Brings up either a dump window or an assembly window. The dump window lists hex and ascii codes for a block of memory and the assembly window disassembles memory. The first line allows you to specify where the window will start its listing: Dump (Assembly) From XXXXXXX where XXXXXXX is an effective address. You can move the insertion bar right into this line and type over whatever is there. You can enter an address directly, specify a register (and the window will start from the address contained in the register), or a register indirect (the window will start from the address in the register, but will remember the register address). Examples: Dump From:

1) 80FFCA Dump listing starts from the absolute address 80FFCA hex. If you scroll the window, the displayed address will change to the address of the first line in the listing.

)2 A5 Dump starts from the address contained in register A5. The address displayed on the Dump From line will be replaced with the address in register A5. If you scroll, the displayed address will again change to reflect the first line in the listing, and if A5 changes, the window will not change.

3) 0(A1) Dump starts from the address in A1 plus zero (in this case). The displayed address on the Dump From line does not change to the address in A1, rather it now displays 00000000(A1) indicating that the listing is anchored to the register. As you scroll, the

zeroes will change to reflect how many bytes from the address in A1 the first line in the listing is - also, if A1 changes, the window will automatically change to the new value of A1.

The most common entry for an assembly window is 0(PC) which says to disassemble from the program counter. Then as you step through the program, the window automatically scrolls so that the first line is where the program counter is. The windows list - from left to right - the address, any registers that contain that address (Note the P - for program counter - next to the first line when you disassemble from 0(PC)), the resource the listing comes from if any (assembly window only), and then either hex and ascii bytes, or disassembled instructions. In addition, the assembly window will display comments to the right, indicating the destination of branches. Additional dump windows can be activated by holding down Shift while clicking on Dump in the menu bar - this is the only display that can have multiple windows. You will find if handy to have, in addition to the dissambly window, a dump window anchored to the A7 register (so make the Dump From read 0(A7)) so that you can quickly see what addresses are being pushed on the stack. If you need to see what the actual data of these addresses are, just shift-click Dump to bring up successive dump windows, and make each window dump from successive addressess (4 bytes each) on the stack. Remember that the stack moves backwards, so the first thing pushed on the stack will be to the right (in the dump window) of the second thing pushed on the stack, etc.

Brkpts / Cmd-b

Allows the setting of up to eight breakpoints. Simply enter the address of the breakpoint into one of the 8 slots. To remove a breakpoint, type a hyphen for the first digit of the address to remove and hit return. Breakpoints cause TMON to halt execution of the application at the address of the breakpoint. I generally use breakpoints to skip out of long loops. For example, if you are stepping through a section of code and you find a DBRA loop (usually moving a section of data) where the data register has some god-awful value like 63 (often used to move strings), enter a breakpoint at the address of the instruction immediately after the DBRA and then exit. TMON will break execution after the loop has finished.

Regs / Cmd-r

Displays the 16 registers, PC, and status flags, any of which can be modified by typing right over the current values. The flags are displayed as the letter that I have been using - C for Carry, Z for Zero, etc. When the letter is capitalized, the flag is set. To change the value of the flag, simply change the capitalization.

Heap / Cmd-h

Displays memory blocks in the application heap zone. Basically this window lists all allocated blocks of memory in the applications heap zone (in the form of the pointers to the blocks), the size of the block, a digit that is meaningless to me, and the blocks status - either 1) Free, not allocated to anything yet, 2) Nonrel, non-relocatable, 3) Handle at ...., relocatable block with handle at the address specified, or 4) INVALID which means there is a big problem somewhere.

File / Cmd-f

Brings up a window listing all open resource files by file reference number. In most cases, the last number in the list refers to the System File. Entering a file reference number after the Resource file # prompt lists the files resources and where in memory they are. From left to right, the information displayed is: Resource type, Resource ID #, Attributes, location in memory. Attributes are as follows: R = System reference, H = Load into system heap, P = purgeable, L = locked, T = protected, 1 = pre-loaded (loaded at startup time), W = write into resource file. To return to a list of file reference numbers, click the insertion bar before the file number you previously typed in and hit return.

Exit / Cmd-e

Returns control to the Mac. Execution starts from the current value (which can be modified, of course, via the Regs window).

Gosub / Cmd-g

Same as Step (below), except that all JSR and BSR instructions are treated as a single instruction and the subroutine is called invisibly to you. In other words, this command executes exactly as if you had set a breakpoint immediately after the JSR or BSR and then exited. I often use this command the first time through a program to quickly find which JSR calls the subroutine that bombs. If you look at the Font/DA Mover listing above and condsider the Da Mover portion, imagine this as a protected program that Nosy won't handle. You are presented with several subroutines which you certainly don't want to spend valuable time tracing. So, you Gosub each one until you get a bomb. Then you know which one you need to spend time tracing.

Step / Cmd-s

Executes the instruction pointed to by the PC. This command allows you to execute a program one instruction at a time with one limitation (or boon) which is that traps are executed as if they were a single instruction. Use the Trace command to step through the actual ROM trap code. All windows that are affected by the executed instruction are updated automatically.

Trace / Cmd-t

Same as Step, except that ROM traps will be followed into their ROM code. You will never need to do this to crack a program, however if you want to see what a trap is really doing, use this command.

Num / Cmd-n

Brings up TMON's calculator. Any expression (almost) will be evaluated and displayed. For example, entering a trap name will return the trap number; entering a mathematical expression (or a number) will return the result in hex and decimal, etc. There are a million variations on this, non of which I have ever used, so if you have a question, get in touch with me for more info.

User / Cmd-u

This has a wealth of handy commands, but my desctiptions my descriptions will be limited to commands that I have used. There are three different screens associated with the User window: A000 trap functions, Control functions, and Memory functions. To switch pages, click on the line that reads Toggle Pages and press return until you arrive at the page you desire.

Control Functions:

Look for labels:	Unknown.
Label table	Unknown.
Label add/remove	Unknown.
Label file load	Unknown.
Registers	Unknown. Has something to do with TMON's internal registers.
Leave TMON: queue...	Similar to Exit, except that TMON will trap out all events and regain control when you click the mouse. When TMON regains control, the previously generated events will be available in the event queue. To activate this function, click on the Leave TMON... line and press Return.

Leave application... Use this function when your application system bombs. Using 0 as a parameter, attempts to quit to the active shell (usually the Finder), using 1 will attempt to re-launch the program. If you are in a program (not necessarily one you are cracking) and it system bombs, you will dive into the monitor. Use this function with a parameter of 0 and usually the app will quit to the Finder leaving any other open applications running normally.

Shut Down If the above does not gracefully exit to the Finder, you may need to use this function. The higher the number of the parameter you use, the safer your shutdown will be. If you have to resort to 0 (re-boot), you will have the long wait for boot up associated with turning the computer off then on.

Memory Functions:

Block Move Moves blocks of memory. Requires three parameters: source address, destination address, and length. Enter these three address after one another on the line and hit return.

Block Compare Compares blocks of memory using the same three parameters as the Block Move command. Any differences will be displayed as "Mismatch at xxxx/yyyy" where xxxx is the address of the source and yyyy is the address of the destination where the blocks do not match.

Fill Fills a block of memory with a specified value. Takes four parametes with the fourth being optional: beginning address, ending address, fill value, and optionally, the size of the fill value - 1 for byte fill, 2 for word fill, and 4 for long word fill.

Find Finds a specified value in a specified range of memory. Takes four parameters: search value, search value size (same as size from Fill command above), start address and end address. If any matches are found, they will be displayed between the curly braces.

Template Displays a memory location as if it were a Mac data structure, showing you all the current values. TMON currently knows only four data structures: WindowRecord, ControlRecord, TERecord, and ParamBlock (see IM for descriptions of these). Clicking on the Template line hitting Return will cycle through these four templates. Template takes one parameter, an address. So, after finding the structure you wish to display, enter an address that contains a structure of that type and hit return. TMON will list all current values for the fields of that structure. Note that information will be meaningless unless there is actually a structure of the desired type at the address you specify. This command could be helpful in looking at key disk checkers by allowing you to look at the ParamBlock the program is currently using to read the disk - although I have never used this.

Stack addresses	Attempts to recognize as labels the supplied address. This function defaults to an address of SP - stack pointer. To use this, just click to left of SP and hit return. The function will then look at the first address on the stack and see if it matches any labels that it currently knows. For example, right after a JSR, the stack contains the return address. If TMON k knows a label for the return address (and it will if there was a label to the left of the JSR in the assembly window) then using the Stack Addresses command will display the label in curly braces. Hitting return repeatedly will then move up the stack, analysing each successive stack address. Click in front of the SP and hit return to reset the command to the original stack pointer.
Stack crawl	Attempts to find the return address of a procedure that has a currently active stack frame. Remember that most compiled programs use the LINK and UNLK instructions to set up stack frames to temprarily store local variables. If you know what register is being used as the stack frame pointer (A6 is the only one I have ever seen and this is the default value TMON uses), then the Stack Crawl can use that register to analyse the stack and try to determine the return address and display it in the curly braces.
Load resource	Loades the resource specified by the two parameters (type and id #) into memory and displays the address of the resource in the curly braces.
Print	Allows you to print listings longer that contained in the active window. Clicking on the Print line and hitting return toggles the print mode between Dump, Assembly, File, and Heap. Once the mode has been selected, the print command needs a start and end address. Type these in, hit return, and TMON will print the desired output to the serial port (meaning that you cannot use a laserwriter, but you can use an imagewriter.)

A000 Trap Functions:

Trap record	Unknown. Allows you to record any traps called by the program, but requires a lot of complicated set up.
Record	Unknown. Used to allocate a table for trap record (above).
Trap	Unknown. Used by programmers to test the heap anytime a trap that affects the heap zone is called. We don't need this to crack.
Heap	Unknown. Similar to Trap but doesn't wait for a trap to execute. We don't need this one either.
Trap discipline	A programmer's feature. Trap discipline is a means of checking traps for faulty parameters. Select a range of traps and a PC range (see Trap Intercept below) and TMON will check all traps within that range. If it finds a trap with questionable

parameters, the monitor will be entered. There are two strengths of discipline: lenient and strict. To toggle these, click on the trap discipline line and hit return.

Trap checksum — Unknown. Another function that programmers would use to check application problems. Since we are cracking an application that already works, we don't need this one.

Checksum — Unknown. Used to specify the checksum for Trap Checksum.

