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:

```
MOVE100,D0;move 100 into D0
MOVE$5000,A1        ;move address $5000 into A1
MOVED0,(A1)         ;move D0 into address in A1
MOVED0,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.
MOVED0,100;put the number 10 into address 100.
MOVED0,101
MOVED0,102
```

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):

```
MOVE100,A0;put first address into A0.

While A0 not equal to 200 do the following:

    MOVE10,(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:

MOVED0,-(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

File Edit Display Reforma

New
Open... ⌘O
Close ⌘K
Save
Save As...
Revert
Append to Font/DA Mo.txt
Append To...
Delete
Page Setup
Print Window
Quit Nosy ⌘Q
to TTY mode ⌘Y

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.

Edit Display Reforma

Undo	⌘Z
Cut	⌘X
Copy	⌘C
Paste	⌘V
Clear	
Select All	⌘A
Find	⌘f
Change	⌘M
Goto Line	⌘J
Grab Clip & Find	⌘g
Show Insert pt	⌘I
insert pt to Top	⌘t
sel to Notes	⌘n

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.

Display Reformat M

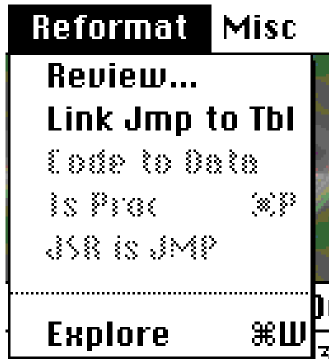
Code|Data blk ⌘d
Refs to ⌘R
Call chain

Sys syms map
Trap refs map
Globals map
Rsrc map
Strings

Data Blks
Case jumps
Mystery procs
ROM Patch Info
Bad Blks
Blk tbl ⌘B
Code Blks ⌘S

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).

Misc Search_Rsrc Tables

Extract comments ⌘E
Append to .aci
name cHange ⌘h
Addr to File pos
Convert to .asm fmt

save .snt, reRead .aci

✓Journal commands
Proc rel addresses
format Maps by Addr
cmds to Notes Wind
Set Source Path
Extract Map Names

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...

Tables Windows 10:30:

Record/ALL names	
Fields of	⌘
OS Traps	
TB Traps	
Sys Syms	⌘]
Sys Errs	
Constants	
Ascii	
Calculate	⌘[
Convert Hex to Dec	⌘-
add/ZAP Type Defs	

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.

```
Cmd:
<Hex|Dec|Asc|Zero><B|W|L>n_item, Wstr, Str, WZSTR, Zstr, Jumpc, JUMPP, BRA<L|C>
Undo, Code,  Quit, New{Byt} cnt, <NewLast|NUntil>{hhh}, NewWstr, New*, cOmbine
Mname=fmt1,fmt2., =mac_name, <H|D|A><Fld>cnt, ADDR, LADR<C|P>, DRVHD=pfx
                                -Data Blk-

   502: 'NV..A...Bn..Jx..'     data11   DC.W    $4E56,$FF86,$41EE,$FF86,$426E,$A,$4A78,$3F
   512: 'k.1n....B...Bh..'              DC.W    $6B1E,$316E,8,$16,$42A8,$12,$4268,$1A
   522: 'B...p..`f.=h. ..'              DC.W    $42A8,$1C,$7007,$A260,$6606,$3D68,$20,8
   532: '0... x.X.h.Ng. P'              DC.W    $302E,8,$2078,$358,$B068,$4E,$6708,$2050
   542: '".f.`. P".f. x.X'              DC.W    $2208,$66F4,$6010,$2050,$2208,$6604,$2078,
   552: '=h.N..N^ _TON.'                DC.W    $3D68,$4E,$A,$4E5E,$205F,$544F,$4ED0
```

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

