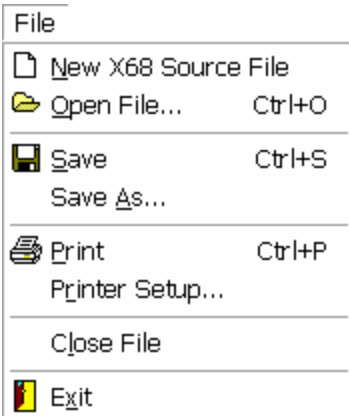


File menu:



New X68 Source File - Creates a new source window using the predefined source template.

Open File... - Loads an existing 68000 source file into the editor.

Save - Save the currently selected source file. Defaults to Save As if the file is being saved for the first time.

Save As... - Save the currently selected source file with the option to change the file name prior to saving. The default extension for 68K Editor is .x68 and it is recommended that all source files be saved with this extension.







Print - Prints the currently selected source file.

Printer Setup... - Setup the printer.

Close File - Closes the currently selected source file.

Exit - Closes all open files and exits the program.

Edit menu:

Edit	
 Undo	Ctrl+Z
 Redo	Ctrl+Alt+Z
<hr/>	
 Cut	Ctrl+X
 Copy	Ctrl+C
 Paste	Ctrl+V
Select All	Ctrl+A
<hr/>	
+* Comment Selection	Ctrl+Alt+C
-* Uncomment Selection	Ctrl+Alt+U
<hr/>	
 Find...	Ctrl+F
Find Next	F3
Find And Replace...	Ctrl+R

Undo - Undo the last edit operation.

Redo - Redo the last undo operation.

Cut - Removes the selected text and places it in the Window's Clipboard.

Copy - Copies the selected text to the Window's Clipboard.

Paste - Inserts the contents of the Window's Clipboard into the document at the present cursor position.

Select All - Selects all text in the current document.

Comment Selection - Inserts an asterisk '*' at the start of each selected line making it a comment.

Uncomment Selection - Removes one asterisk '*' from the start of each selected line.

Find... - Opens a dialog where a search phrase may be entered. The first occurrence of the phrase in the document is highlighted.

Find Next - Searches the document for the next occurrence of the search phrase.

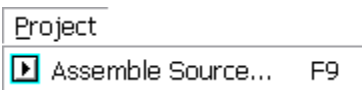
Find And Replace... - Opens a dialog where a search phrase and replacement phrase may be entered. The search phrase is found and may be replaced if desired.

Two additional commands are available by keyboard shortcut only:

Ctrl+> - Inserts a space at the start of each selected line.

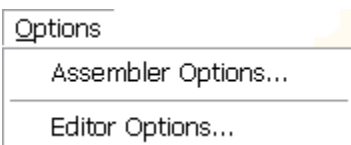
Ctrl+< - Removes one space from the start of each selected line.

Project menu:



Assemble Source... This will assemble the currently selected source window. If any errors occur a dialog will inform the user and the error messages with line number will be added to the list at the bottom of the source window. Double-click on an error to highlight the corresponding line in the source.

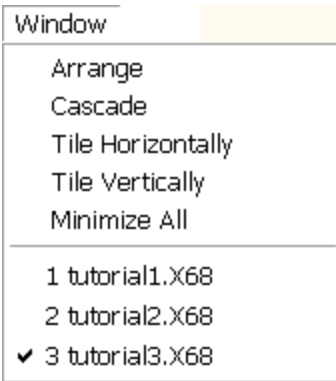
Options menu:



Assembler Options... - Opens the Assembler Options window where the behavior of the assembler may be adjusted and the default source template may be edited. See [Options](#) in this help system.

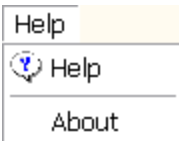
Editor Options... - Opens the Editor Options window where the font and tab characteristics of the editor may be adjusted. See [Options](#) in this help system.

Window menu:



The opened edit windows may be arranged or selected as the active window.

Help menu:



Help - Displays this help system but then you already know that, don't you.

About - Displays version and contact information.

Right-Click menu:

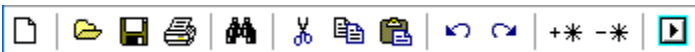
Right clicking on an edit window opens the following menu:

Undo	Ctrl+Z
Redo	Ctrl+Alt+Z
Cut	Ctrl+X
Copy	Ctrl+C
Paste	Ctrl+V
Select All	Ctrl+A
Comment Selection	Ctrl+Alt+C
Uncomment Selection	Ctrl+Alt+U
Reload File	


Some of the more commonly used commands from the main edit menu are available. Plus one new menu item:


Reload File - Reloads the current file from disk.


The Toolbar




The icons on the toolbar match the corresponding action in the menu. The actions available on the toolbar are as follows from left to right:


 **New** - Creates a new source window using the predefined source template.


 **Open File** - Loads an existing 68000 source file into the editor.


 **Save** - Save the currently selected source file. Defaults to Save As if the file is being saved for the first time.

 **Print** - Prints the currently selected source file.


 **Find...** - Opens a dialog where a search phrase may be entered. The first occurrence of the phrase in the document is highlighted.

 **Cut** - Removes the selected text and places it in the Window's Clipboard.

 **Copy** - Copies the selected text to the Window's Clipboard.

 **Paste** - Inserts the contents of the Window's Clipboard into the


document at the present cursor position.

 **Undo** - Undo the last edit operation.

 **Redo** - Redo the last undo operation.

+* **Comment Selection** - Inserts an asterisk '*' at the start of each selected line making it a comment.

-* **Uncomment Selection** - Removes one asterisk '*' from the start of each selected line.

 **Assemble Source...** This will begin the assembling process. If any errors occur a dialog will inform the user and the error messages with line number will be added to the list at the bottom of the source window. Double-click on an error to highlight the corresponding line in the source.

Getting Started

The following is a quick introduction to 68000 assembly language programming with EASy68K.

An assembly language program consists of:

labels - User created names that mark locations in a program.

opcode - Specific instructions the microprocessor can perform.

operands - Additional data required by some instructions.

directives - Commands to the assembler.

macros - A user created collection of source code that may be easily reused when writing a program.

comments - User created strings of text used to document a program.

Each line of an assembly language program contains a combination of the following four fields.

Label Field

A label is used to identify a location in a program or a memory location by name. Instructions or directives that require a location may use a label to indicate the location. A label normally begins in the first column of the line. It must be terminated with a space, tab or a colon. If a colon is used it does not become part of the label. If a label does not start in the first column it must be terminated with a colon. Only the first 32 characters of the label are significant. Two types of labels are available: Global and Local. A global label may be referenced from anywhere in the program. As such, global labels must be a unique name. Global labels should start with a letter and be followed by letters, numbers or underscores. Local labels may be reused in a program. Local labels must start with a dot '.' and be followed by letters, numbers or underscores. Global labels define the boundaries of a local label. When a local label is defined it may only be referenced from code above or below the local label until the next global label is encountered. The assembler creates a unique name

for local labels by appending the local label name to the preceding global label and replacing the dot with a colon ':'. Only the first 32 characters of the resulting name are significant.