Trap intercept — Allows you to specify a trap or range of traps that, when encountered, will cause the monitor to be entered. Simply click on the line and enter the trap name WITH a leading underscore, a space, and then the second trap. This specifies a range of traps to look for, the range being in numerical order of trap numbers. If only one trap is specified, only that trap will be checked. Use this to catch a program that uses a dialog to prompt for a serial number. If the trap entered is _ModalDialog, the monitor will be entered just before the dialog is drawn. Optionally, a PC range may be entered after the trap range. This would specify that TMON regain control only if the specified trap is encountered within the specified PC range. I have never used a PC range.

Trap signal — Similar to Trap Interrupt, except that once the trap range and optional PC range have been entered, the user must hit the interrupt switch to enter the monitor. Once the interrupt switch (or Programmer's Key) has been pressed, TMON will continue execution until a trap within the specified range has been encountered.

Options / Cmd-o

Allows setting of seven monitor global functions. I am not exactly clear what the various settings mean, so I jus leave them all on.

Print / Cmd-p

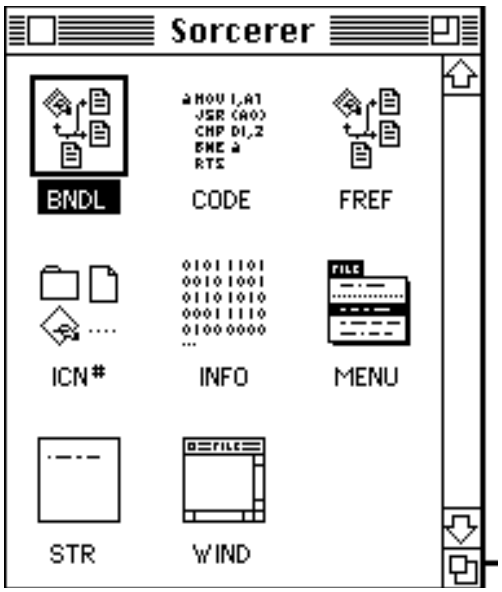
Causes the active window to be dumped to whatever port has been set during setup (achieved by launching the TMON application). This prints a window's contents only! To get long printouts, use the print command in the User Window.

How to crack Sorcerer

We are now going to look at a typical key-disk protection scheme. The important concepts to grasp here are how to quickly isolate the protection, and then how to remove it. Don't worry too much about the particulars, unless you happen to have a copy of Sorcerer you want to crack.

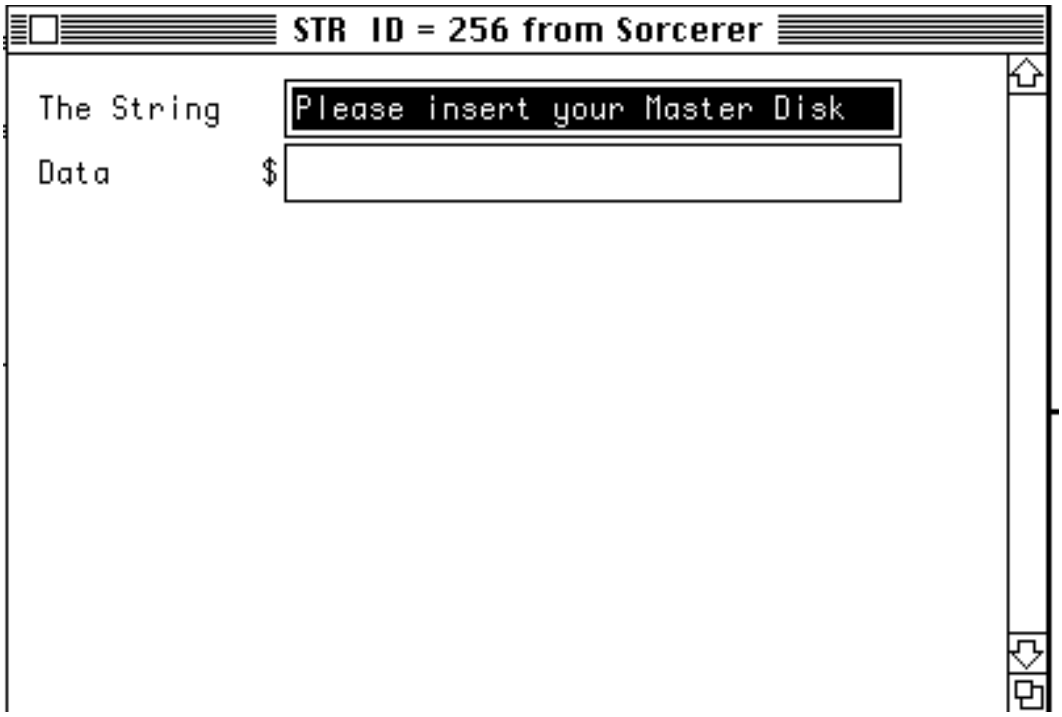
First off, how do we know that it is protected? Dumb question, but this is really important to beginning the crack. With Sorcerer, we note that when launched from the hard drive, it brings up a dialog box (or alert) requesting the key disk. So the logical place to start is with Resedit to try and figure out what resource the program is using to display the alert.

After you open the application in Resedit, we see the following:



Well, there are no ALRT or DLOG resources, so the program is generating its own dialog internally. If there was a set of ALRT or DLOG resources, we would quickly scan them and try to determine which was the one that the program displays to request the key-disk. If we could locate the ID # of the correct DLOG resource, we would go into Nosy and bring up the Traps ref map, see which procs called GetNewDialog, or Alert (if the resource in question was ALRT), and then check all the procs Nosy listed to see which one called GetNewDialog with the ID # we had found in Resedit.

Often you will find that there are DLOGs or ALRTs, but none of them have the correct message. If this is the case, then we would be in the same spot we are right now. The next thing to consider is that the string "Please Insert the Original Disk" (or whatever the string is) has to come from somewhere. You can try to locate it in Nosy, but often the string will be in a string resource. Look at the Resedit window above, and note the STR resource. Let's take a look:



Perfect! There is the culprit. So, all we have to do is find the part of Sorcerer that uses STR resource # 256. Since there are several ways to load a string, you might want to forget the Traps ref map and start tracing the program. If the program is huge, this might not be the way to go. If you look at the Traps ref map for Sorcerer, you would eventually find that proc108 calls the trap GetString. This would be an excellent place to start. Otherwise you might just find the proc called Sorcerer and start tracing there...An important note: tracing programs from start to error sucks. If you can figure out which trap is causing the problem, then by all means do so. If you are not familiar enough with the various traps, then you might well have to trace. Get a hold of IM and learn the Dialog Manager and the Resource Manager!

OK, let's start with the procedure Sorcerer:

```
D50:                                QUAL    Sorcerer ; b# =59  s#1  =proc38


D50: 4EBA 004C      1000D9E Sorcerer JSR     proc39
D54: 4E56 0000      'NV..'           LINK    A6,#0
D58: 2C5F           ',_'             POP.L   A6
D5A: 4E55 FCB6      'NU..'           LINK    A5,#-$34A
D5E: 9FED 0010          $10          SUBA.L  glob27(A5),A7
D62: 4EBA 0042      1000DA6          JSR     proc41
D66: 41ED FCB2        -$34E          LEA     glob2(A5),A0
D6A: 2F08           '/.'             PUSH.L  A0
D6C: 4EBA F292      1000000          JSR     proc1
D70: A8FE           '..'             _InitFonts
D72: 3F3C FFFF      '?<..'           PUSH    #$FFFF
D76: 4267           'Bg'             CLR     -(A7)
D78: 4EBA F49A      1000214          JSR     FlushEvents
D7C: A912           '..'             _InitWindows
D7E: A9CC           '..'             _TeInit
D80: 42A7           'B.'             CLR.L   -(A7)
D82: A97B           '.{'             _InitDialogs ; (resumeProc:ProcPtr)
D84: A850           '.P'             _InitCursor
D86: 4EAD 0092      20003F6          JSR     proc108(A5)
D8A: 4EBA 0390      100111C          JSR     proc54
D8E: 4EBA 0156      1000EE6          JSR     %_TERM
D92: 4E5D           'N]'             UNLK    A5
D94: 4EBA 000E      1000DA4          JSR     proc40
D98: 4E75           'Nu'             RTS


D9A: 4E5E                   data20   DC.B    'N^Nu'
```

A quick scan should reveal that possible problem areas are proc39, proc41, proc1, proc108, and proc54 since these are procedures that we can't see from this listing which is normal enough by itself. Luckily, if you were to look at the first three procs called, they are very short and very benign. If these were long, complex procedures, I might seriously consider going into TMON and setting a Trap Intercept to pick up _InitFonts so that TMON would grab control of the program early. Then when I launch Sorcerer, if TMON breaks in then the error is later in the program, but if Sorcerer bombs, then the error was before the InitFonts. That is a quick way to locate the problem.

So, let's take a look at the next procedure, proc108:

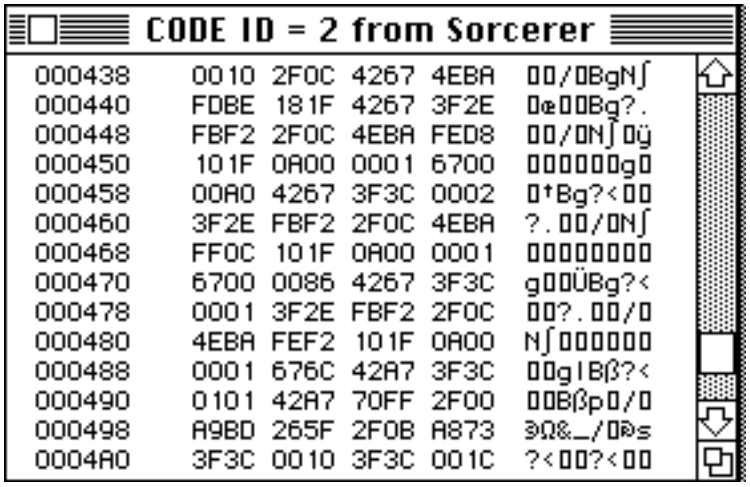
```
3F6:                                    QUAL    proc108 ; b# =194  s#2  =proc108

                                vem_1    VEQU  -1040
                                vem_2    VEQU  -1038
                                vem_3    VEQU  -1036
                                vem_4    VEQU  -1034
                                vem_5    VEQU  -1030
                                vem_6    VEQU  -774
                                vem_7    VEQU  -512
3F6:                                    VEND

                                ;-refs -  1/proc37    1/Sorcerer

3F6: 4A6F EBE6      'Jo..'  proc108  TST     -$141A(A7)
3FA: 4E56 FBE6      'NV..'           LINK    A6,#-$41A
3FE: 48E7 0F18      'H...'           MOVEM.L D4-D7/A3-A4,-(A7)
402: 41EE FE00      200FE00          LEA     vem_7(A6),A0
406: 2848           '(H'             MOVEA.L A0,A4
408: 486E FBFA      200FBFA          PEA     vem_5(A6)
40C: 486E FBF0      200FBF0          PEA     vem_1(A6)
410: 486E FBF6      200FBF6          PEA     vem_4(A6)
414: A9F5           '..'             _GetAppParms ; (VAR apName:Str255; VAR apRefNum:INTEGER;
                                                     VAR apParam:Handle)
416: 4267           'Bg'             CLR     -(A7)
418: 41EE FCFA      200FCFA          LEA     vem_6(A6),A0
41C: 2F08           '/.'             PUSH.L  A0
41E: 486E FBF2      200FBF2          PEA     vem_2(A6)
422: 4EAD 0052      1000162          JSR     GetVol(A5)
426: 3E1F           '>.'             POP     D7
428: 4267           'Bg'             CLR     -(A7)
42A: 486E FBFA      200FBFA          PEA     vem_5(A6)
42E: 3F2E FBF2      200FBF2          PUSH    vem_2(A6)
432: 3F3C 0010      '?<..'           PUSH    #16
436: 2F0C           '/.'             PUSH.L  A4
438: 4267           'Bg'             CLR     -(A7)
43A: 4EBA FDBE      20001FA          JSR     proc103
43E: 181F           '..'             POP.B   D4
440: 4267           'Bg'             CLR     -(A7)
442: 3F2E FBF2      200FBF2          PUSH    vem_2(A6)
446: 2F0C           '/.'             PUSH.L  A4
448: 4EBA FED8      2000322          JSR     proc105
44C: 101F           '..'             POP.B   D0
44E: 0A00 0001      '....'           EORI.B  #1,D0
452: 6700 00A0      20004F4          BEQ     lem_2
456: 4267           'Bg'             CLR     -(A7)
458: 3F3C 0002      '?<..'           PUSH    #2
45C: 3F2E FBF2      200FBF2          PUSH    vem_2(A6)
460: 2F0C           '/.'             PUSH.L  A4
462: 4EBA FF0C      2000370          JSR     proc106
466: 101F           '..'             POP.B   D0
468: 0A00 0001      '....'           EORI.B  #1,D0
46C: 6700 0086      20004F4          BEQ     lem_2
470: 4267           'Bg'             CLR     -(A7)
```

```
472: 3F3C 0001      '?<..'           PUSH    #1
476: 3F2E FBF2      200FBF2          PUSH    vem_2(A6)
47A: 2F0C           '/.'             PUSH.L  A4
47C: 4EBA FEF2      2000370          JSR     proc106
480: 101F           '..'             POP.B   D0
482: 0A00 0001      '....'           EORI.B  #1,D0
486: 676C           20004F4          BEQ.S   lem_2
488: 42A7           'B.'             CLR.L   -(A7)
48A: 3F3C 0101      '?<..'           PUSH    #257
48E: 42A7           'B.'             CLR.L   -(A7)
490: 70FF           'p.'             MOVEQ   #-1,D0
492: 2F00           '/.'             PUSH.L  D0
 494: A9BD              '..'                  _GetNewWindow ; (windowID:INTEGER; wStorage:Ptr;
                                                    behind:WindowPtr):WindowPtr
496: 265F           '&_'             POP.L   A3
498: 2F0B           '/.'             PUSH.L  A3
49A: A873           '.s'             _SetPort ; (port:GrafPtr)
49C: 3F3C 0010      '?<..'           PUSH    #16
4A0: 3F3C 001C      '?<..'           PUSH    #28
4A4: A893           '..'             _MoveTo ; (h,v:INTEGER)
4A6: 4267           'Bg'             CLR     -(A7)
4A8: A887           '..'             _TextFont ; (font:FontCode)
4AA: 42A7           'B.'             CLR.L   -(A7)
4AC: 3F3C 0100      '?<..'           PUSH    #256
4B0: A9BA           '..'             _GetString ; (stringID:INTEGER):StringHandle
4B2: 2C1F           ',.'             POP.L   D6
4B4: 2046           ' F'             MOVEA.L D6,A0
4B6: 2F10           '/.'             PUSH.L  (A0)
4B8: A884           '..'             _DrawString ; (s:Str255)
4BA: 4267           'Bg'             CLR     -(A7)
4BC: 42A7           'B.'             CLR.L   -(A7)
4BE: 3F3C 0001      '?<..'           PUSH    #1
4C2: 4EAD 002A      1000186          JSR     Eject(A5)
4C6: 3E1F           '>.'             POP     D7
4C8: 486E FBF4      200FBF4 lem_1    PEA     vem_3(A6)
4CC: 4EBA FEEE      20003BC          JSR     proc107
4D0: 4267           'Bg'             CLR     -(A7)
4D2: 3F2E FBF4      200FBF4          PUSH    vem_3(A6)
4D6: 2F0C           '/.'             PUSH.L  A4
4D8: 4EBA FE48      2000322          JSR     proc105
4DC: 1A1F           '..'             POP.B   D5
4DE: 4267           'Bg'             CLR     -(A7)
4E0: 42A7           'B.'             CLR.L   -(A7)
4E2: 3F2E FBF4      200FBF4          PUSH    vem_3(A6)
4E6: 4EAD 002A      1000186          JSR     Eject(A5)
4EA: 3E1F           '>.'             POP     D7
4EC: 1005           '..'             MOVE.B  D5,D0
4EE: 67D8           20004C8          BEQ     lem_1
4F0: 2F0B           '/.'             PUSH.L  A3
4F2: A914           '..'             _DisposWindow ; (theWindow:WindowPtr)
4F4: 4CDF 18F0      'L...'  lem_2    MOVEM.L (A7)+,D4-D7/A3-A4
4F8: 4E5E           'N^'             UNLK    A6
4FA: 4E75           'Nu'             RTS


4FC: '............'         data76   DC.W    $8100,8,0,$4FC,$FC00,0
```

The first thing to do here is to quickly scan for trap names. There are quite a few, but one should stick out. Remember that we are looking for some reference to STR #256. Note the GetString trap. Immediately before the trap is a PUSH #256...that's our guy! So, at this point, we know where the string is being loaded and drawn. Since this procedure is called from the Main procedure, we can bet that the key-disk check is also in this proc. Note that this is not always the case - often when you find the procedure that loads the dialog or string, you need to back trace to find out where the actual error generator is located. That is where the Refs line (right below the VEND) in the listing comes in handy. Note that this proc is called by not only Sorcerer, but also by proc37. This might mean that the program checks the key-disk later in its execution. But if you load up proc37, you would find that it simply Unloads the segment so it is harmless.

At this point, all we need to do is disable the disk-check. So, start scanning down the listing and ask yourself "Where is a branch that will skip over the GetString trap?". If you find that branch and make it always branch then odds are the program is cracked. Nosy will help out here. We are looking for a spot in the listing that a branch can jump to that will skip over the error. We have two choices in this listing: lem_1 and lem_2. Check out lem_1, and you will see a couple of problems with it. First of all, see what piece of code branches to it. There is a JSR Eject, then a test, and a BEQ lem_1. Also note that there is a DisposeWindow after it. We might guess that DisposeWindow is disposing the error dialog. We might also guess that lem_1 is being used as a loop to eject bad disks and request key-disks. Well, let's give lem_2 a shot. Now this one looks good - it is located right down at the procedures exit, so, if something is branching here, all the above stuff gets skipped.

So, just select lem_2 and hit cmd-f to let Nosy find all the references to lem_2 in the listing. Line 452 is the key. Note, D0 gets a result from an unknow procedure, then is EORd with 1, and then the branch occurs. It sure looks like changing that branch from BEQ to BRA would gurrantee that the error never occured. Let's try it. From the assembly instruction listing, we see that BEQ is 67, and BRA is 60. So, look at the first line in the above listing and we see that it is segment 2. So, open CODE resource 2 in Resedit, and skip down to address 456 (remember, take the Nosy address and add 4 to find the Resedit address).



There it is, on line 450. See the 6700? That sure matches what we find in Nosy, so that is our guy. Change the 67 to 60 by clicking to the right of the 67, hitting backspace or delete, and typing 60. That's it! Now quit Resedit and save changes. Launch the program and the protection is gone!

Let me quickly mention one last thing. The above crack involved looking for a branch that would skip over the problem area, and making damn sure that that branch always executed. But suppose that the program was setup so that after the disk check, the program branched *to* the error section. In this case, we would want to make sure the branch never executed. There are two ways to do this. First off, you can change the branch to its logical opposite - BCC to BCS, BNE to BEQ, etc. That way, the condition that triggers the error will now trigger the opposite, and run properly. The second method is to simply replace the trap with a NOP. That way, the branch never executes no matter what happens.

Look for upcoming material on more specific cracking methods and more actual cracks.

later - The Shepherd

Beta Notes: 10/17/91

The following bold entries constitute a tentative outline for topics to dicuss in detail. Some of these topics will require a fair amount of research on my part - in particular, the Eve and Encryption sections will take some time. After this section come the live cracks. These represent an attempt to take a novice cracker through every step of the cracking process detailing choices and decisions that I would make as I go and why I would make them.

Any feedback would be greatly appreciated - especially from any novice crackers who find parts of this document incomprehensible. Note that this is a rough draft - there are bound to be errors although hopefully no logical ones (just syntactical and/or spelling).

Determining where to start looking

1) Types of protection

    a) Serial number schemes

    b) Registration codes

    c) Network serial checks [AppleTalk driver stuff]

d) Hardware plugs - see below

e) Encryption - see below

f) Time stamps

g) Key disk

How to break into programs

1) Trap interrupts

a) Dialog/Alert traps

b) MenuSelect

c) InitFonts etc.