```
Cmd: i
Undo, Is_proc, Quit, Revert_to_data, New... cr for next

                                   -Data Blk-

   502: 4E56 FF86       'NV..'  data11   LINK    A6,#-$7A
   506: 41EE FF86       'A...'           LEA     -122(A6),A0
   50A: 426E 000A       'Bn..'           CLR     10(A6)
   50E: 4A78 03F6          $3F6          TST     FsFcbLen
   512: 6B1E            1000532          BMI.S   lah_1
   514: 316E 0008 0016 '1n....'          MOVE    8(A6),ioVRefNum(A0)
   51A: 42A8 0012       'B...'           CLR.L   ioNamePtr(A0)
   51E: 4268 001A       'Bh..'           CLR     ioWDIndex(A0)
   522: 42A8 001C       'B...'           CLR.L   ioWDProcID(A0)
   526: 7007            'p.'             MOVEQ   #7,D0        ;Trap = GetWdInfo
   528: A260            '.`'             _HFsDispatch ,HFS  ; (A0|IOPB:WDPBRec):D0\OSErr
   52A: 6606            1000532          BNE.S   lah_1
   52C: 3D68 0020 0008 '=h. ..'          MOVE    ioWDVRefNum(A0),8(A6)
   532: 302E 0008       '0...'  lah_1    MOVE    8(A6),D0
```

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        -$BE0           PEA      glob3(A5)     Push the address of glob3 on the stack.
50AE: A86E             '.n'                _InitGraf ; (globalPtr:Ptr)  InitGraf, we see in IM, that
                                                              InitGraf must be called once near the start of
                                                              a program. It requires one parameter, a
                                                              pointer to the first QD global variable. This
                                                              parameter is first pushed on the stack in the
                                                              previous instruction.
50B0: A8FE           '..'              _InitFonts             These traps can all be found in IM.
50B2: A912           '..'              _InitWindows
50B4: 2F3C 0000 FFFF '/<....'          PUSH.L   #$FFFF
50BA: 201F           ' .'              POP.L    D0
```

50BC: A032	'.2'	_FlushEvents ; (whichMask,stopMask:EventMask)		
50BE: A9CC	'..'	_TeInit		
50C0: 42A7	'B.'	CLR.L	-(A7)	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.
50C2: A97B	'.{'	_InitDialogs ; (resumeProc:ProcPtr)		
50C4: A930	'.0'	_InitMenus		
50C6: 486E FFF4	200FFF4	PEA	vho_1(A6)	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).
50CA: A910	'..'	_GetWMgrPort ; (VAR wPort:GrafPtr)		
50CC: 2F2E FFF4	200FFF4	PUSH.L	vho_1(A6)	Now, our GrafPtr is used to set the current Port.
50D0: A873	'.s'	_SetPort ; (port:GrafPtr)		
50D2: 206E FFF4	200FFF4	MOVEA.L	vho_1(A6),A0	Now A0 contains our GrafPtr.
50D6: 4868 0008	'Hh..'	PEA	8(A0)	This instruction says to add 8 to A0,and push that address on the stack.
50DA: A87B	'.{'	_ClipRect ; (r:Rect)		
50DC: 2F3C 000E 000C	'/<....'	PUSH.L	#$E000C	The MoveTo trap requires two integers to be passes, but only one value is being pushed on the stack. Since the instruction says to push long, 4 bytes are being put on the stack, and an integer is only two bytes. Even though one instruction is being used, there are actually two parameters being passed to the MoveTo trap.
50E2: A893	'..'	_MoveTo ; (h,v:INTEGER)		
50E4: 3F3C 0029	'?<.)'	PUSH	#41	
50E8: 4EBA CE84	2001F6E	JSR	DRAWRESS	It turns out that DRAWRESS will draw the 41st string in the STR# resource. If you look in Resedit, you will see that this is "3.8" the version number.
50EC: 3F3C 000C	'?<..'	PUSH	#12	Note the lack of a size specifier. Remember that this means use the word (two bytes) size. Textsize needs an integer and IM tells us that an integer is two bytes - or one word.
50F0: A88A	'..'	_TextSize ; (size:INTEGER)		This is pretty easy - sets the fontsize to 12 point.
50F2: 422D F4EF	-$B11	CLR.B	glob25(A5)	Here is the .B size specifier, meaning clear only the low byte of glob25.
50F6: 42A7	'B.'	CLR.L	-(A7)	
50F8: 3F3C 0004	'?<..'	PUSH	#4	

50FC:	A9B9	'..'	_GetCursor ; (cursorID:INTEGER):CursHandle		

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...

Address	Code	ASCII	Label	Instruction	Operand	Comment
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	This pops the parameter passed,
D90:	3F02	'?.'		PUSH	D2	so that the selector parameter can be put ahead of it on the stack.
D92:	2F08	'/.'		PUSH.L	A0	Now the 2nd parm can be put back on the stack and the trap called.
D94:	ADED	'..'		_Pack6	AutoPop; (selector:INTEGER)	