Operation Field

The operation field follows the label field. It must be separated from the label by at least one space or tab. If no label is present on the line there must be at least one space or tab before the operation.

Operations may be 68000 opcodes, assembler directives or macro calls.

Operand Field

Operands contain extra information required by the item in the operation field. Not all operations required extra information so an operand may not be required. The operand field must be separated from the operation field by at least one space or tab.

Comment Field

The last item on the source line is a comment area. User entered text is placed here to document the program. The assembler ignores everything in the comment field. The comment field must be separated from the previous field by at least one space or tab.

For example:

<u>label</u>	<u>opcode</u>	<u>operand</u>	
<u>comment</u>			
.Loop	ADD	D0, D1	Add two
numbers			
	BMI	.Loop	Loop
while negative			

Assembly language programming requires direct interaction with the microprocessor. The 68000 microprocessor contains eight data registers D0 through D7. Data registers are general purpose. They may be thought of as 8 bit, 16 bit or 32 bit integer variables. There are eight address registers A0 through A7. Address registers are 32

bits long. They are most commonly used to reference variables. The status register (SR) contains status flags that indicate the results of comparisons. The EASy68K simulator displays the registers of the 68000 as:

Registers									
D0=	00000009	D4=	00000000	A0=	00000000	A4=	00000000	T S INT XNZVC	Cycles
D1=	12345678	D5=	00000000	A1=	0000101E	A5=	00000000	SR=	0010000000000000 48
D2=	00000000	D6=	00000000	A2=	00000000	A6=	00000000	US=	00FF0000
D3=	00000000	D7=	00000000	A3=	00000000	A7=	01000000	SS=	01000000 PC=0000101E
Address -----Code----- Line -----Source----->>									

The following program will display the text "Hello World" and the number contained in D1. This is the program as it appears in the editor.

```

outputdemo.X68

    ORG    $1000    the program will load into address $1000

* Display HELLO message
* see Help/Simulator IO for a complete list of task numbers
START  MOVE    #14,D0    put text display task number in D0
        LEA     HELLO,A1    load address of string to display into A1
        TRAP    #15        activates input/output task

* Display the contents of register D1
* task number 3 is used to display the contents of D1.L as a number
        MOVE.L  #12345678,D1    put a number in D1 so we can display it
        MOVE    #3,D0    task number 3 in D0
        TRAP    #15        display number in D1

* Stop execution
        MOVE.B  #9,D0
        TRAP    #15        Halt Simulator

HELLO   DC.B    'Hello World', $D,$A,0    null terminated string with newline

        END      START

```

What it's all about:

ORG \$1000 the program will load into address \$1000
ORG \$1000 defines where the program will be located in memory. Dollar sign '\$' indicates a hexadecimal number. The remaining blue text on the line is comment.

* Display HELLO message

Lines that begins with an asterisk '*' are comments.

START **MOVE** #14,D0 put text display task number in D0
START is a label. The START label is referenced by the END directive in the last line in the program. It is used to indicate where the program should begin execution. MOVE is the opcode, #14,D0 is the operand. The MOVE instruction moves the number 14 into data register D0. The '#' sign in front of the 14 indicates an immediate or literal value. Without the '#' the instruction would get the contents of address 14 and put it in register D0.

LEA **HELLO,A1** load address of string to display into A1
LEA loads the address of HELLO into address register A1. A1 now points to the string at the address defined by the HELLO label.

TRAP #15 activates input/output task
TRAP #15 is used to run Simulator I/O routines (tasks) that are built into the Sim68K simulator. The task number to run is in register D0. In this case the task number is 14 which tells Sim68K to display the null terminated string pointed to by register A1. See [Simulator I/O](#) in this help for a complete list of TRAP #15 tasks.

The next three lines of code display the number 12345678.

MOVE.L #12345678,D1 put a number in D1 so we can
display it
 MOVE #3,D0 task number 3 in D0
 TRAP #15 display number in D1

The following two lines of code are used to stop a program. They tell the simulator to halt.

```
MOVE.B  #9,D0
TRAP    #15
```

[Halt Simulator](#)

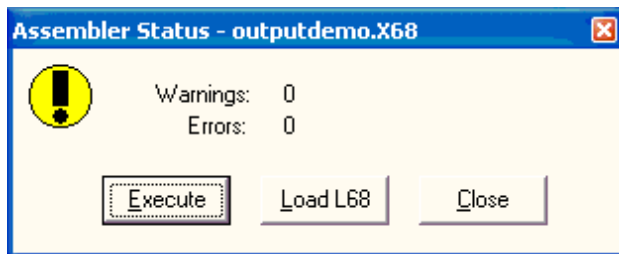
```
HELLO  DC.B    'Hello World',$D,$A,0  null terminated string with
newline
```

This line defines a text string named HELLO. DC.B defines constant bytes. The text string is enclosed in single quotes. \$D is carriage return, \$A is line feed and 0 is the null terminator.

```
END      START
```