2) Manual entrance of TMON [Good luck]

3) Automatic TMON entrance via code modification [_Debugger trap insertion]

a) Determining an address with Nosy

b) Determining an address from the Jump Table

c)

Using TMON, Nosy, and ResEdit together

1) Determining address offsets

2) Nosy vs TMON

a) Why Nosy "feels better"

b) Why TMON is virtually omniscient

TMON Tricks

1) TMON tricks with register values, flags, and instruction modification

2) One step ModalDialog hassles [Serial number schemes]

3) TMON Pro shortcuts

Determining the type of crack to apply

1) Bypasses vs cracks

2) Finding the key code

3) Branch switching

    a) Mention something about branch op-codes - 2 and 4 byte instructions and offsets

4) Flag/variable modification

5) Code modification

Everything you always wanted to know about the CODE 0 Jump Table.

1) What it is and how it works

2) Locating an entry point

3) Modifications

Hardware plugs

1) General tips [Device Manager stuff]

2) Eve bullshit

Encrypted Code

Unless you are one hell of a genius at cryptology and have lots of time to kill, the encrypted CODE resources will have to be de-crypted and written back to the program. Here is why: to decrypt itself, a program will usually either take a known seed number and use it on each encrypted byte of the code or else it will start with some byte in the code and do a forward decrypt, i.e. the first byte decrypts the second byte, the new second byte decrypts the third byte, and so on. A simple method might be to have some code that looks like this:

```
                MOVE    #1000,D0
                LEA     encryptedshit,A0
                LEA     encryptedshit-1,A1
loop1           EOR.L   (A1)+,(A0)+
                DBRA    D0,loop1
encryptedshit   Here is where the encrypted gibberish begins.
```

This is a simple example, but note how it functions. D0 gets the number of longwords to decrypt, A0 is the destination (where the decrypted stuff will go - which is right back over the encrypted stuff) and A1 gets the decrypting key which is the long word that was previously decrypted. Then the code simply loops D0 times writing over the encrypted code with the decrypted code. After this code has finished, the program continues execution right where the encrypted (and now decrypted) code begins. Now cosider: somewhere in the encrypted stuff is the error check that you have to modify. This will be simple enough to locate assuming that you can run the decryption routine and then immediately regain control in TMON. The problem is that when you go to modify the error check so that it always passes, the modification screws up the decryption routine. This is because the decryption routine requires the exact original values to run properly since these values are the keys that the code uses. So a crack using traditional methods requires that you not

only change the error branch, but that you also change every other encrypted value such that the decryption routine still runs properly - no small feat!

A much more feasable method would be to decrypt the code, make the necessary modifications to the error routine, and then disable the decryption routine (just branching around it would do) and writing the whole mess (un-encrypted) back to the original code resource.

So much for the theory, now if I could just crack one of these suckers...

Live Cracks

MultiClip 2.0

This program uses a network checking algorithm to determine whether multiple copies with the same serial number are currently running - if you don't use this program on a network, you will never see the error.

Step 1: Where to start looking.

There are actually several good places to begin looking for the protection (especially if you have already cracked it - but I will assume that you have not). First of all, since the program scans the network, it is probably using the _Open Trap somewhere early in its code to to access the Appletalk driver. Second, it displays an error dialog (or alert) so we could open it up in Resedit, find the error dialog (and note its ID # for later use) and then Nosy it and look at procedures that call ModalDialog or one of the Alert traps to try and find the one that displays the dialog with the proper ID #. Third, we could have TMON trap either 1) ModalDialog if it is a dialog or 2) StopAlert, CautionAlert or NoteAlert if it is an Alert and begin tracing from that point backwords. Fourth, we could just Nosy it and start from the top (the slow way).

Whenever a program displays an error dialog (not a serial number dialog which seems to be in vogue these days) I almost always find the ID # of the dialog or alert and begin looking at procs in Nosy, so let's start there. In Resedit, we note that it is Dialog (and not Alert) #128 that is the problem. On to Nosy. After Nosy analyzes the INIT resource, open up the Trap Refs List under the Display menu and scroll down to GetNewDialog. Here you will find two listings: ASKNAME and PUTREGISTERDLOG. Since there are only two we can quickly check them both out (if there were a bunch, I would probably try a different method). First let us look at ASKNAME - here is the listing down to the GetNewDialog:

```
  42BA:                                 QUAL    ASKNAME ; b# =184  s#1  =proc54

                                vdu_1     VEQU  -26
                                vdu_2     VEQU  -18
                                vdu_3     VEQU  -12
                                vdu_4     VEQU  -10
                                vdu_5     VEQU  -8
                                param1    VEQU  8
                                funRslt   VEQU  12
   42BA:                                VEND

                             ;-refs - com_43    MYFILTERFORNAME

   42BA: 4E56 FFE6      'NV..'  ASKNAME  LINK    A6,#-$1A
   42BE: 48E7 0318      'H...'           MOVEM.L D6-D7/A3-A4,-(A7)
   42C2: 2C2E 0008      2000008          MOVE.L  param1(A6),D6
   42C6: 42A7           'B.'             CLR.L   -(A7)
   42C8: 4EBA E642      100290C          JSR     proc19
   42CC: 285F           '(_'             POP.L   A4
   42CE: 486E FFF8      200FFF8          PEA     vdu_5(A6)
   42D2: A874           '.t'             _GetPort ; (VAR port:GrafPtr)
   42D4: 42A7           'B.'             CLR.L   -(A7)
   42D6: 302C 001E      '0,..'           MOVE    30(A4),D0
   42DA: D07C 0014      '.|..'           ADD     #20,D0
   42DE: 3F00           '?.'             PUSH    D0
   42E0: 42A7           'B.'             CLR.L   -(A7)
   42E2: 70FF           'p.'             MOVEQ   #-1,D0
   42E4: 2F00           '/.'             PUSH.L  D0
     42E6:  A97C                  '.|'                      _GetNewDialog  ;  (DlgID:INTEGER;  wStorage:Ptr;
behind:WindowPtr):DialogPtr
```

The first thing to do is to locate the _GetNewDialog and determine its associated parameters: actually all we care about is the first parameter, the ID #. Tracing backwords, we see that -1 is the third parm, 0 is the second parm, and 30(A4) + #20 (from the ADD #20,D0) is the first parm. Well, we have a problem here. Instead of a nice plain ID # being passed to GetNewDialog, the ID # is hidden on the stack frame somewhere. At this point it is best to mark this proc as indeterminite and go on to the next one. If we must come back to this one then we will have to figure out if ID #128 is valid for this proc and go from there. So let us look at PUTREGISTERDLOG

```
 33AC:                                  QUAL    PUTREGISTERDLOG ; b# =141  s#1  =proc35
```

```
                            vdb_1     VEQU  -286
                            vdb_2     VEQU  -278
                            vdb_3     VEQU  -276
                            vdb_4     VEQU  -274
                            vdb_5     VEQU  -272
                            vdb_6     VEQU  -270
                            vdb_7     VEQU  -268
                            vdb_8     VEQU  -264
                            vdb_9     VEQU  -262
                            vdb_10    VEQU  -256
                            param1    VEQU  8
 33AC:                                VEND

                            ;-refs - INIT1

                            PUTREGISTERDLOG
 33AC: 4E56 FEE2      'NV..'          LINK    A6,#-$11E
 33B0: 2F0C           '/.'            PUSH.L  A4
 33B2: 206E 0008      2000008         MOVEA.L param1(A6),A0
 33B6: 43EE FF00      200FF00         LEA     vdb_10(A6),A1
 33BA: 703F           'p?'            MOVEQ   #63,D0
 33BC: 22D8           '".'    ldb_1   MOVE.L  (A0)+,(A1)+
 33BE: 51C8 FFFC      10033BC         DBRA    D0,ldb_1
 33C2: 42A7           'B.'            CLR.L   -(A7)
 33C4: 3F3C 0080      '?<..'          PUSH    #128
 33C8: 42A7           'B.'            CLR.L   -(A7)
 33CA: 70FF           'p.'            MOVEQ   #-1,D0
 33CC: 2F00           '/.'            PUSH.L  D0
  33CE:  A97C                  '.|'                        _GetNewDialog  ;  (DlgID:INTEGER;  wStorage:Ptr;
behind:WindowPtr):DialogPtr
```

Once again, find the GetNewDialog and determine the parms. Here we have -1 for the third, 0 for the second, and lo and behold, 128 for the first. This is definately our procedure. Note that this is an extremely easy example as no attempt has been made to disguise the ID # - it is clearly 128, the value we have been looking for all along.

Determining how to implement the crack.

The obvious place to start looking is just before the error dialog has been loaded. Here is that section of code from the above procedure:

```
        LINK    A6,#-$11E
        PUSH.L  A4
        MOVEA.L param1(A6),A0
        LEA     vdb_10(A6),A1
        MOVEQ   #63,D0
ldb 1   MOVE.L  (A0)+,(A1)+
        DBRA    D0,ldb_1          Next comes the code we just looked at
        CLR.L   -(A7)
        PUSH    #128
        CLR.L   -(A7)
        MOVEQ   #-1,D0
        PUSH.L  D0
        _GetNewDialog
```

As we look at this code, keep in mind what it is that we are looking for. We know that the program is capable of loading without this error, so somewhere it has to be checking the network and then either branching to the error code (if it detects a copy of itself) or else branching around the error code. So we need to find the branch that is causing this segment of code to execute. A quick scan of the code that precedes the error dialog code should reveal nothing of

interest. A Link followed by a 63 word Move Loop - no branches of any consequence whatsoever. If you are wondering why we can immediately eliminate the DBRA D0,ldb1 (after all, it is a branch) then ask yourself this: 1st, where does the branch go? Answer: to the line above the branch instruction. 2nd, what (if any) conditions is it checking? Answer: it checks to see if D0 (an obvious loop counter in this case) is equal to zero. If the branch does not either 1) branch directly to the error code (in this case it would have to be branching to the CLR.L -(A7)) or 2) branch around the error code (somewhere after the GetNewDialog and the ensuing ModalDialog and probably even an ensuing DisposeDialog) then the branch is almost certainly a bad candidate. You particulaly should be able to immediately eliminate loop terminator branches like the one above.

