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 decompil