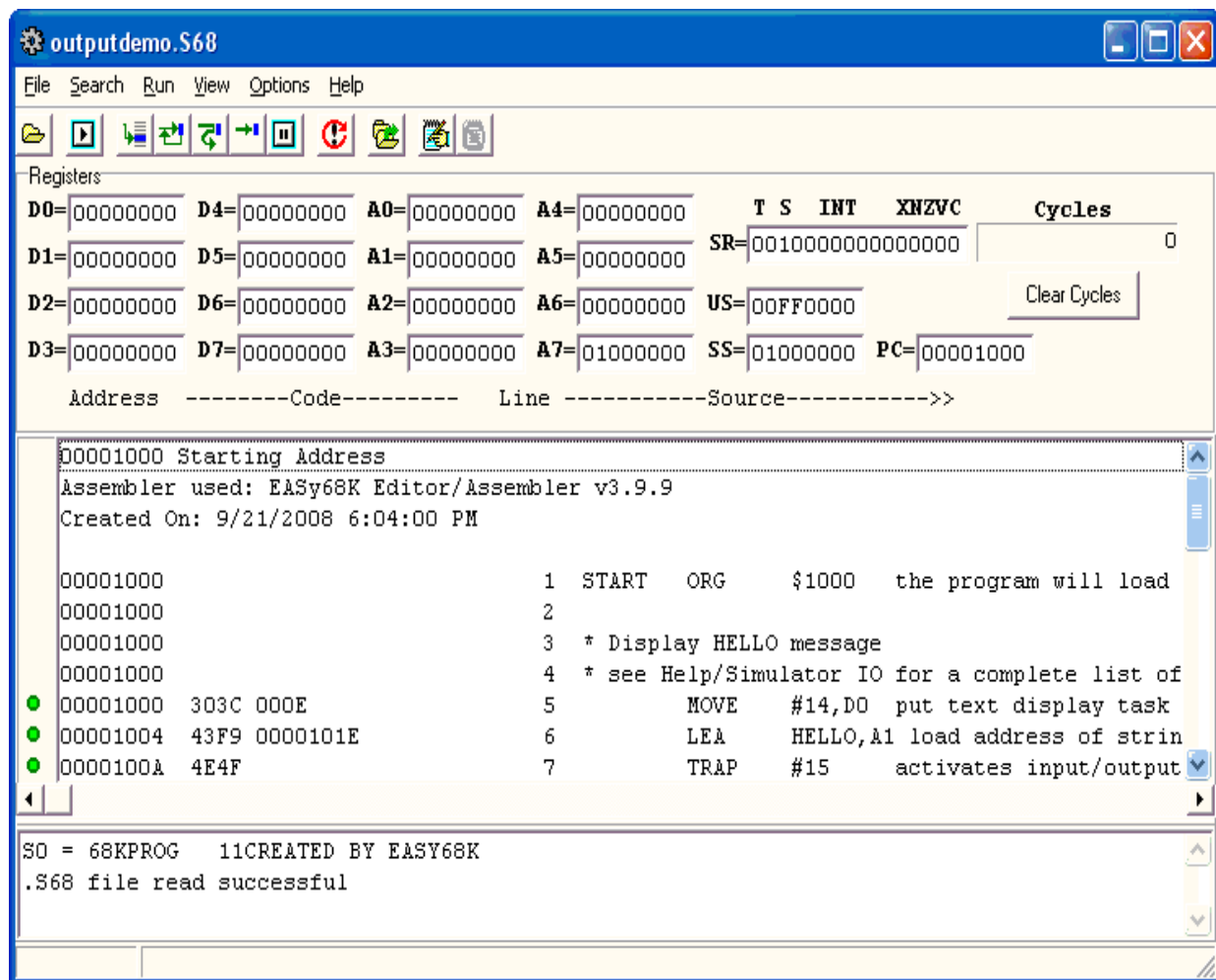
END indicates the end of the assembly program. START is the name of the label where the simulator will begin running the program.

To run the program click the Assemble Source button on the toolbar

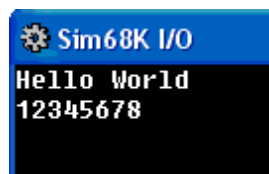


If errors are detected error messages will be displayed at the bottom of the editor window. Double click on an error message to highlight the line of code in the source. Error messages are also added to the listing file (.L68 extension) if the option is selected to create it (default). Source files have a default extension of (.X68). Files that may be executed by the simulator have an extension of (.S68).

If no errors occur, click the Execute button and the program is loaded into the 68000 simulator Sim68K.



To run the program click the Run button on the toolbar.  The output of the program is displayed.



See the EASY68K EXAMPLES folder for more example programs. More examples are also available at www.easy68k.com

EASy68K Warning and Error Messages

"WARNING: Address expected."

The starting address of the program was missing from the [END directive](#). The required syntax for the end directive is:

END Address

Where Address is a literal value or label that specifies the starting address of the program.

"WARNING: ASCII constant exceeds 4 characters"

[ASCII](#) constants such as 'ABCD' are limited to 4 characters. The resulting value is a 32 bit binary number comprised of the ASCII values of each character. 'ABCD' results in the number \$41424344 which is the ASCII code for each letter appended together beginning with \$41 the ASCII code for 'A'.

"WARNING: DO expected."

The structured assembly statements [WHILE](#), [UNTIL](#) and [FOR](#) must end with the word DO.

For example `until d1 <eq> #0 do`

"WARNING: END directive missing, starting address not set."

All programs must contain one [END directive](#) which marks the end of the assembled program. The END directives argument indicates the starting address of the program. The END directive should be the last statement in the program.

"WARNING: Forcing SHORT addressing disables range checking of extension word"

The compiler will not create an error if the address specified is outside the range of a 16 bit offset.

Addresses are [forced to use the short form](#) by appending a .w for example: `move.b label.w, d0`

"WARNING: Forward reference may cause 'Symbol value differs...' error."

A [macro definition](#) should be located above the macro call. Placing macro definitions below the macro call will result in an error if the macro call is between a branch and the branch destination.

"WARNING: Numeric constant exceeds 32 bits"

The literal number in the source file is too big. A 32 bit number may not exceed 4,294,967,295 decimal or \$FFFFFFFF hexadecimal.

"WARNING: Origin value is odd (Location counter set to next highest address)"

The address used with the ORG directive must always be an even number. If an odd address is specified, EASy68K will automatically adjust the number up to the next even location.

"WARNING: THEN expected."

The structured assembly statement [IF](#) should end with the word THEN. For example `if d1 <ne> #0 then`

"ERROR: Absolute address exceeds 16 bits"

Absolute short [addressing](#) must be in the range -32768 through 32767.

"ERROR: Branch instruction displacement is out of range or invalid"

The target address specified for the [branch instruction](#) is too far away. Short displacements must be from -128 to 127 bytes away. Word displacements must be from -32768 to 32767 bytes away. A [JMP](#) instruction may be used to transfer program control to any address.

"ERROR: Comma expected"

The operand is not complete.

"ERROR: Displacement out of range"

The displacement value used exceeds the allowed range. Word

displacements must be in the range -32768 through 32767. Byte displacements must be in the range -128 through 127.

"ERROR: Division by zero attempted"

An expression in the source code contained a mathematical operation that attempted to divide a number by zero. This error is reported by the assembler while it is attempting to resolve a literal number in the source code. It should not be confused with the exception that occurs if the 68000 [DIVU](#) or [DIVS](#) instruction attempts a divide by zero.

"ERROR: ENDM expected"

Macro definitions must end with the [ENDM](#) directive.

"ERROR: FOR without ENDF."

The structured [FOR](#) statement must have a matching ENDF.

"ERROR: Forward references not allowed with this directive"

The directive is referencing an address that is higher than the current address of the instruction. This directive only permits access to addresses that are lower than the current address. Forward references are not permitted because the referenced address may change during the assembly process.

"ERROR: IF without ENDI."

The structured [IF](#) statement must have a matching ENDI.

"ERROR: Immediate data exceeds 8 bits"

8 bit data must be in the range -128 through 127 for signed values and 0 through 255 for unsigned.

"ERROR: Immediate data exceeds 16 bits"

16 bit data must be in the range -32768 through 32767 for signed values and 0 through 65535 for unsigned.

"ERROR: Invalid addressing mode"

EASy68K did not recognize a valid 68000 addressing mode for this instruction. Verify that the addressing mode is supported by the

instruction. Check for missing or mismatched parenthesis, spaces or punctuation.

"ERROR: Invalid argument"

The argument used in the conditional assembly directive is invalid or missing.

"ERROR: Invalid bitfield."

The specification of the [bitfield](#) is invalid. EASy68K only supports valid 68000 addressing modes with the bitfield instructions.