Well, since we have eliminated the only branch in this procedure above the GetNewDialog, we will have to look elsewhere. The next obvious place to look is in the procedure that called this one. Again Nosy makes this a snap. Take a look at the line right above the code listing that read refs - INIT1. The refs line tells you every procedure that calls the one you are currently looking at. Luckily, there is only one, so let us look at it next. Since this is a long procedure, I am only listing the section that surrounds the JSR PUTREGISTERDLOG line. I should also mention that I am writing this with a copy that I cracked a while ago and in un-cracking it for this document, could not remember exactly what the changed code was. I will show you where your code listing might differ from mine below:

```
196: 4268 0004      'Bh..'          CLR     4(A0)

19A: 4228 0006      'B(..'          CLR.B   6(A0)

19E: 4228 0007      'B(..'          CLR.B   7(A0)

1A2: 43FA 036E      1000512         LEA     data2,A1    ; len= 1

1A6: 45E8 0009      'E...'          LEA     9(A0),A2

1AA: 4EBA 0392      100053E         JSR     proc2

1AE: 43FA 03A2      1000552         LEA     data4,A1    ; 'Multi'

1B2: 4EBA 038A      100053E         JSR     proc2

1B6: 43FA 03AC      1000564         LEA     data7,A1    ; len= 2

1BA: 4EBA 0382      100053E         JSR     proc2

1BE: 4A6E FFEC      200FFEC         TST     vab_2(A6)

1C2: 6756           100021A         BEQ.S   lab_13

1C4: 4FEF FFFE      'O...'          LEA     -2(A7),A7

1C8: 2F2E FFEE      200FFEE         PUSH.L  vab_3(A6)

1CC: 4EBA 2C88      1002E56         JSR     proc29
```

```
1D0: 301F           '0.'            POP     D0
1D2: 6646           100021A         BNE.S   lab_13
1D4: 4FEF FFCE      'O...'          LEA     -50(A7),A7
1D8: 204F           ' O'            MOVEA.L A7,A0
1DA: 317C FFF6 0018 '1|....'        MOVE    #$FFF6,ioCRefNum(A0)
1E0: 216E FFEE 001E 200FFEE         MOVE.L  vab_3(A6),ioSEBlkPtr(A0)
1E6: 317C 00FC 001A '1|....'        MOVE    #252,CSCode(A0)
1EC: A004           '..'            _Control ; (A0|IOPB:ParamBlockRec):D0\OSErr
1EE: 4FEF 0032      'O..2'          LEA     50(A7),A7
1F2: 206E FFEE      200FFEE         MOVEA.L vab_3(A6),A0
1F6: A01F           '..'            _DisposPtr ; (A0/p:Ptr)
1F8: 486D FFFC           -4         PEA     glob1(A5)
1FC: A86E           '.n'            _InitGraf ; (globalPtr:Ptr)
1FE: A8FE           '..'            _InitFonts
200: A912           '..'            _InitWindows
202: A9CC           '..'            _TeInit
204: 42A7           'B.'            CLR.L   -(A7)
206: A97B           '.{'            _InitDialogs ; (resumeProc:ProcPtr)
208: A850           '.P'            _InitCursor
20A: 42B8 0A6C         $A6C         CLR.L   DeskHook
20E: 487A 0302      1000512         PEA     data2       ; len= 1
212: 4EBA 3198      10033AC         JSR     PUTREGISTERDLOG
216: 4EFA 0316      100052E         JMP     com_2
21A: 4227           'B''    lab_13  CLR.B   -(A7)
21C: A99B           '..'            _SetResLoad ; (AutoLoad:BOOLEAN)
21E: 42A7           'B.'            CLR.L   -(A7)
220: 2F3C 4452 5652 '/<DRVR'        PUSH.L  #'DRVR'
226: 487A 2156      100237E         PEA     data35      ; len= 12
22A: A9A1           '..'            _GetNamedResource ; (theType:ResType; name:Str255):Handle
22C: 1F3C 0001      '.<..'          PUSH.B  #1
```

```
230: A99B           '..'            _SetResLoad ; (AutoLoad:BOOLEAN)
```

First off, we need to find the line that calls the error procedure we just finished looking at. In this case the line will be either JSR PUTREGISTERDLOG or BSR PURREGISTERDLOG. We find the correct line just above lab 13. Now, quickly note the structure we are dealing with: we have JSR PUTREGISTERDLOG (which does all the error dialog stuff) followed by a JMP instruction. So the program is leaving the main flow of control after doing the error dialog. This is important because we can see that logically, there should be a branch that skips this piece of code and continues on with lab 13. If we scan backwords from the JSR PUT... we see a bunch of Initialization traps preceded by some Moves - but then notice this code:

```
JSR     proc2

TST     vab_2(A6)

BEQ.S   lab_13

LEA     -2(A7),A7

PUSH.L  vab_3(A6)

JSR     proc29

POP     D0

BNE.S   lab_13
```

Here is where I forget what the original code looked like so your listing might say BEQ.S lab 13 (for the second branch that is). Anyways, this code looks really good since it branches around the error section. At this point, we might hazard a guess and simply make these Branch instructions always execute by changing them to BRA lab 13. This might be an incorrect crack since the program could be making other checks above this code - we can eliminate this chance by continuing scanning upwards looking for references to lab 13 until the beginning of this procedure. What I would do in a case like this is make a real fast check of about 50 or so lines of code above this looking for branches refering to lab 13. If I find one, modify it...if not, then make the crack and test it. If the crack fails, then I would know to keep looking.

A quick note: The flow of the program seems to suggest that merely changing the first branch from BEQ to BRA would suffice since this instruction always executes (it is not branched around anywhere) and once this instruction branches to lab 13 there would be no need to change the second branch. However, I am writing this having already cracked this program and the method I used was to change the second branch only. Since I know that this works and cannot test any

other method (not having a network at my disposal), I will proceed in this manner. The would-be cracker could certainly try changing the first branch and it looks to me as if this would work.

So how is the crack applied? Well, in this case, it looks like the program branches to lab 13 only if the serial check is OK (i.e. there are no extra copies running on the network) so we need to to make this branch always execute. The easiest way to do this is to change the BNE.S lab 13 to BRA.S lab 13 - branch not equal turns into branch always. So, simply pop over to Resedit and open the proper resource (INIT in this case). To determine the ID of the resource, look at the top of the procedure window in Nosy. The first line will contain an s# followed by a number. This is the segment number or ID # of the resource (in this case it is obvious since there is only one INIT resource, but for CODE resources this is really handy). Once the resource is open (make sure you do not have the Resedit disassembler running - if you do, select Open Using Hex Editor from the Resource menu) scan down to the line that most closely matches the line you want to modify - in this case our line is 1D2 so find line 1D0 in Resedit and look over 2 bytes. There should be the code 6646. Just click in front of the 66, backspace to delete it, and type 60 (You can find these op-code numbers in the Cracker's Guide Part 1). Now quit and save changes and the crack is complete.

Infini-d 1.1

This program uses the common serial number / personalize dialog scheme.

Step 1: Where to start looking.

We have two good options here: 1) Find the Dialog ID # in Resedit and use Nosy's Trap Refs List or 2) trap ModalDialog in TMON and start tracing from there. I tend to use the second method, usually because I can implement the crack on the fly in TMON and actually run the program. Then I go back later and figure out how do a full crack with Nosy. Note that withe the second method we do not have to go through every stupid dialog in the program. Rather we can simply find the unfriendly ModalDialog and let TMON tell us which code resource we are in.

First, drop into TMON and set a trap intercept for _ModalDialog then exit TMON and launch Infini-D. TMON will proceed to stop execution at the first ModalDialog trap. Since it is possible for a program to have ModalDialog traps before the one that actually does the serial number stuff my first step is to immediately exit TMON and keep track of

how many ModalDialogs occur before the serial number dialog comes up. In this case it is the first ModalDialog, so I would have to then quit and start over, this time not exiting TMON when the trap occurs.

Once you are in TMON, open an Assembly window to (PC) to look at the code that is executing. I forget exactly, but essentially what you would see is the ModalDialog trap followed by a couple of meaningless instructions and an RTS. Since nothing happens after the ModalDialog, we would need to Step through the RTS to get back to the procedure that called this one.

I should make a quick note here: this technique of making an on the fly crack via TMON usually means that you are going to ruin the application, i.e. you are going to end up with a serialized program that no longer needs to be cracked. This is not a true crack, rather this is a bypass - once this is done, the program is personalized and ready to run; in a sense you are letting the program crack itself. If you wanted to make a true cracked copy, you would have to look at exactly which branches were modified in TMON and then go into Resedit and change the same instructions (with an un-serialized copy of the application).

OK, enough about that. Here is the code you would see:

```
PEA      $157A(A5)

MOVE.L  $000C(A6),-(A7)

_ModalDialog

UNLK     A6

RTS
```

Since the procedure ends right after the ModalDialog call, we need to step through the RTS to see what called this procedure...and here is that code:

```
001E50B4: LINK.W    A6,#$FFFE

001E50B8: PEA      `FFFE(A6)

001E50BC: CLR.L    -(A7)
```

```
001E50BE: JSR      $1572(A5)

001E50C2: ADDQ.L    #8,A7

001E50C4: CMPI.W    #$0001,`FFFE(A6)

001E50CA: BEQ.S    ^$001E50D8

001E50CC: CMPI.W   #$0002,`FFFE(A6)

001E50D2: BEQ.S    ^$001E50D8

001E50D4: MOVEQ    #$00,D0

001E50D6: BRA.S    ^$001E50DA

001E50D8: MOVEQ    #$01,D0

001E50DA: TST.W    D0

001E50DC: BEQ.S    ^$001E50B8

001E50DE: CMPI.W    #$0001,`FFFE(A6)

001E50E4: BNE.S    ^$001E50EA

001E50E6: MOVEQ    #$01,D0

001E50E8: BRA.S    ^$001E50EC

001E50EA: MOVEQ    #$00,D0

001E50EC: UNLK     A6

001E50EE: RTS
```

Well, there is quite a bit of comparing and branching going on here so we had better see if we can figure out what is happening. After the Link, the dialog handle is pushed on the stack, space for a return value (or maybe a parameter with value 0) is put on the stack and then the ModalDialog procedure is called. This is pretty standard. Next, the stack is restored to its original value and something is compared to 1, branch if so, then compare the same thing to 2 and branch if so. Notice an important thing here, namely that this procedure never calls GetDItem or GetIText nor does it call any more subroutines so this procedure cannot be the one that checks the serial number. So it is probably a safe bet that this procedure is testing to see what exactly the user did - hit OK? hit Cancel? Type in a keystroke? Assuming for the moment that this is the case, take a wild guess what the various dialog item numbers are? You guessed it...1 is the OK button, 2 is the Cancel button. Now look at the code and you can quickly see what is happening (still assuming our item number theory is correct). First, if the item number hit was one (OK button) then branch down, and put a 1 in D0. If the item number hit was 2 (Cancel button) then do the same thing. Otherwise put a zero in D0. Finally, TST D0 and if it was 0 (neither button hit) then loop back and call ModalDialog again. At this point the program knows one of the buttons was hit. So, if it was not the OK button, branch down and put 0 in D0 otherwise put a 1 in D0 (so that's Cancel

= 0, OK = 1). When we look at the procedure that called this one, we know that D0 will tell that procedure what happened (either OK or Cancel).