Address	Code	ASCII	Instruction	Operand	Comment
513C:	285F	'(_'	POP.L	A4	proc 61 is returning a handle to the intl resource that it loaded, so save it in 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	A4 now has a handle to Pack #7.
5152:	2F0C	'/.'	PUSH.L	A4	
5154:	4EAD 00CA	1000AA6	JSR	HNoPurge(A5)	
5158:	4EAD 0172	1000D7C	JSR	proc59(A5)	This proc calles Pack2 with a selector of 2. This reads the Disk Initialization package into memory.
515C:	42A7	'B.'	CLR.L	-(A7)	Clear space on stack for a returned handle.
515E:	2F3A 00EE	200524E	PUSH.L	data259 ; 'ICON'	
5162:	4267	'Bg'	CLR	-(A7)	Push the integer 0.
5164:	A9A0	'..'	_GetResource ; (theType:ResType; ID:INTEGER):Handle		
5166:	285F	'(_'	POP.L	A4	A4 has a handle to Icon resource ID 0.
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	A4 has a handle to Icon resource ID 1.
5176:	4267	'Bg'	CLR	-(A7)	Make space for the returned RefNum.
5178:	A994	'..'	_CurResFile ; :RefNum		Note - no parameters passed.
517A:	3B5F FFE0	-$20	POP	glob58(A5)	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)
```

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

```
51C6: 426D F626          -$9DA            CLR      glob43(A5)
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)	

Address: Code	Ref	Label	Op	Operand	Comment
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
```

```
 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
```

Now do the same thing with the right list box.

```
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
```

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          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.
 102: 6700 FF06      100000A          BEQ      lab_1
 106: 4CDF 10C0      'L...'           MOVEM.L  (A7)+,D6-D7/A4      At this point, either an error
                                                             occurred or the user has hit the Quit button.
 10A: 4E5E           'N^'             UNLK     A6
 10C: 4E75           'Nu'             RTS

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

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)     Check the list box (it seems)
5DD6: 660C           2005DE4          BNE.S    lid_1          Look at what this branch is skipping.
5DD8: 7E01           '~.'             MOVEQ    #1,D7          The string is "Copy".
5DDA: 3F3C 0003      '?<..'           PUSH     #3             DIMITEM needs the item number to dim as
                                                             a parm.  Item 3 is the Remove button.
5DDE: 4EBA A308      20000E8          JSR      DIMITEM        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.
```

5DE2: 607A	2005E5E	BRA.S	lid_8	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	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.
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	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.
5DF4: 206D FECC	-$134 lid_3	MOVEA.L	glob50(A5),A0	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...
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	Look familiar? Let's guess that this is a vRefNum for the list box containing the selection.
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	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.
5E16: 4246	'BF'	CLR	D6	Otherewise, D6 is zeroed (no volume available).
5E18: 4A46	'JF' lid_4	TST	D6	

5E1A:	57C0	'W.'		SEQ	D0	D0=FF hex if there is no volume.
5E1C:	4A00	'J.'		TST.B	D0	
5E1E:	6616	2005E36		BNE.S	lid_5	This branch executes if D6 was zero and will cause 1 to moved into D7 - "Copy".
5E20:	2F00	'/.'		PUSH.L	D0	Save D0 on the stack (not a parameter)
5E22:	4267	'Bg'		CLR	-(A7)	Create space on the stack for the return value.
5E24:	3F06	'?.'		PUSH	D6	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)	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.
5E2A:	121F	'..'		POP.B	D1	Pop off the locked status.
5E2C:	201F	' .'		POP.L	D0	Pop off the original D0.
5E2E:	8001	'..'		OR.B	D1,D0	Or them so that, in effect, the AND instruction below will be ANDing both D0 and D1 with 1.
5E30:	0240 0001	'.@..'		ANDI	#1,D0	Check to see if one of the two contains a non-zero value,
5E34:	6702	2005E38		BEQ.S	lid_6	and if so, do not put a 1 in D7 (the string is not "Copy").
5E36:	7E01	'~.'	lid_5	MOVEQ	#1,D7	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.
5E38:	4267	'Bg'	lid_6	CLR	-(A7)	
5E3A:	206D FECC	-$134		MOVEA.L	glob50(A5),A0	And here is basically the same as above except that the other list box's volume is being checked
5E3E:	2050	' P'		MOVEA.L	(A0),A0	
5E40:	3F28 0058	'?(.X'		PUSH	88(A0)	Push the vRefNum of the volume from which the selection has been made.
5E44:	4EAD 0032	10004CA		JSR	proc6(A5)	Locked Volume?
5E48:	101F	'..'		POP.B	D0	
5E4A:	670A	2005E56		BEQ.S	lid_7	Go if not locked.
5E4C:	3F3C 0003	'?<..'		PUSH	#3	If the volume is locked, we cannot remove anything so dim the Remove Button.
5E50:	4EBA A296	20000E8		JSR	DIMITEM	
5E54:	6008	2005E5E		BRA.S	lid_8	
5E56:	3F3C 0003	'?<..'	lid_7	PUSH	#3	Else activate the Remove Button (volume is not locked).
5E5A:	4EBA A2AE	200010A		JSR	UNDIMITE	

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.