"ERROR: Invalid block length"

The size of the memory block requested is invalid.

"ERROR: Invalid constant shift count"

Immediate values used to specify [shift](#) counts must be between 1 and 8.

"ERROR: Invalid opcode"

The opcode is not a valid [68000 instruction](#) or defined macro. EASy68K only supports 68000 opcodes and the bitfield instructions from the 68020.

"ERROR: Invalid operator"

An expression in the source code contains an unrecognized operator.

"ERROR: Invalid size code"

Size codes are normally specified by using a .B for byte, .S for short, .W for word, or .L for long. Verify that the size code is valid for the instruction and addressing mode.

"ERROR: Invalid syntax"

EASy68K detected something wrong with the statement it was attempting to assemble. Check for missing parameters, bad or missing punctuation and values that are outside the expected range.

"ERROR: Invalid vector number"

The [TRAP](#) instruction vector number must be between 0 and 15 inclusive.

"ERROR: Label is not allowed"

Labels are not permitted with this directive.

"ERROR: Label required with this directive"

A label must be used with this directive. [Labels](#) must begin in the first column of the line or must be terminated with a colon ':'.

"ERROR: Nested Macro calls are too many levels deep."

Nested [macros](#) are permitted but the assembler has detected what appears to be an infinite macro loop.

"ERROR: No matching DBLOOP statement was found."

A structured UNLESS was encountered with no matching [DBLOOP](#).

"ERROR: No matching FOR statement was found."

A structured ENDF was encountered with no matching [FOR](#).

"ERROR: No matching IF statement was found."

A structured ELSE or ENDI was encountered with no matching [IF](#).

"ERROR: No matching REPEAT statement was found."

A structured UNTIL was encountered with no matching [REPEAT](#).

"ERROR: No matching WHILE statement was found."

A structured ENDW was encountered with no matching [WHILE](#).

"ERROR: Quick immediate data range exceeds 1 byte"

The immediate data used with the [MOVEQ](#) instruction may not exceed 1 byte.

"ERROR: Quick immediate data range must be 1 to 8"

The immediate data used with the [ADDQ](#) and [SUBQ](#) instruction must be 1 through 8 inclusive.

"ERROR: Register list symbol not previously defined"

[Register lists](#) must be defined prior to use.

"ERROR: Register list symbol used in an expression"

[Register lists](#) may not be used as part of an expression.

"ERROR: Symbol defined more than once"

The symbol is defined in the program in more than one location. Only symbols that are defined with the [SET](#) directive may appear in more than one location.

"ERROR: Symbol is not a register list symbol"

[Register lists](#) may be defined and used with the [MOVEM](#) instruction. The register list must be defined prior to use.

"ERROR: Symbol value differs between first and second pass"

The value or address assigned to the symbol changed during the assembly process. This may be caused if a macro call is placed between a branch or jump instruction and its destination address. Place all macro definitions at the top of the source file.

"ERROR: Too many arguments"

Too many arguments were specified when calling a [macro](#).

"ERROR: Unable to access specified file."

There was an error while trying to access the specified [include](#) file.

"ERROR: Undefined symbol"

No definition could be found for the specified symbol. Verify the spelling of the symbol and make sure operands are separated from opcodes and comments by at least one space or tab.

"ERROR: UNLESS without DBLOOP."

The structured UNLESS statement must have a matching [DBLOOP](#).

"ERROR: UNTIL without REPEAT."

The structured UNTIL statement must have a matching [REPEAT](#).

"ERROR: WHILE without ENDW."

The structured WHILE statement must have a matching ENDW.

Assembler Directives

Introduction - Directives are instructions for the assembler to follow. They occupy the second field in the source line which is the same position occupied by instruction opcodes however directives are not 68000 opcodes. "DC" and "DCB" are the only directives that result in data being added to the output file. Directives may also be used to control the assembly of [macros](#), [conditional assembly](#) and [structured syntax](#).

In the following descriptions optional items are shown in brackets []. Items shown in italics should be replaced by the appropriate syntax.

Usage:

```
[label] directive[.size] [data,data,...]
      ^               ^               ^
      \_____ \_____ \_____ varies by
directive
```

Assembler Directives - DC

DC - The DC directive instructs the assembler to place the following values into memory at the current location. The directive has three forms DC.B for byte data, DC.W for word (16-bit) data, and DC.L for long (32-bit) data. The Define Constant directive should not be confused with declaring a constant in C++.

Usage:

```
[label] DC.size data,data,...
```

For example:

```
ORG      $1000                                start of the data
region
depart DC.B  'depart.wav',0                    stores as a NULL
terminated string in consecutive bytes
        DC.L  $01234567                        the value
$01234567 is stored as a long word
        DC.W  1,2                              two words are
stored as $0001 and $0002
        DC.L  1,2                              two long words
are stored as $00000001 and $00000002
```

The 68000 microprocessor requires that word and long word numbers be stored in even memory addresses. The assembler will adjust the memory locations accordingly. Beginning with the 68020 and later versions of the 68000 this word boundary restriction was removed.

The result of the above code would be:

```
00001000  64 65 70 61 72 74 2E 77 61 76 00
0000100C  01234567
```

```
00001010 0001 0002
00001014 00000001 00000002
```

Assembler Directives - DCB

DCB - The Define Constant Block directive fills a block of memory with the specified value. The instruction format is:

Usage:

```
[label] DCB[.size] length,value
```

The label and size fields are optional. Size defaults to .W if omitted.

```
        DCB.W    10,$FF           fill 10 words  
with $00FF
```


Assembler Directives - DS

DS - The Define Storage directive reserves the specified amount of memory at the current location. DS is qualified by .B, .W, or .L and defaults to .W if no size is specified. Unlike DC, no data is stored in the reserved memory. The assembler will force DS.W and DS.L locations to start in even memory addresses. DS.W 0 may be used to force even word alignment.

Usage:

```
[label] DS[.size] n
```

```
TABLE DS.W 256      Reserve 256 words for TABLE
flags DS.B 5        Reserve 5 bytes
      DS.W 0        Force even word alignment
for the following code
```

Note! The ORG directive may also be used to force Word or Long Word alignment.

```
ORG    (*+1)&-2    Force Word alignment
```

```
ORG    (*+3)&-4    Force Long Word alignment
```

Assembler Directives - END

END - The end directive tells the assembler that the end of a program has been reached. The operand specifies the starting address of the program.

Usage :

```
END      address
```

Assuming the starting point of the program was \$1000 the END directive would be:

```
END      $1000
```

A label may be used to specify the start of the program. If the label START specified the starting location of the program the END directive would look like this:

```
START:                                Start of program
      code
      END      START
```

The starting address is placed in the S-Record file that EASy68k creates and is used by the simulator to set the 68000's program counter to the starting address of the program.

Assembler Directives - EQU

EQU - links a name (label) to a value. Similar to defining a constant in C++.