Note that this is one of those problem ModalDialog calls that exits everytime you hit a keystroke so you cannot just type in your name and serial number, hit OK to get back to TMON, and crack the sucker. Rather you have to either 1) settle for only typing in one letter before you crack it or 2) set a breakpoint just past the part were it tests for the OK button being hit, clear the ModalDialog trace, and exit - TMON won't interrupt until you hit the OK button and the breakpoint is encountered.

Finally, here is the last piece of code - the procedure that called the above procedure:

```
001E4FBE: ADDQ.L  #6,A7
001E4FC0:JSR     ^$001E50B4
001E4FC4: MOVE.W  D0,`FFFE(A6)          Here is where we returned from the above procedure. 1 = OK, 0 = Cancel
001E4FC8: CMPI.W  #$0001,`FFFE(A6)
001E4FCE:BNE.S   ^$001E5012             Branch if Cancel hit
001E4FD0:PEA     `FEF8(A6)
001E4FD4: MOVE.W  #$000A,-(A7)
001E4FD8:JSR     ^$001E4F58
001E4FDC: ADDQ.L  #6,A7
001E4FDE:PEA     `FEF8(A6)
001E4FE2:JSR     ^$001E52AC
001E4FE6: ADDQ.L  #4,A7
001E4FE8: MOVE.W  D0,`FFFC(A6)
001E4FEC:TST.W   `FFFC(A6)
001E4FF0:BNE.S   ^$001E5012
001E4FF2: MOVE.W  #$0001,-(A7)
001E4FF6:CLR.W   -(A7)
001E4FF8: MOVE.W     #$0034,-(A7)
```

```
001E4FFC:JSR     $107A(A5)

001E5000: ADDQ.L  #6,A7

001E5002: MOVE.L  582(A5),-(A7)

001E5006: MOVE.W  #$000A,-(A7)

001E500A:CLR.W   -(A7)

001E500C: MOVE.W  #$7FFF,-(A7)

001E5010: SelIText

001E5012: CMPI.W  #$0001,`FFFE(A6)        True if OK was hit

001E5018:BNE.S   ^$001E5020

001E501A:TST.W   `FFFC(A6)          Unknown: returned value from JSR above

001E501E: BEQ.S  ^$001E4FC0

001E5020: CMPI.W  #$0001,`FFFE(A6)

001E5026:BNE.S   ^$001E5070

001E5028:PEA     `FF38(A6)

001E502C: MOVE.W  #$0006,-(A7)

001E5030:JSR     ^$001E4F58

001E5034: ADDQ.L  #6,A7
```

Well, there is a lot of crap here and if you decided to trace the two JSRs you would be in for a long ride. The first thing to try is to deduce what will happen based on what we already know - we know that if the wrong serial number is entered, the program will go back to ModalDialog to let you change it. So we need to find a branch that goes back above line 1E4FC0 (the ModalDialog JSR). If we can find that branch and avoid it, we should be safe. So we will start tracing down from where the program returned, not making any assumptions yet, but looking at where the branches go. Right away you will note two JSRs. Take a look at the parameters passed, and you will note the pair of PEA FEF8(A6) instructions. So this same piece of information is being passed to both subroutines - nothing to write home about, but interesting. The real key you should notice here is that there is a TST and BNE after the second subroutine. This is the first chance the program has to make any decisions (although what decisions we don't know). Let's assume this branch does not execute (you could assume either way and wind up with the answer) i.e. FFFC(A6) = 0 - some stuff happens that we don't care too much about yet, some text is selected, and the button is tested. If it was OK, the return value from the second JSR is TSTed and if it was zero (which we are assumming), branch back to 1E4FC0 - back to the ModalDialog JSR. So this route is incorrect. Going back, we now need to assume that the branch at line 1E4FF0 did execute. This time, we jump right to the button check, skip the branch since OK was hit, and again TST the return value from the second JSR. Since the branch executed, this value cannot be zero, so execution proceeds. Looking down a few lines we note that there does not seem to be any more branches back to the ModalDialog JSR so we can tentatively assume that this is the end of the protection.

To apply the crack immediately, just make sure that branch executes. You can do this by typing BRA right over the BNE in TMON. If, however, you want to make a cracked, unserialized copy (which you can then serialize with anything you like) you need to figure out where code will be in Resedit and change that BNE to BRA. Unlike the listings I have pasted into this document, TMON will tell you exactly where the code is in the file. Refer to the above section on TMON MacNosy and Resedit for details, but essentially just find the Code Resource ID # and the offset from the TMON listing. Then Exit TMON and let Infini-d cancel out. Next open it the proper code resource in Resedit, scan down to the proper offset, and find the BNE (which is 66 in hex) and change it to BRA (60 in hex). Save changes and you are set.

FrameMaker 3.0

Serial number dialog scheme again. This one, however, presents a slight variation - Nosy won't disassemble it properly. This means that you will have to do all your cracking from within TMON.

Step 1: Where to start looking.

The only choice we have is to break in via TMON. The simplest way to do this is to drop into TMON, set a Trace Interrupt for ModalDialog and Exit. Now launch Framemaker 3.0 and wait for TMON to break in Here is the code you would see: (note that this listing is from TMON Pro - a TMON 2.8.x listing will be slightly different)

```
005B4F88:'CODE'®$0003ƒ$040C+$0284 PEA     $01AA(A5)

005B4F8C:'CODE'®$0003ƒ$040C+$0288 PEA     `FDEC(A6)

005B4F90:P 'CODE'®$0003ƒ$040C+$2… _ModalDialog

005B4F92:'CODE'®$0003ƒ$040C+$028E MOVE.W  `FDEC(A6),D0

005B4F96:'CODE'®$0003ƒ$040C+$0292 EXT.L   D0

005B4F98:'CODE'®$0003ƒ$040C+$0294 MOVEQ   #$01,D1

005B4F9A:'CODE'®$0003ƒ$040C+$0296 CMP.L   D0,D1

005B4F9C:'CODE'®$0003ƒ$040C+$0298 BNE     ^$005B50EA                    ;'CODE'®$0003ƒ$040C+$3E6

005B4FA0:* 'CODE'®$0003ƒ$040C+$2… CLR.W   `FDEC(A6)

005B4FA4:'CODE'®$0003ƒ$040C+$02A0 CLR.B   (A3)

005B4FA6:'CODE'®$0003ƒ$040C+$02A2 TST.L   `96FA(A5)

005B4FAA:'CODE'®$0003ƒ$040C+$02A6 BEQ.S   ^$005B4FC4                    ;'CODE'®$0003ƒ$040C+$2C0

005B4FAC:'CODE'®$0003ƒ$040C+$02A8 MOVE.L  A4,-(A7)

005B4FAE:'CODE'®$0003ƒ$040C+$02AA PEA     $3802                                        ;$000037D8+$2A

005B4FB2:'CODE'®$0003ƒ$040C+$02AE JSR     $1702(A5)

005B4FB6:'CODE'®$0003ƒ$040C+$02B2 MOVE.L  A3,-(A7)

005B4FB8:'CODE'®$0003ƒ$040C+$02B4 MOVE.L  A4,-(A7)

005B4FBA:'CODE'®$0003ƒ$040C+$02B6 JSR     $419A(A5)

005B4FBE:'CODE'®$0003ƒ$040C+$02BA LEA     $0010(A7),A7

005B4FC2:'CODE'®$0003ƒ$040C+$02BE BRA.S   ^$005B4FE8                    ;'CODE'®$0003ƒ$040C+$2E4

005B4FC4:'CODE'®$0003ƒ$040C+$02C0 MOVE.L  `FDE8(A6),-(A7)
```

```
005B4FC8:'CODE'®$0003ƒ$040C+$02C4 MOVEQ   #$05,D0
005B4FCA:'CODE'®$0003ƒ$040C+$02C6 MOVE.W  D0,-(A7)
005B4FCC:'CODE'®$0003ƒ$040C+$02C8 PEA     `FDEE(A6)
005B4FD0:'CODE'®$0003ƒ$040C+$02CC PEA     `FDF0(A6)
005B4FD4:'CODE'®$0003ƒ$040C+$02D0 PEA     `FDF4(A6)
005B4FD8:'CODE'®$0003ƒ$040C+$02D4 _GetDItem
005B4FDA:'CODE'®$0003ƒ$040C+$02D6 TST.L   `FDF0(A6)
005B4FDE:'CODE'®$0003ƒ$040C+$02DA BEQ.S   ^$005B4FE8                  ;'CODE'®$0003ƒ$040C+$2E4
005B4FE0:'CODE'®$0003ƒ$040C+$02DC MOVE.L  `FDF0(A6),-(A7)
005B4FE4:'CODE'®$0003ƒ$040C+$02E0 MOVE.L  A3,-(A7)
005B4FE6:'CODE'®$0003ƒ$040C+$02E2 _GetIText
005B4FE8:'CODE'®$0003ƒ$040C+$02E4 MOVE.L  A3,-(A7)
005B4FEA:'CODE'®$0003ƒ$040C+$02E6 JSR     ^$005B5670                  ;'CODE'®$0003ƒ$040C+$96C
005B4FEE:'CODE'®$0003ƒ$040C+$02EA TST.L   D0
005B4FF0:'CODE'®$0003ƒ$040C+$02EC ADDQ.L  #4,A7
005B4FF2:'CODE'®$0003ƒ$040C+$02EE BMI     ^$005B50C8                  ;'CODE'®$0003ƒ$040C+$3C4
005B4FF6:'CODE'®$0003ƒ$040C+$02F2 CMPI.L  #$00000005,D0
005B4FFC:'CODE'®$0003ƒ$040C+$02F8 BGT     ^$005B50C8                  ;'CODE'®$0003ƒ$040C+$3C4
005B5000:'CODE'®$0003ƒ$040C+$02FC ADD.L   D0,D0
005B5002:'CODE'®$0003ƒ$040C+$02FE MOVE.W  ^$005B500A(D0.L),D0                ;'CODE'®$0003ƒ$040C+$306
005B5006:'CODE'®$0003ƒ$040C+$0302 JMP     ^$005B5008(D0.W)            ;'CODE'®$0003ƒ$040C+$304
```

If you try to step through this and enter your name etc., you will find that ModalDialog is exiting after any keystroke. The way to get around this hassle is to get rid of the Trace Interrupt and set a breakpoint after the OK button is hit. How you ask? Well, take a look at the code that follows the ModalDialog. First, D0 gets the dialog item that was modified. Next D1 gets the value 1 and the two are compared. From Resedit, you can find the dialog item numbers for all the items and it turns out that item 1 is the OK button, and item 5 is the serial number - these are the two important ones since the program can't proceed until the OK button is hit (we don't care about the cancel button being hit) and then the program must check the serial number. Following the compare, we note that if they are not equal (i.e. OK button not hit) then it goes off somewhere. The next instruction must be the one that executes after the user hits the OK button. So set your breakpoint at the line that reads CLR.W FDEC(A6) which is at address 5B4FA0 (this will vary) - and in fact you can see the asterisk in the listing denoting that I have done just that. Now exit, enter your name and company and serial number (keep typing anything until the OK button lights up) and hit OK. Now TMON breaks in again at the breakpoint. Now we can begin the crack.

