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:

THE BASICS

Number Systems

We will be dealing with three different number systems. The difference between the number systems is simply at which number one decides to carry into the next column. In Decimal (the first system), we carry at the 10th number. That is, any given digit can only hold 10 values, namely, the numbers 0 - 9. Once we get to the carry value, we carry a one into the next column and reset the previous column to zero which is precisely what happens when you go from 9 to 10 (or 99 to 100 in which you carry twice, etc).
The second number system is called binary. In this system, the carry value is 2. This means that a given digit (called a bit in binary) can hold 2 values: 0 and 1. To add one to a number in binary, you use the same principle as in decimal, except that the carry is a different value. To add 1 to 8 in decimal, you just add 1 and there is no carry (because the ones column hasn't reached the carry value (10) yet). To add 1 to 9 in decimal, you have to carry the one to the next column (because you have passed the carry value) and reset the ones column to 0. So, counting in binary looks like this:
0
1
Add one to zero: we haven't reached the carry value (2) yet.
10
Add one to one: now we have 2 so we have to carry one to the next column and reset the first column.
11
Add one to zero (in the first column) and you just get one.
100
Add one to the first column and you get 2 so carry 1 to the second column and zero the first column. Add the carried one from the first column to the second column and you are adding 1 + 1 which is 2 - carry again. So, carry the one to the third column and zero the second column.
101
And so on...
110
111
1000
1001

1010
1011
1100
1101
1110
1111
And here we are at 15 decimal.

OK, we refer to binary because it is the native numbering system of the computer and also because in some of the instructions, the individual bits represent different information. Unfortunately, binary is hell for us humans. That brings us to the third major numbering which is hell for the computer AND hell for us! But both sides can deal so it's not too bad.
Hexadecimal is the third system and its carry value is, of all things, 16.
Now, we don't have 15 digits so hexadecimal uses the letters A-F for its last values. Here is how to count in hexadecimal;

Hex	Decimal	Binary
0	0	0
1	1	1
2	2	10
3	3	11
4	4	100
5	5	101
6	6	110
7	7	111
8	8	1000
9	9	1001
A	10	1010
B	11	1011
C		

12
1100
D
13
1101
E
14
1110
F
15
1111
10
16
10000

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

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

Signed Numbers and 2's Complement.

The basic binary system has no way of representing negative numbers. To accomodate this, we use what is called a sign bit. The sign bit is simply the leftmost bit we a talking about (meaning that often we have a 32 bit piece of data, but only care about 8 or 16 of the bits - so the sign bit is the 8th or 16th bit respectively), and is set to one for negative numbers. This means that if you want an 8 bit number to be negative, then it's eighth bit must be 1 (and 16th bit must be one for 16 bit numbers, etc.).

Two's Complement is an operation (yes there is an assembly instruction to perform it) that converts a positive integer to its negative equivalent (e.g. 1 to -1, 5 to -5, etc). To perform it, simply invert every bit in the number, then add a binary 1 to it. Take the number 00000001 (the eight bit integer 1). To make this -1, invert every bit (11111110) and add binary 1 to it -> 11111111. This then is -1 (or FF hex) as an eight bit integer. What happens if we want to treat this as a 16 bit integer? Big trouble, because now the sign bit is bit 16 and god only knows what is in bit 16. So, assembly has an instruction called Extend that extends a number out any number of binary places to make sure that any bits to the left of the original number don't affect its value.

All of this is relatively unimportant, since the assembly program you are trying to crack has already taken care of all these details and I have yet to see this type of information be critical to the cracking process. I simply wanted to get this out in the open so that you will have a better understanding of some of the instructions that will come up in the assembly instruction listings.

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

Basic Architecture and Addressing Schemes

CPU Registers

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

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

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

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

Addressing Schemes:

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

for example: MOVE 100,D1

moves the number 100 into the data register D1. You might be wondering whether 100 is binary, decimal, or hexidecimal. Well, right now we don't care, but as a general rule, we will assume that a number is decimal, unless it is prefixed by a dollar sign $. TMON and Nosy will be very explicit about telling you what type of number the command is using - but more on that when we talk about TMON and Nosy.

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

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

Examples:

```
MOVE
100,D1
;100 is in decimal
```

MOVE
D1,D2

MOVE
D0,100
;A little different here: since 100 is the receiving address (the second one) it will be treated as a memory address. So this instruction moves the contents of D0 into memory address 100.