Usage:

```
label EQU value
```

```
BACK_SP EQU $08          ASCII code for  
backspace
```

Equates BACK_SP with the hex value \$08 which is the ASCII code for backspace. In programs we can now use the word BACK_SP to represent \$08. This makes our program much easier to read and modify. Once an expression has been equated to a value it may in turn be used in other equate directives, example:

```
Length EQU 30  
Width  EQU 25  
Area   EQU Length*Width
```

Assembler Directives - FAIL (Programmer Generated Error)

DESCRIPTION - The FAIL directive forces an assembler error and prints the optional message. If no message is provided the message: "ERROR: Unspecified user defined error." is used. The total error count is incremented as with any other error. The FAIL directive is normally used with conditional assembly directives. The assembly proceeds normally after the error has been printed.

Usage:
[label] FAIL message

Example:

```
foo    MACRO
        IFC \1, ''    ;if argument 1 is missing
            FAIL ERROR, Argument missing in call to
foo macro.
        MEXIT
        ENDC
        .
        .
        ENDM
```

Assembler Directives - Include

INCLUDE - Inserts the specified EASy68K source file at the current location. The inserted file may contain any valid EASy68K source. The filename must include the extension and must be enclosed in single (') or double (") quotes if any part of the file path or name includes spaces.

Usage:

```
[label] INCLUDE file_name
```

Example:

```
* Include macro definitions
    INCLUDE "C:\EASy68K\macros\input output
macros.x68"
```

Assembler Directives - INCBIN

INCBIN - Inserts the specified binary file into the S-Record output file. The data from the included file is not processed in any way. The filename must include the extension and must be enclosed in single (') or double (") quotes if any part of the file path or name includes spaces.

Usage:

```
[label] INCBIN file_name
```

Example:

```
* WAV data for laser sound  
laser INCBIN "C:\Sounds\laser.wav"
```

Assembler Directives - LIST, NOLIST

These directives have no effect if "Generate List File" is not checked in the [Assembler Options](#).

LIST - Turns on listing of the assembled source to the corresponding .L68 file.

NOLIST - Turns off listing of the assembled source to the corresponding .L68 file.

NOLIST has no effect on error messages which are always written to the list file if "Generate List File" is checked.

The default is listing on.

```
        NOLIST
* This code will not be listed to the .L68 file
        .
        LIST
* This code will be listed to the .L68 file
        .
```

Assembler Directives - MEMORY

MEMORY - Defines the access level for a range of memory. Four access types are supported. They are:

ROM

Memory may be read or written but the writes have no effect.

Read

Memory may be read from but not written to. Any attempt to write to the memory results in a bus error exception.

Protected

Memory may only be accessed when the Supervisor flag is set in the Status Register. Accessing the memory when the Supervisor flag is not set results in a bus error exception.

Invalid

Any memory access results in a bus error exception.

Memory not specified in a specific range is assumed to be RAM. RAM may be read or written.

Usage:

```
[label] MEMORY type startAddress,endAddress
```

Where *type* is ROM, Read, Protected or Invalid. *startAddress* and *endAddress* must evaluate to a valid 24 bit address.

```
MEMORY      ROM    $001000,$002000
MEMORY Protected $00F000,$00FFFF
```

The memory map information is saved in the .S68 ([S-Record](#)) file generated by the assembler in the form of S0 records. The EASy68K simulator uses the information from the S0 records to set the memory map on the [hardware form](#).

Assembler Directives - OFFSET

OFFSET - Used to define a table of offsets. No machine code is generated by instructions or directives following an OFFSET directive. Use an ORG directive to end the offset section and cause the assembler to generate machine code again.

	OFFSET	0	Temporary origin for data
label1	DS.W	1	
label2	DS.B	2	
	ORG	*	Restore previous code origin

ORG * restores the code to the address in use prior to the OFFSET.

Stack frames are commonly used in subroutines to create local variables on the stack. Accessing the variables from the stack can be a little tedious. Here is an example that might make things just a little easier.

SIZE	EQU	-3*4	3 long words, negative
			for stack frame
	OFFSET	SIZE	Offset to local variables
is determined			
num1	DS.L	1	
num2	DS.L	1	
num3	DS.L	1	
	ORG	*	Restore previous code
			origin
	LINK	A0, #SIZE	Create stack
			frame
	MOVEM.L	A0-A1, -(A7)	Put some
			more stuff on the stack just for fun

```
        MOVE.L    #$11111111, (num1, A0)    Access the
local variable using a label
        MOVE.L    #$22222222, (num2, A0)
        MOVE.L    #$33333333, (num3, A0)
```

Assembler Directives - OPT

The OPT directive is used to set assemblers options from **OPT** within the source file. The OPT directive is followed by one of the following options:

- MEX** Macro EXpand. The following macro calls will be expanded by the assembler.
- NOMEX** No Macro EXpand. The following macro calls will NOT be expanded by the assembler.
- SEX** Structured EXpand. The following structured assembly statements will be expanded by the assembler.
- NOSEX** No Structured EXpand. The following structured assembly statements will NOT be expanded by the assembler.
- WAR** Enables display of warning messages during assembly.
- NOWAR** Disables display of warning messages during assembly.
- BIT** Enables assembly of bit field instructions. Only 68000 addressing modes are supported. Adds *
[sim68k]bitfield to .L68 file which automatically enables bit field support in Sim68K. Note! There is not a matching NOBIT option because there is no way to selectively disable bit field support in Sim68K during program execution.

```
code      OPT      MEX
          .          macros expanded in this
          .
```

OPT	NOMEX	
.		macros not expanded in
this code		

Assembler Directives - ORG

ORG - The origin directive tells the assembler where the next item is to be located in memory. The operand following ORG is the absolute value of the origin. A previously declared label may be used to specify the origin address. Forward references are not permitted.

```
ORG    $400    Origin for data
```

```
ORG    *        Set origin to current code  
location
```

The ORG directive may be used to force Word and Long Word memory alignment:

```
ORG    (*+1)&-2  Force Word alignment
```

```
ORG    (*+3)&-4  Force Long Word alignment
```

Assembler Directives - PAGE

PAGE - The page directive writes the NEW_PAGE_MARKER to the list (.L68) file if the list file is created during assembly. The page directive produces no object code. No label is permitted and any comments are ignored. If the list file is printed from the EASy68K editor a new page command will be sent to the print device each time a NEW_PAGE_MARKER is printed. The NEW_PAGE_MARKER is not printed. The NEW_PAGE_MARKER will appear in the list file as the text string "<-----
PAGE ----->"

Usage :

PAGE

Assembler Directives - REG

REG - The REG directive is used to define a register list for use with the MOVEM instruction.

```
AllRegs  REG      D0-D7/A0-A6
          .
Code      MOVEM.L AllRegs, -(SP)      Push all
registers onto stack
          .
```

Assembler Directives - SECTION

SECTION - This directive causes the program counter to be restored to the address following the last location allocated in the indicated section (or to zero if used for the first time). The ORG directive may be used within a section, at any time, to set the current program location. The assembler does not check for overlapping sections. The section directive provides a convenient way of separating code from data within a program.