Address: Code	Data	Instruction	Operand	Comment
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 fairly 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

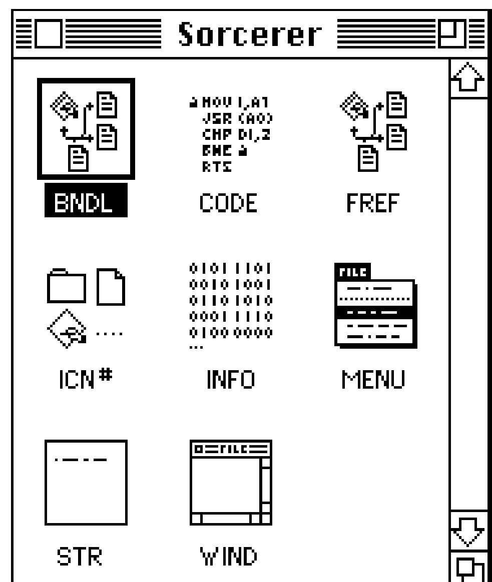
Causes the active window to be dumped to whatever port has been set during setup (achieved by launching the TMON application). This prints a window's contents only! To get long printouts, use the print command in the User Window.

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.

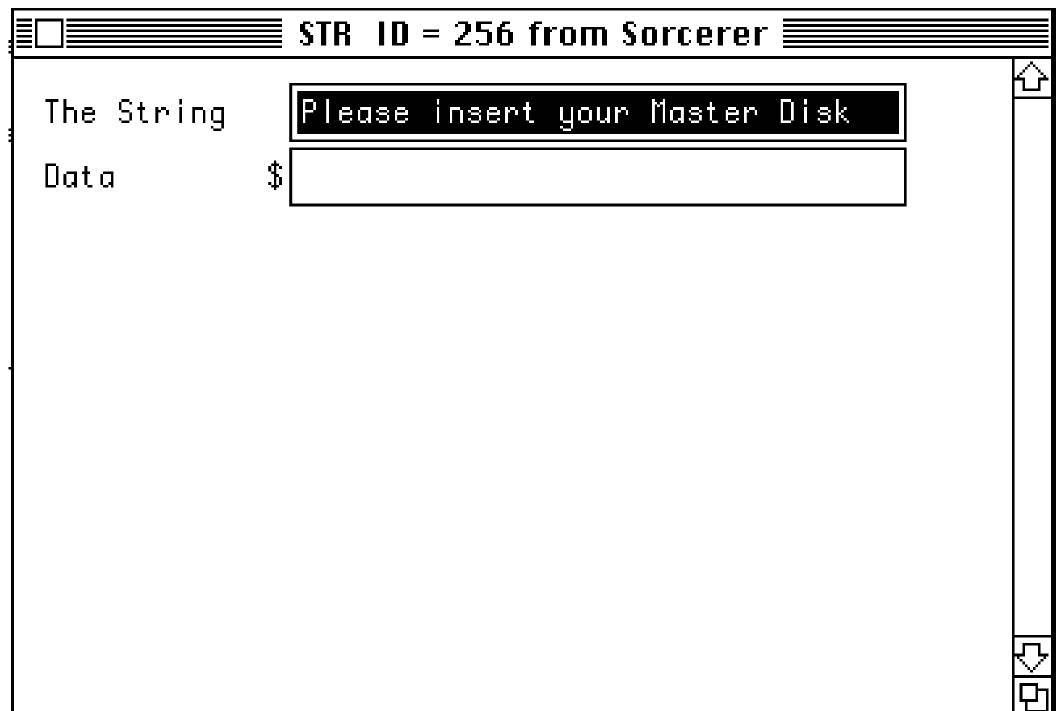
After you open the application in Resedit, we see the following:



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).

```
CODE ID = 2 from Sorcerer
000438    0010 2F0C 4267 4EBA
000440    FDBE 181F 4267 3F2E
000448    FBF2 2F0C 4EBA FED8
000450    101F 0A00 0001 6700
000458    00A0 4267 3F3C 0002
000460    3F2E FBF2 2F0C 4EBA
000468    FF0C 101F 0A00 0001
000470    6700 0086 4267 3F3C
000478    0001 3F2E FBF2 2F0C
000480    4EBA FEF2 101F 0A00
000488    0001 676C 42A7 3F3C
000490    0101 42A7 70FF 2F00
000498    A9BD 265F 2F0B A873
0004A0    3F3C 0010 3F3C 001C
```

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