MOVE
$55,D5
;$ indicates 55 is in hexadecimal

MOVE
$97BA54,A1
;moves the hex address 97BA54 into A1.
Remember here that the last two instructions are essentially the same. They both move some number into a register. However, the last instruction - since it moves the number into an address register - is setting up a pointer and a whole host of new instructions become available to it that are not available to the D registers.
Later we will note that there are several parameters that can be attached to the MOVE instruction (and many other instructions, for that matter). These will be covered later. This section is simply to show you how various kinds of information is manipulated. Note that in Direct Addressing, you see exactly what it is that is being moved: in the first example, you can see directly that the decimal number 100 is being moved into register D1. Any subsequent operations on D1 will involve the number 100.
Indirect Addressing: (extremely important)
This scheme involves moving some address into an address register and then operating not on the number in the address register, but rather on the address that is contained in the address register.
Example:
MOVE
100,(A1)
;moves the decimal number 100 into the address pointed to (or contained in) by A1.
Re-examine the last example of Direct Addressing. The command moved the number $97BA54 into address register A1. Since it is an address register, we can think of $97BA54 as an address rather than just a number. It may well be just a number, but odds are it will eventually be used as an address. The instruction above moves the decimal number 100 into the address $97BA54. It does not move the number 100 into address register A1. The parentheses mean that whatever is in A1 is actually an address and that this memory address will now contain the number 100.
Example:
MOVE
(A1),$1000

This instruction looks at the contents of A1, treats the contents as a memory address, and gets whatever is contained in that address and moves into hex address 1000.
Example:
MOVE
(A1),(A2)
This instruction looks at the contents of A1, grabs the contents of the

address it contains, and places this value into the address pointed to by A2. Lets look at a simple program and examine the memory that it deals with:

```
MOVE
100,D0
;move 100 into D0

MOVE
$5000,A1
;move address $5000 into A1

MOVE
D0,(A1)
;move D0 into address in A1

MOVE
D0,A1
;move D0 into register A1
```

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

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

```
MOVE
10,D0

;D0 now contains the fill number.

MOVE
D0,100
;put the number 10 into address 100.

MOVE
D0,101

MOVE
D0,102
```

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

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

While A0 not equal to 200 do the following:

MOVE
10,(A0)

Increment A0 to next address

End While Loop.
```

Note that this program is much simpler. Once the address register is set to the correct address, we can move the number 10 into this address then just increment the value in A0 which effectively makes A0 point to the next address. Note also that we could have MOVEd the number 10 into D0 and then inside the loop MOVEd D0,(A0) which would have had the same result but with one more instruction.

Auto Increment Addressing:

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

```
MOVE
D0,-(A1)
;decrement A1 to the previous address and put the contents of D0 into this new address.

MOVE
D0,(A0)+
;move D0 into address pointed to by A0 and then increment A0 to point to the next address.

MOVE
(A0)+,(A1)+
;move the contents of memory pointed to by A0 into the memory address pointed to by A1 and then increment both registers.
```

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

```
MOVE
100,A0
```

```
While A0 not equal to 200 do:

MOVE
10,(A0)+
;fill the address and increment to next address.

end while loop.
```

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

```
MOVE
$A000,A0
;put source address in A0.

MOVE
$B000,A1
;put destination into A1.

MOVE
(A0)+,(A1)+
;move one piece of password to destination and increment both pointers.

MOVE
(A0)+,(A1)+
;move next piece of password to destination.
```

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

This about wraps up addressing schemes and register introduction. Next I want to look at one instruction - MOVE - and consider all the parameters one might use with it.
The first thing to consider is that there are several types of MOVE instruction. There is the basic MOVE that we have used up until now. This is used to move data around.
MOVEA is used to move addresses. Example:

```
MOVEA
$5000,A0.
```

Yes - we should have been using this in the above examples when moving addresses into address registers, but I wanted to show addressing types, not instruction types. The Move Address is used just like the Move command, but lets you know that it is an address that is being moved (which means simply that the destination is an Address register).
MOVEQ
Move Quick: A shortcut instruction that moves an eight bit signed integer into a data register.
Two things to note: 1) a eight bit integer translates to -128 to +127 in

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

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

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

Example:

```
MOVEM
D4-D7/A0-A5,$5000.
```

Moves data registers D4,D5,D6 and D7, and address registers A0,A1,A2,A3,A4, and A5 into memory starting at $5000. This command is used primarily at the start and end of subroutines to save the contents of registers. Note that by reversing the arguments (so that $5000 comes first), the registers are restored to their original values which were saved in the above instruction.
There are a couple of other forms of the MOVE instruction, but they are rare and unimportant for cracking. If you see one, you should be able to figure out what it is doing. Now, we look at modifying the operands of the MOVE instruction.
Up until now, we have worked under the assumption that registers (and memory) contain 32 bits of information. This is not quite true. First of all, a memory address can hold 8 bits of information. Luckily, the Mac is smart enough to know that if we are moving a 32 bit register into memory, it needs to use 4 consecutive memory addresses. Secondly, we aren't limited to just 32 bit instructions. Consider:

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

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

to move the low 16 bits (I'll illustrate low bits in a second) into the address pointed to by A0 and the address following A0. The last instruction moves the low 8 bits of D0 into just the address pointed to by A0.
OK: here is what all that really means. Consider:

Instruction
Memory Address Contents->
$5000
$5001
$5002
$5003

MOVE
$5000,A0

??
??
??
??

MOVE
$12345678,D0

??
??
??
??

MOVE.B
D0,(A0)

$78
??
??
??

MOVE.W
D0,(A0)

$56
$78
??
??

MOVE.L
D0,(A0)

$12
$34
$56
$78
Question marks indicate that the instruction did not affect that memory address. Note that 1) when the information to be moved is longer than 8 bits

it is automatically moved into successive memory addresses, and 2) the information is stored from most significant to least significant. The terms most and least significant (or high and low) are used to designate the higher vs lower portions of the number. In the number $1FF hex, the most significant byte is 01 and the least significant byte is FF. In the number $12345678, the MSB (most significant byte) is 12 and the LSB is 78. In that same number, the most significant word (2 bytes) is 1234 and the least significant word is 5678. I will often make references to both most/least significant bytes and most/least significant bits.
One last thing before we move on is to note that when using the auto increment/decrment addressing modes, the amount of increment or decrement is dependent upon the size of the data being moved (which makes sense). If you say MOVE.W D0,(A0)+ then A0 will be incremented 2 bytes so that it then points one address past the data just moved into it. Likewise, if the instruction was MOVE.L D0,(A0)+, then A0 would be incremented by 4 bytes and would again point one address past the data just moved.
Also, often the size identifier is left off the instruction (like in MOVE D0,D2). When this is the case, it means the instruction is using a word size operand or MOVE.W. If the instruction is referring to byte or long-word size operands, it will explicitly say so in the command - MOVE.B or MOVE.L.
Special Registers:
Program Counter, denoted PC. This register always points to the instruction to be executed. You won't usually care what is contained in the PC, but you will want to do your assembly listings from wherever it currently is. TMON makes it very simple to start dis-assembling from the current PC so that you can see on-screen the instructions that are going to be executed.
The Status Register: very important.
This guy is how the processor keeps track of what just happened. For example, anytime you compare two values, you need to know if they were equal, not equal, one was bigger, etc. All this type of information is contained in the Status register. Basically, the status register is a 16 bit register in which certain bits contain information that you will want to access. Don't worry about which bits mean what because assembly language has operators that refer to the bits with nice, easy to remember mnemonics. Here are the bits that you will care about:
Z
the zero flag. This flag is set if the result of an operation is zero, or if two compared values are the same - it is cleared otherwise. For example, ADD.B $FF,1 would result in the number $100. But since we specified a byte size operation, the byte result is 0 and the flag would be set.
C
the carry flag. This contains the carry from an arithmetic operation. If you add two 8 bit (.B) numbers, the carry flag contains the 9th bit. Say you add $FF and 1 again. The result is a byte valye of 0 with a carry into the next bit. This carry would show up in the c flag. This bit also receives bits that are shifted out of a number during shift or rotate instructions. (See commands list).
N
the negative flag. Set if the high bit (meaning the 8th bit when using the .B specifier, the 16th bit for the .W, etc) of an operation gets set. Also gets set if the result of an operation is negative.
V
the overflow flag. Set whenever an operation yields a result that cannot be properly represented. For example, when adding the bytes 7F and 01, the result - 80 - cannot be represented in 8 bits. In eight bits, the eighth bit is the sign bit (telling whether the number is posative or negative). Note that this only happens if you are adding bytes - if the command added words,

then the result CAN be represented in 16 bits. This flag won't be used too much.
X
the extended flag. This is basically a copy of the carry bit, but not all operations affect it. The X flag is used to enable multi-precision instructions, that is, instructions can be intermixed without always affecting the X flag (in this case , the multi-precision carry bit). Once again, not used to much.
This probably doesn't make too much sense. That's OK, because you will get the hang of it when we look at a batch of code listings. The only reason I am listing them here is because TMON can display these flags and their current values. This allows you to predict where the program is going when it decides to branch somewhere. These flags are used to control program flow and, as such, are the single most important element to cracking. This is how you tell a program that the password you just typed was equal (and not unequal) to the password the program is looking for. We will look at the branch instructions later on. These instructions almost all use the Status Register Flags.
The final special register is actually just the A7 address register. The reason it is special is because it is used as the stack pointer on the Mac. The Stack is basically a chunk of memory that is used for special situations such as jumping to a subroutine and having to remember where the program jumped from so it can return when the subroutine is finished. The stack is also an excellent way to pass values to a subroutine. This will be illustrated later. All you need to understand is that the Stack is a piece of memory and can be manipulated as such. To refer to the stack, refer to the A7 register. Also, the stack moves backwards as it is used. Therefore, when a program wants to put a number on the stack it uses the pre-decrement indirect addressing mode:

```
MOVE
D0,-(A7)
;puts the value in D0 onto the stack and moves the stack pointer back one address.
```

```
MOVE
(A7)+,D0
;puts the value on the stack into D0 and increments the stack pointer to the next stack value.
```

And of course, to get the value back off the stack, you would use Post-Increment. These are not always used, but when they are used, it moves the stack pointer to the next available piece of stack space. When we begin working with Traps, you get a good workout with the stack so don't worry if this doesn't make complete (or any) sense yet.
Traps
Traps are a quick and easy method of accessing the 9 jillion built-in subroutines found in the Macintosh ROM. Traps do everything under the sun and are probably the main reason that all the Mac programs look alike. When a program wants do anything from drawing text to bringing up dialog boxes to putting up menus, traps are used. Why not just call the subroutines directly? Well, the problem is that every time Apple comes out with new system software, they change the addresses of one or more of these subroutines that almost all programs need. This would create chaos for applications, so Apple uses the idea of a trap table. The trap table is a means of associating the trap name (actually it's machine language code) with the proper address of the subroutine. So, no matter what the system version (within reason), an application can use the trap table to correctly call the subroutine it wants.

These traps are easy to spot: they all start with an underscore and then the name of the trap, e.g. _GetNewDialog.
A quick note about traps and viruses / anti-virus programs. If you were ever wondering how a virus program works, consider that a virus needs to be able to write portions of itself onto a disk. To do this, it needs to have access to an operating system that can do the actually writing. It could either pack an operating system around with itself (unwieldy and difficult to change when apple modifies the system) or use the trap table to call the traps that write resources. Now, the trap table can be patched by a program...i.e. a programmer can substitute his own subroutine into the trap table so that any program that calls the trap to do something, actually calls the new subroutine. Knowing this, an application could be written that patches the trap table and monitors the activity of any trap that writes resources. (I haven't de-compiled the newer virus programs, but I know that's how vaccine worked). The anti-viral program then just sits back and intercepts any of these traps, takes a look to see just what it is that is being written and where. If it looks suspicious (like writing an nVIR resource to the system!) then it lets you know. WDEF was a really great virus because the programmer figured a way to bypass this method. The first thing WDEF does is try to determine exactly which system it is operating under, and, if it is one that it recognizes (the 6.0x series I believe) it will re-patch the trap table with the original system values so that it can write to the disk without being monitored! The key is that this only works if WDEF knows the original values of the trap table and, since they often change, this means that WDEF is only effective on certain system versions. (Note that if it cannot re-patch the trap table, it will attempt to run and hope that there is no anti-virus program running).
Well, back to assembly. Almost every trap needs some parameters to operate. For example, GetNewDialog needs several parameters, including the ID # of the dialog to load, and several other things; and it returns a pointer to the dialog. Here is where the stack becomes important. Most traps use the stack to pass parameters and return values. Consider the following code (which will probably be incomprehensible)

```
CLR
-(A7)
;put 0 (word length since there is no size specifier ) on the stack

MOVE
$2FF,-(A7)
;Put $2FF (word size again) on the stack.

CLR.L
-(A7)
;put a nil-pointer on stack

_StopAlert


MOVE
(A7)+,D0
```

This little subprogram brings up an alert dialog. If we were to look in Inside Mac vol 1under StopAlert we would find that it requires 2 arguments and

returns one result. If we were programming, we would care what types of information these parameters are (integer, pointer, etc.) but since we are cracking, we can assume that the program to be cracked has already figured all this out.
Anytime a trap returns a value the calling code must allocate space on the stack before it puts the parameters on the stack. That is precisely what the CLR instruction does. (CLR or clear, puts zero into its operand so CLR.L   D0 would put 32 zero bits in D0) This is a fast way to move the stack pointer back one byte...we don't actually care what get puts on the stack (zero in this case) because the trap is going to replace that number with its return result. Since Inside Mac says StopAlert returns an integer and that an integer is 2 bytes or 1 word, we first clear an integer's worth of space on the stack.
Next we start putting arguments on the stack in the same order as Inside Mac says. The first thing is the alertID which is an integer. This is simply the number of the alert - i.e. the number you would see if you looked at the alerts in Resedit. So, this number ($2FF in my example) is moved onto the stack. The second argument is filterproc and is a procpointer (nothing more than a pointer). This argument is used only if the built-in dialog handlers don't quite cut it for you're application (maybe you have special command keys to watch for or something). If this is the case, you would pass a pointer to you're filtering procedure in this argument. Since I don't care about this, I will pass a nil pointer (one that points to nothing - this is defined as $00000000 [a long word] in assembly).
Once I have put the proper information on the stack, I can