Usage:

```
[label] SECTION [<number>]
```

<number> must be in the range 0..15. No section numbers are reserved in any way. By default, the assembler will begin with section 0. Labels may be used for section numbers. If no section number is specified then a label is required and will be set to the value of the current section (0..15).

```
CODE    EQU    0
DATA    EQU    1

msg1     SECTION DATA
         ORG     $2000
         DC.B    'Hello World', $d, $a, 0
         SECTION CODE
         ORG     $1000
         <code>
         SECTION DATA
msg2     DC.B    'EASy68K Rules!', $d, $a, 0
```

Macros may be written to change sections and restore the previous section as shown in the following example:


```

DATA      EQU      0
CODE      EQU      1

MAC1      MACRO
SECT\@    SECTION
          SECTION DATA
          DC.B      'Hello World.'
          SECTION SECT\@
          ENDM

```

It is also possible to write a macro that modifies its behavior using conditional assembly based upon the section it is in when invoked as shown in the following example:

```

DATA      EQU      0
CODE      EQU      1

MAC2      MACRO
SECT\@    SECTION
          IFEQ SECT\@-CODE
              NOP
          ENDC
          IFEQ SECT\@-DATA
              DC.B   'Greetings'
          ENDC
          ENDM

```


See [Macro Assembly](#) for more help on writing macros.

Assembler Directives - SET

SET - The SET directive is very similar to the EQU directive. The difference is that SET allows the symbol to be redefined and EQU does not.

Size	SET	38	Size defined as 38
	.		
	.		
Size	SET	32	Size redefined as 32
	.		

Assembler Directives - SIMHALT

SIMHALT - The SIMHALT directive produces the object code \$FFFF FFFF. This sequence of numbers is interpreted by the EASy68K simulator as a command to halt the simulator. Normally \$FFFF would result in a LINE_1111 exception. It may still be used for that purpose as long as it is not followed by \$FFFF. Pressing the Pause button  on the toolbar will re-enable the simulator controls following a SIMHALT. Program execution may be continued with the instruction following SIMHALT. No registers are modified.

Usage:

```
LABEL    SIMHALT    comment
```

=== Disable SIMHALT in Simulator ===

The SIMHALT directive may be disabled in the EASy68K simulator by embedding the following comment in the 68000 source file:

```
*[sim68k]SIMHALT_OFF
```

The result will be to treat the object code \$FFFF FFFF as a "Line F" exception instead of SIMHALT. The SIMHALT_OFF comment must be at the beginning of the source line and is not case sensitive. It is processed by the EASy68K simulator during the loading of the .L68 file so it takes effect prior to the execution of any code regardless of where it appears in the file.

Structured Control - Introduction

In assembly language, commands such as CMP and Bcc (branch instruction) can be used to implement the If, While, Until, and For statements of higher level languages. However, use of these commands to create rudimentary structured control is often time consuming and complicated. The EASy68K assembler accepts certain structured language statements and translates them into their equivalent CMP and Bcc statements. These statements do affect the condition code register.

===Keywords for Structured Control===

The following keywords are reserved for structured syntax:

DBLOOP	ENDF	ENDI
ENDW	ELSE	FOR
IF	REPEAT	UNLESS
UNTIL	WHILE	

The following keywords are used in the structured syntax but are not reserved:

AND	DOWNT0	T0
BY	OR	
DO	THEN	

Note that AND and OR are reserved instruction mnemonics, however.

The five Structured Control statements, **DBloop**, **If**, **While**, **Until**, and **For** are discussed later in this section.

Legend:

optional items are shown in brackets []

op is any valid operand

code is any number of lines of 68000 code

size is specified by one of the following: B, W or L for byte, word or long respectively

expression is any valid conditional expression

extent is specified by one of the following: S or L for short or long respectively

NOTE! The assembler creates and adds labels to your program in order to implement the structured control statements. These labels all begin with an underscore '_' and are followed by an eight digit hexadecimal number. User created labels should not start with an underscore in order to avoid errors.

Structured Control - IF

IF - Executes the enclosed code if the <expression> is true.

The IF statement has the following syntax:

```
IF[.size] expression THEN[.extent]  
    code  
ENDI
```

or

ELSE - Executes the enclosed code if the *expression* is false.

```
IF[.size] expression THEN[.extent]  
    code  
ELSE[.extent]  
    code  
ENDI
```

size - The value B, W, or L, specifying the size of the operand comparison. These values correspond to the Byte, Word, or Long word data size.

A size may not be specified when the expression consists of only a condition code.

expression - The expression tested. For a description of the expression syntax, see the [Expression Syntax](#) page.

extent - Optional value S or L, indicating the size of the forward branch to use (short or long).

code - The series of assembly commands executed.

One space should be used to separate each part of the statement.

EXAMPLE

```
IF.B D0 <LT> #5 THEN.S  
    code  
ENDI
```

Structured Control - While

While - Loops through the commands specified while the <expression> is true.

The While statement has the following syntax:

```
WHILE[.size] expression DO[.extent]  
    code  
ENDW
```

size - The value B, W, or L, specifying the size of the operand comparison. These values correspond to the Byte, Word, or Long word data size.

A size may not be specified when the expression consists of only a condition code.

expression - The expression tested. For a description of the expression syntax, see the [Expression Syntax](#) page.

extent - Optional value S or L, indicating the size of the forward branch to use (short or long).

code - The series of assembly commands executed continually while the expression is true.

One space should be used to separate each part of the statement.

===Notes===

If **expression** is false upon entry, the **code** instructions are never executed.

An **expression** of <T> is used to create an infinite loop.


```
WHILE <T> DO          infinite loop
    code
ENDW
```

The CCR's (condition code register) flags are set and tested by using this command before the ***code*** instructions are executed if at all.

EXAMPLE

```
WHILE.B D0 <LT> #5 D0.S
    code
ENDW
```

Structured Control - Repeat

Repeat - Repeats the commands specified until the <expression> is true.

The Repeat statement has the following syntax:

```
REPEAT
    code
UNTIL[.size] expression [DO[.extent]]
```

size - The value B, W, or L, specifying the size of the operand comparison. These values correspond to the Byte, Word, or Long word data size.

A size may not be specified when the expression consists of only a condition code.

expression - The expression tested. For a description of the expression syntax, see the [Expression Syntax](#) page.

extent - Optional value S or L, indicating the size of the branch to use (short or long). If no extent is specified the assembler will use a short branch if possible.

code - The series of assembly commands executed until the <expression> is true.

One space should be used to separate each part of the statement.

===Notes===

The **code** instructions are executed once even if the **expression** is true upon entry.

The CCR's (condition code register) flags are set and tested by using this command after each execution of the **code** instructions.