Determining how to implement the crack.

Before you continue, think about what the program must do at this point if it wants to validate your serial number (here it helps to have read Inside Mac on dialogs). First the program must obtain a pointer to the dialog item #5 (the serial number field) and then it must obtain a pointer to the text contained in that item. Knowing this, you can just scan down until you see a GetDItem trap followed closely by a GetIText trap. After this last trap, the program can do its validation. Here is that piece of code:

```
MOVE.L  A3,-(A7)
_GetIText
MOVE.L  A3,-(A7)
JSR      ^$005B5670
TST.L    D0
ADDQ.L  #4,A7
BMI      ^$005B50C8
CMPI.L  #$00000005,D0
BGT      ^$005B50C8
ADD.L    D0,D0
MOVE.W  ^$005B500A(D0.L),D0
JMP      ^$005B5008(D0.W)
```

We can note that A3 is the pointer that will point to the text after the trap. Once A3 has the text, a subroutine is called and D0 is tested. At this point, we cannot be sure whether the branch executes if the serial passed or failed, so we had better take a quick look at the code at address 5B50C8. I am not going to show it here, but that code does some crap then calles ParamText and then a Dialog call so it is probably safe to guess that the branch above jumps to the error code.

With this assumption in mind, what can we do about it? An initial guess would be to just make that BMI either not execute or even better, make the BMI branch down to the ADD.L D0,D0. Unfortunately, if you look at the last two lines, you can see that D0 not only determines whether the code branches to the error routine, but is then used for a JMP instruction so we had better take care of D0. Let's take a quick look at that JSR up a few lines that sets D0 in the first place and remember, we are trying to figure out what D0 should be set to. Also remember that the branch is a BMI meaning that the error occurs if the high bit of D0 is set.

```
004B1508:'CODE'®$0003ƒ$04C8+$096C LINK.W  A6,#$FF00
004B150C:'CODE'®$0003ƒ$04C8+$0970 MOVEM.L A3/A4,-(A7)
004B1510:'CODE'®$0003ƒ$04C8+$0974 LEA     `FF00(A6),A4
004B1514:'CODE'®$0003ƒ$04C8+$0978 MOVEA.L $0008(A6),A3
004B1518:'CODE'®$0003ƒ$04C8+$097C MOVEQ   #$00,D0
004B151A:'CODE'®$0003ƒ$04C8+$097E MOVE.B  (A3),D0
004B151C:'CODE'®$0003ƒ$04C8+$0980 MOVEQ   #$06,D1
```

```
004B151E:'CODE'®$0003ƒ$04C8+$0982 CMP.L   D0,D1
004B1520:'CODE'®$0003ƒ$04C8+$0984 BLE.S   ^$004B1526                  ;'CODE'®$0003ƒ$04C8+$98A
004B1522:'CODE'®$0003ƒ$04C8+$0986 MOVEQ   #`FF,D0
004B1524:'CODE'®$0003ƒ$04C8+$0988 BRA.S   ^$004B1592                  ;'CODE'®$0003ƒ$04C8+$9F6
004B1526:'CODE'®$0003ƒ$04C8+$098A MOVEQ   #$00,D0
004B1528:'CODE'®$0003ƒ$04C8+$098C MOVE.B  (A3),D0
004B152A:'CODE'®$0003ƒ$04C8+$098E MOVEQ   #$28,D1                                  ;'('
004B152C:'CODE'®$0003ƒ$04C8+$0990 CMP.L   D0,D1
004B152E:'CODE'®$0003ƒ$04C8+$0992 BGE.S   ^$004B1534                  ;'CODE'®$0003ƒ$04C8+$998
004B1530:'CODE'®$0003ƒ$04C8+$0994 MOVEQ   #`FF,D0
004B1532:'CODE'®$0003ƒ$04C8+$0996 BRA.S   ^$004B1592                  ;'CODE'®$0003ƒ$04C8+$9F6
004B1534:'CODE'®$0003ƒ$04C8+$0998 MOVE.L  A4,-(A7)
004B1536:'CODE'®$0003ƒ$04C8+$099A PEA     $3802                                    ;$000037D8+$2A
004B153A:'CODE'®$0003ƒ$04C8+$099E JSR     $1702(A5)
004B153E:'CODE'®$0003ƒ$04C8+$09A2 MOVE.L  A4,-(A7)
004B1540:'CODE'®$0003ƒ$04C8+$09A4 JSR     $0532(A5)
004B1544:'CODE'®$0003ƒ$04C8+$09A8 MOVE.L  A3,-(A7)
004B1546:'CODE'®$0003ƒ$04C8+$09AA MOVE.L  A4,-(A7)
004B1548:'CODE'®$0003ƒ$04C8+$09AC JSR     $0392(A5)
004B154C:'CODE'®$0003ƒ$04C8+$09B0 TST.L   D0
004B154E:'CODE'®$0003ƒ$04C8+$09B2 LEA     $0014(A7),A7
004B1552:'CODE'®$0003ƒ$04C8+$09B6 BEQ.S   ^$004B1558                  ;'CODE'®$0003ƒ$04C8+$9BC
004B1554:'CODE'®$0003ƒ$04C8+$09B8 MOVEQ   #$05,D0
004B1556:'CODE'®$0003ƒ$04C8+$09BA BRA.S   ^$004B1592                  ;'CODE'®$0003ƒ$04C8+$9F6
004B1558:'CODE'®$0003ƒ$04C8+$09BC MOVE.B  $0001(A3),D0
004B155C:'CODE'®$0003ƒ$04C8+$09C0 SUBI.B  #$30,D0
004B1560:'CODE'®$0003ƒ$04C8+$09C4 BCS.S   ^$004B1590                  ;'CODE'®$0003ƒ$04C8+$9F4
004B1562:'CODE'®$0003ƒ$04C8+$09C6 CMPI.B  #$02,D0
004B1566:'CODE'®$0003ƒ$04C8+$09CA BHI.S   ^$004B1590                  ;'CODE'®$0003ƒ$04C8+$9F4
004B1568:'CODE'®$0003ƒ$04C8+$09CC MOVEQ   #$00,D1
004B156A:'CODE'®$0003ƒ$04C8+$09CE MOVE.B  D0,D1
```

```
004B156C:'CODE'®$0003ƒ$04C8+$09D0 ADD.W   D1,D1

004B156E:'CODE'®$0003ƒ$04C8+$09D2 MOVE.W  ^$004B1576(D1.W),D1                ;'CODE'®$0003ƒ$04C8+$9D

004B1572:'CODE'®$0003ƒ$04C8+$09D6 JMP     ^$004B1574(D1.W)             ;'CODE'®$0003ƒ$04C8+$9D8
```

There are no traps here to quickly tell us what is happening, but we can quickly look at the lines that affect D0. Basically, there are a bunch of interspersed MOVEQ instructions putting various values into D0. One of the values is $FF which (since the high bit of $FF is set - in fact, all the bits of $FF are set) must trigger the error in the previous procedure. Other values include 5 and 0. Right now, that is enough information to proceed with the previous procedure - if we need more in depth info, we can always come back. So we have the following code again:

```
MOVE.L  A3,-(A7)

JSR      ^$005B5670

TST.L    D0

ADDQ.L  #4,A7

BMI      ^$005B50C8

CMPI.L  #$00000005,D0

BGT      ^$005B50C8

ADD.L    D0,D0

MOVE.W  ^$005B500A(D0.L),D0

JMP      ^$005B5008(D0.W)
```

Once again, we have an initial BMI which tells us that $FF won't work for D0. We also have BGT after comparing D0 with 5 which branches to the error - so D0 must be between 0 and 5 (the other values we noted from the subroutine above). At this point, I would (and did) simply try inserting values into D0. I started with 5 and the program went into Demo mode - strike one. Next I tried 1 and some other error occured. Finally, I tried 0 and the program continued flawlessly.

So you are asking, how exactly might you go about inserting these values into D0? Consider: once D0 is set to the proper value, the two branches become meaningless since they would not execute anyways (they only execute if there is an error). This little tidbit tells us that we can safely overwrite these instructions with anything we like. So we have several free bytes to put our own code into (don't panic yet - this is pretty straightforward) and all our code has to do is set D0 to 0 then proceed. One quick note: Never Never Ever modify code that affects the stack. If you do, you can easily cause system errors later on down the road. In the above code, this translates into not changing the ADDQ.L #4,A7 (A7 is the stack pointer, remember?). So what is the easiest way to put 0 into D0? Use a MOVEQ instruction. This is particularly nice because you probably do not know the machine hex code for instructions (like me). But that subroutine we looked at before is chalk full of MOVEQ instructions. If you look, a MOVEQ 0 #0,D0 translates into 70 00. So far so good except that the stupid BMI is one of those 4 byte branches. So we still have two bytes left that will be garbage since we just changed the first two. This is an excellent candidate for a NOP instruction - a two byte instruction that does absolutely nothing. The code for this (from the Cracker's Guide Part 1) is 4E 71.

So, open a dump window to the PC and find the BMI (I think it is 68 00 00 D4 or something like that). Change the four values to 70 00 4E 71 and now the program loads D0 with the correct value and proceeds as if nothing had happened. Now you have the crack, but you want to make a cracked / un-serialized copy right? So, unstuff a fresh copy of the application, open it in Resedit, and open the proper CODE resource. To find the ID #, look back at the TMON listing. It says CODE 0003 plus some benutia about the File reference number and then +nnnn where nnnn is the offset from the beginning of the Code resource. There is all you need. Open CODE ID 3 and jump down to line 2E8 (since 2EE is our byte) and change the 68 00 00 D4 to 70 00 4E 71. Now run it and enter anything you like for the serial number.