EXAMPLE

```
REPEAT  
    code  
UNTIL.B D4 <LT> #5 D0.S
```

Structured Control - For

For - Utilizes *op1* as a counter and loops while *op1* is between the values of *op2* and *op3* inclusive. Loops may be written that count up (TO) or down (DOWNTO). The user may specify the step size with *op4*. If no step size is specified a default step size of #1 is used.

The For statement has the following syntax (**One space should be used to separate each part of the statement**):

```
FOR[.size] op1 = op2 TO op3  [BY op4]  DO[.extent]  
    code  
ENDF
```

or

```
FOR[.size] op1 = op2 DOWNTO op3  [BY op4]  
DO[.extent]  
    code  
ENDF
```

size - Optional B, W, or L, specifying the size of the operand comparison. These values correspond to the Byte, Word, or Long word data size. If any of the four operands is an address register, **size** may not be B (byte).

op1 - The counter register. Must be an addressing mode that allows modification of the destination.

op2 - The initial number to be stored in **op1**. May be any addressing mode.

op3 - The number to be counted up/down to. The direction of count is specified by either **TO** or **DOWNTO**. May be any addressing mode.

op4 - Optional amount to step by. If omitted, you must NOT include the **BY** keyword. The step defaults to one. May be any addressing mode.

extent - Optional value S or L, indicating the size of the forward branch to use (short or long).

code - The series of assembly commands executed until the counter **op1** equals **op3**.

===Notes===

Immediate numbers may be used for **op2**, **op3**, and **op4**. To do this, simply place a # sign before the number. For example:

```
FOR.L D1 = #7 TO #36 BY #2 D0.S  
    code  
ENDF
```

The FOR-TO loop is not executed if **op2** is greater than **op3** upon entry. Similarly, the FOR-DOWNT0 loop is not executed if **op2** is less than **op3**.

The FOR uses signed comparisons so the following code does not loop because #255 is interpreted as -1 on byte size comparisons. Changing from FOR.B to FOR.W or FOR.L will correct the loop.

```
FOR.B D1 = #1 TO #255 D0.S  
    code  
ENDF
```

The bits of the CCR (condition code register) are modified before the **code** is run.

Structured Control - DBloop

DBloop - Implements a loop using the decrement and branch conditional instruction (DBcc). After each pass through the loop the data register Dn is decremented by 1. The loop continues until Dn = -1 or until an optional <expression> is true.

The DBloop statement has the following syntax:

```
DBLOOP Dn = op1  
    code  
UNLESS[.size] [expression]
```

size - The value B, W, or L, specifying the size of the operand comparison. These values correspond to the Byte, Word, or Long word data size.

A size may not be specified when the expression consists of only a condition code.

expression - The expression tested. For a description of the expression syntax, see the [Expression Syntax](#) page. If the expression field is empty or contains <F> the assembler will output a DBRA instruction resulting in a loop that loops *op1* + 1 times.

code - The series of assembly commands executed until the **expression** is true or Dn is decremented to -1.

op1 - The initial number to be stored in data register Dn. May be any addressing mode.

One space should be used to separate each part of the statement.

===Notes===

The **code** instructions are executed once even if the **expression** is true upon entry. Compound expressions are not supported.

The CCR's (condition code register) flags are modified by this command after each execution of the **code** instructions.

EXAMPLE

```
DBLOOP D0 = #5                loops 6 times
    code
UNLESS.W (A0) <LT> #$200       exit loop if (A0) <
$200
```

```
DBLOOP D0 = #4                loops 5 times
    code
UNLESS
```

Structured Control - Expression Syntax

When using the **DBloop**, **If**, **Repeat**, and **While** structured control statements, the programmer must include an expression that specifies the condition to test. The EASy68k assembler accepts three different types of expressions: Condition Only, Relational and Compound. All of the conditional expressions use conditional tests which are formed by enclosing a two-letter condition inside angle brackets <>. See [Valid Condition Codes](#) below for a complete list of supported conditions.

=== Condition Only ===

The Condition Only expression uses only a conditional test. With the Condition Only expression the assembler will insert a single branch instruction to accomplish the desired logical result. For example, the code:

```
IF <EQ> THEN
    code
ENDI
```

simply checks the Z bit of the CCR and runs ***code*** if Z is set. The code created by the assembler can be seen in the assembler created .L68 file by checking the option "Structured Expanded" under the menu [Options/Assembler Options](#) or using the assembler directive "[OPT SEX](#)". The IF statement would create code that looks similar to the following:

```
IF <EQ> THEN
    BNE      .00000000
    code
```



```
ENDI  
.00000000
```

The assembler has inserted the BNE .00000000 instruction and the matching .00000000 label so the **code** inside the IF statement will only execute when the <EQ> condition is TRUE.

===Relational Expressions===

The Relational Expression consists of two operands and a conditional test. These operands and the conditional test are combined in the form:

```
op1 <xx> op2
```

In this form, the expression is translated to the following when assembled:

```
CMP    op1,op2  
Bxx    .LABEL
```

Only operands that are valid in a CMP instruction may be used but the order is not important. The assembler will arrange the operands as needed to create a valid CMP/Bxx instruction sequence. The various compare instructions and their legal addressing modes are listed below in Table1. The assembler will attempt to create a valid CMP/Bxx instruction sequence to implement the logic of the specified expression. For example, the expressions:

```
if D1 <gt> #5 then      or      if #5 <lt> D1 then
```

both create the compare/branch sequence:

```
CMP #5,D1  
BLE .label
```

The assembler will generate an error if the operands provided may not be arranged to construct a valid CMP instruction, as listed in Table1.

===Compound Expressions===

All of the structured statements except for DBloop may use compound expressions. Compound expressions are made up of two relational expressions joined by a logical operator (AND or OR). Each relational expression is evaluated to be true or false and then is AND-ed or OR-ed to the other. If the result of the compound expression may be determined after the first relational expression is evaluated then the second expression is not evaluated. This is sometimes referred to as "Short-Circuit" evaluation.

A size may be specified for each relational expression making up a compound expression. To do this, append the size of the first relational expression on the directive; the second relational expression's size is appended on the logical operator. For example the code:

```
IF.W    D3 <GT> NUMBER    OR.L    (A2) <eq> D7    THEN
```

causes the first comparison to be a word and the second to be a long word.

Compare Instructions and their Legal Addressing Modes				
Instruction		First Operand		Second Operand

Dn	CMP		(A11)	
	CMP		Dn	
	(A11) ²			
	CMPA		(A11)	

An	CMPA		An	
(All) ²	CMPI		Immediate	
alterable) ¹	CMPI		(Data alterable) ¹	
	CMPM		(An)+	
				Immediate ²
				(An)+

(Table1)				

Valid Condition Codes:

The expression consists of a two-letter condition code enclosed in angle brackets <cc>. The condition codes are the same as those used in the Bcc instruction. The following condition codes may be used:

<CC>	Carry Clear
<CS>	Carry Set
<EQ>	Equal
<GE>	Greater or Equal
<GT>	Greater Than
<HI>	Higher (Unsigned Comparison)
<HS>	Higher or Same (Unsigned Comparison)
<LE>	Less than or Equal
<LO>	Lower (Unsigned Comparison)
<LS>	Lower or Same (Unsigned Comparison)
<LT>	Less Than
<MI>	Minus
<NE>	Not Equal
<PL>	Plus
<VC>	Overflow Clear
<VS>	Overflow Set

Special case

 <T> True (Used with While statement to
create an infinite loop)

 <F> False (May be used in Unless
statement to force use of DBRA.)