QuickFormat 7.01

[due to burn-out, the final sections have not been written up]

```
    33E:                                QUAL    CHECKFOR ; b# =508  s#3  =proc196

                            ;-refs -  3/INITPROG

    33E: 4E56 FFE4      'NV..'  CHECKFOR LINK    A6,#-$1C
    342: 48E7 0108      'H...'           MOVEM.L D7/A4,-(A7)
    346: 594F           'YO'             SUBQ    #4,A7
    348: 2F3C 6465 6D6F '/<demo'         PUSH.L  #'demo'
    34E: 3F3C 0080      '?<..'           PUSH    #128
    352: A81F           '..'             _Get1Resource ; (theType:ResType; ID:INTEGER):Handle
    354: 285F           '(_'             POP.L   A4
    356: 200C           ' .'             MOVE.L  A4,D0
    358: 6656           30003B0          BNE.S   lih_2
    35A: 594F           'YO'             SUBQ    #4,A7
    35C: 7004           'p.'             MOVEQ   #4,D0
    35E: 2F00           '/.'             PUSH.L  D0
    360: 4EAD 0082      10005EA          JSR     NewHandle(A5)
    364: 285F           '(_'             POP.L   A4
    366: 2F0C           '/.'             PUSH.L  A4
    368: 4EAD 0092      1000614          JSR     HLock(A5)
    36C: 2054           ' T'             MOVEA.L (A4),A0
    36E: 20BC 000F 423F ' ...B?'         MOVE.L  #$F423F,(A0)
    374: 2F0C           '/.'             PUSH.L  A4
    376: 2F3C 6465 6D6F '/<demo'         PUSH.L  #'demo'
    37C: 3F3C 0080      '?<..'           PUSH    #128
    380: 487A 007C      30003FE          PEA     data209     ; len= 2
    384: A9AB           '..'            _AddResource ; (theResource:Handle; theType:ResType; theID:INTEGER;
name:Str255)
    386: 554F           'UO'             SUBQ    #2,A7
    388: A9AF           '..'             _ResError ; :OSErr
    38A: 4A5F           'J_'             TST     (A7)+
    38C: 6714           30003A2          BEQ.S   lih_1
    38E: 3F3C 008B      '?<..'           PUSH    #139
    392: 1F3C 0001      '.<..'           PUSH.B  #1
    396: 4EAD 0462      2000B7C          JSR     DOSTANDA(A5)
    39A: 554F           'UO'             SUBQ    #2,A7
    39C: A9AF           '..'             _ResError ; :OSErr
    39E: 4EAD 0452      20009FE          JSR     DOERROR(A5)
    3A2: 2F0C           '/.'    lih_1    PUSH.L  A4
    3A4: A9AA           '..'             _ChangedResource ; (theResource:Handle)
    3A6: 2F0C           '/.'             PUSH.L  A4
    3A8: A9B0           '..'             _WriteResource ; (theResource:Handle)
    3AA: 2F0C           '/.'             PUSH.L  A4
    3AC: 4EAD 009A      100061E          JSR     HUnLock(A5)
    3B0: 2F0C           '/.'    lih_2    PUSH.L  A4
    3B2: 4EAD 0092      1000614          JSR     HLock(A5)
    3B6: 2E3C 176F 7C4E '.<.o|N'         MOVE.L  #$176F7C4E,D7
    3BC: 2054           ' T'             MOVEA.L (A4),A0
    3BE: BE90           '..'             CMP.L   (A0),D7
    3C0: 6606           30003C8          BNE.S   lih_3
    3C2: 422D FDE2        -$21E          CLR.B   glob73(A5)
    3C6: 6020           30003E8          BRA.S   lih_4
    3C8: 554F           'UO'    lih_3    SUBQ    #2,A7
    3CA: 2F07           '/.'             PUSH.L  D7
    3CC: 4EBA FE68      3000236          JSR     DODEMODI
    3D0: 1B5F FDE2        -$21E          POP.B   glob73(A5)
    3D4: 102D FDE2        -$21E          MOVE.B  glob73(A5),D0
```

```
    3D8: 5300           'S.'            SUBQ.B  #1,D0
    3DA: 670C           30003E8         BEQ.S   lih_4
    3DC: 2054           ' T'            MOVEA.L (A4),A0
    3DE: 2087           ' .'            MOVE.L  D7,(A0)
    3E0: 2F0C           '/.'            PUSH.L  A4
    3E2: A9AA           '..'            _ChangedResource ; (theResource:Handle)
    3E4: 2F0C           '/.'            PUSH.L  A4
    3E6: A9B0           '..'            _WriteResource ; (theResource:Handle)
    3E8: 2F0C           '/.'    lih_4   PUSH.L  A4
    3EA: 4EAD 009A      100061E         JSR     HUnLock(A5)
    3EE: 4CDF 1080      'L...'          MOVEM.L (A7)+,D7/A4
    3F2: 4E5E           'N^'            UNLK    A6
    3F4: 4E75           'Nu'            RTS
```

Finder 7 Menus

```
    458:                                QUAL    GETPASSW ; b# =31  s#1  =proc14

                              vap_1      VEQU  -288
                              vap_2      VEQU  -280
                              vap_3      VEQU  -276
                              vap_4      VEQU  -274
                              vap_5      VEQU  -272
    458:                                VEND

                            ;-refs - DOCOMMAN

    458: 4E56 FED8      'NV..'  GETPASSW LINK    A6,#-$128
    45C: 48E7 0018      'H...'          MOVEM.L A3-A4,-(A7)
    460: 4A2D FEFE        -$102         TST.B   glob59(A5)
    464: 670C           1000472         BEQ.S   lap_1
    466: 487A 01B4      100061C         PEA     data23      ; 'Password has already
    46A: 4EBA 139E      100180A         JSR     OUTPUTTE
    46E: 6000 00A2      1000512         BRA     lap_5
    472: 3F2D FEBA        -$146 lap_1   PUSH    glob28(A5)
    476: A998           '..'            _UseResFile ; (frefNum:RefNum)
    478: 594F           'YO'            SUBQ    #4,A7
    47A: 3F3C 0101      '?<..'          PUSH    #257
    47E: 42A7           'B.'            CLR.L   -(A7)
    480: 70FF           'p.'            MOVEQ   #-1,D0
    482: 2F00           '/.'            PUSH.L  D0
      484:  A97C                '.|'                      _GetNewDialog  ;  (DlgID:INTEGER;  wStorage:Ptr;
behind:WindowPtr):DialogPtr
    486: 285F           '(_'            POP.L   A4
    488: 2F0C           '/.'            PUSH.L  A4
    48A: 3F3C 0002      '?<..'          PUSH    #2
    48E: 486E FEEC      200FEEC         PEA     vap_3(A6)
    492: 486E FEE8      200FEE8         PEA     vap_2(A6)
    496: 486E FEE0      200FEE0         PEA     vap_1(A6)
     49A: A98D            '..'           GetDItem ; (dlg:DialogPtr; itemNo:INTEGER; VAR kind:INTEGER; VAR
item:Handle; VAR box:Rect)
    49C: 42A7           'B.'    lap_2   CLR.L   -(A7)
    49E: 486E FEEE      200FEEE         PEA     vap_4(A6)
    4A2: A991           '..'            _ModalDialog ; (filterProc:ProcPtr; VAR itemHit:INTEGER)
    4A4: 0C6E 0001 FEEE 200FEEE         CMPI    #1,vap_4(A6)
    4AA: 66F0           100049C         BNE     lap_2
    4AC: 2F2E FEE8      200FEE8         PUSH.L  vap_2(A6)
    4B0: 486E FEF0      200FEF0         PEA     vap_5(A6)
    4B4: A990           '..'            _GetIText ; (item:Handle; VAR text:Str255)
    4B6: 487A 0152      100060A         PEA     data22      ; 'cc5187efH28b911af'
    4BA: 486E FEF0      200FEF0         PEA     vap_5(A6)
```

```
    4BE: 4EBA FC4A      100010A          JSR     proc6
    4C2: 6642           1000506          BNE.S   lap_3
    4C4: 594F           'YO'             SUBQ    #4,A7
    4C6: 486E FEF0      200FEF0          PEA     vap_5(A6)
    4CA: A906           '..'             _NewString ; (theString:Str255):StringHandle
    4CC: 265F           '&_'             POP.L   A3
    4CE: 2F0B           '/.'             PUSH.L  A3
    4D0: 2F3C 5354 5220 '/<STR '         PUSH.L  #'STR '
    4D6: 3F3C 0080      '?<..'           PUSH    #128
    4DA: 487A 012C      1000608          PEA     data21      ; len= 2
     4DE: A9AB                '..'                _AddResource ; (theResource:Handle; theType:ResType;
theID:INTEGER; name:Str255)
    4E0: 3F2D FEBA        -$146          PUSH    glob28(A5)
    4E4: A999           '..'             _UpdateResFile ; (frefNum:RefNum)
    4E6: 1B7C 0001 FEFE   -$102          MOVE.B  #1,glob59(A5)
    4EC: 487A 00C0      10005AE          PEA     data20      ; 'Thanks for registeri
    4F0: 4EBA 1318      100180A          JSR     OUTPUTTE
    4F4: 4A2D FEFE        -$102          TST.B   glob59(A5)
    4F8: 6714           100050E          BEQ.S   lap_4
    4FA: 2F2D FEB0        -$150          PUSH.L  glob25(A5)
    4FE: 487A 0086      1000586          PEA     data19      ; ''Thank you for payin
    502: A91A           '..'             _SetWTitle ; (theWindow:WindowPtr; title:Str255)
    504: 6008           100050E          BRA.S   lap_4
    506: 487A 001A      1000522 lap_3    PEA     data18      ; 'For only $10, you ca
    50A: 4EBA 12FE      100180A          JSR     OUTPUTTE
    50E: 2F0C           '/.'    lap_4    PUSH.L  A4
    510: A982           '..'             _CloseDialog ; (dlg:DialogPtr)
    512: 4CDF 1800      'L...'  lap_5    MOVEM.L (A7)+,A3-A4
    516: 4E5E           'N^'             UNLK    A6
    518: 4E75           'Nu'             RTS
```

==========

If I swiped any Kracks or dialog from you, thanks for contributing.

==========