¹Data alterable - The operand must be an alterable memory location or register.

²This addressing mode is only supported within structured control expressions. The assembler will rearrange the order of the operands to create a legal addressing mode.

Assembler Commands - Operators

The following operators are used in EASy68K. These operators only affect the output of the assembler. They are not "run-time" operators.

Unary Operators

```
* The current address
- unary minus
~ one's complement
$ hex digit
% binary digit
@ octal digit
' ASCII
```

Binary Operators

```
+ add
- subtract
* multiply
/ divide
& logical AND
! or | logical OR
^ exclusive OR
>> shift right
<< shift left
\ modulus
```

Operator Precedence

Operators at the top of the table have higher precedence (are evaluated before) lower operators. Operators on the same row have equal precedence and are evaluated left to right.

```
>> <<
& ! | ^
* / \
+ -
```

Assembler Commands - Macro

A macro is a user declared sequence of instructions. When a macro is invoked the code contained in the macro is inserted into the program in place of the macro invocation. Unlike a subroutine, a macro does not exist as code until it is invoked and then the code is inserted into the program. Each macro must be defined by a unique name. Once a macro is defined the code that it contains may be inserted into a program by using the macro name in the assembly source in the same way an instruction or opcode would be used. The macro definition should appear in the code before it is invoked. Calling macros that are defined later in the code may result in an error if the macro call is between a branching instruction and its destination label. The error message:

ERROR: Address or value of symbol has changed during assembly.

See 'Help' on Macros for possible causes will be displayed if such an error occurs. Placing the macro definitions at the top of the code will prevent macro calls from causing this error.

Macros are defined using the syntax:

```
macName MACRO
    code
ENDM
```

Up to thirty-six parameters may be used in the macro as \0 through \9 and \A through \Z. The parameters are specified when the macro is invoked. The assembler replaces the macro argument \n with the parameter specified. Parameter \0 always refers to the .size code letter (B or W or L). Use angle brackets to enclose arguments that contain white space <this is one argument>. See MACRO.X68 in the examples folder for an example.

Example:

```

        OPT    MEX
* Convert the ASCII character in \1 to upper case
toUpper MACRO
        IFNC    \0,B                                ;if .B
not specified
        FAIL ERROR: toUpper is .B only
        MEXIT                                       ;exit
macro
        ENDC
        CMP.B   #'a',\1
        BLO     NOT\@                               ;if not
lower case
        CMP.B   #'z',\1
        BHI     NOT\@                               ;if not
lower case
        SUB.B   #$20,\1                             ;convert
to upper case
NOT\@
        ENDM

START    ORG     $1000
        MOVE.B  #'a',D0
        toUpper.B D0                                ;invoke
toUpper macro, D0 is parameter \1

        MOVE.B  #9,D0
        TRAP    #15                                 ;Halt
Simulator
        END START
```

Labels Within Macros

The \@ directive creates a unique label. The \@ directive should be used to create labels in macros to avoid having multiple definitions of

the same label when a macro is invoked more than once. The \@ directive may be used in conjunction with other text to create a descriptive label. The assembler will replace \@ with a string of the form _n, where n is a unique decimal number.

References to an assembler-generated label always refer to the label of the given form defined in the current level of macro expansion. Such a label is referenced as an operand by specifying the same character string as that which defines the label.

The MEXIT directive

The directive MEXIT may be used to exit macro processing. All remaining statements in the macro are skipped up to the ENDM directive.

The NARG directive

NARG is a special symbol when referenced within a macro expansion. The value assigned to NARG is the index of the last argument passed to the macro in the parameter list (even if nulls). NARG is a reserved word inside a macro but is undefined outside of macro expansion, and may be referenced as a user-defined symbol.

In the following example the NARG directive is used to check for the presence of two arguments. If the macro is called with fewer or more than two arguments an error is created by using the [FAIL](#) directive and the macro is exited with MEXIT:

```
MAC      MACRO
          IFNE NARG-2                      ;if not
2 arguments
          FAIL ERROR, MAC requires 2 arguments
          MEXIT
        ENDC
        MOVE.\0    \1,D0
```

```

        CLR.L      \2
        ENDM

        OPT        MEX
START    ORG        $1000
        MAC        #'A'                ;causes
error 'ERROR, MAC REQUIRES 2 ARGUMENTS'
        MAC        #1,D0

        MOVE.B     #9,D0
        TRAP       #15                ;Halt
Simulator
        END        START

```

Null arguments and Conditional Assembly in Macros

The [conditional assembly](#) directive IFxx may be used to test for missing (null) parameters. To assemble conditionally if parameter 1 is null, either of the following directives would be correct:

```

        IFC ' ', '\1'
        or
        IFC '\1', ' '

```

To assemble conditionally if a parameter is present would use either of the IFNC formats analogous to the above two.

The condition IFARG may also be used to check for the presence of an argument in a macro call. The syntax is:

```

        IFARG n
        <statements>
        ENDC

```

Where n specifies the argument (0 - 9, A - Z). If the specified argument is not present in the macro call the enclosed statements

are not included in the program.

It is possible to specify a null argument in a macro call by an empty string (not a blank) ' ' or empty angle brackets <>. The following listing is a portion of an .L68 file created by EASy68K. Line 7 contains the assembler option [MEX](#) to expand macros in the listing file. The macro is named foo and is defined in lines 8 through 14. Line 9 in the macro is using the conditional assembly directive IFC to check for the presence of argument 2. Lines 17m through 36m are macro calls and the expanded macro statements as indicated by the lower case m following the line number. If nested macro calls are made multiple m's are displayed. The macro foo is called on line 17m with 3 arguments. Line 18m shows the result of comparing argument 2 with ' ' was FALSE. Since the result of the IFC conditional statement was false the NOP instruction on line 10 of the macro was not assembled. On line 23m angle brackets <> are used to specify argument 2 is null. Line 24m shows the result of the IFC statement is TRUE and line 25m shows the NOP instruction has been assembled as a result. The same is true on line 30m where an empty string ' ' is used for argument 2.

		7	OPT	MEX
		8	foo MACRO	
		9	IFC	'\2', ''
		10	nop	
		11	ENDC	
		12	move	#\1, d0
		13	move	#\3, d1
		14	ENDM	
		15		
		16	START	
		17m	foo	1, 2, 3
no null argument				
	FALSE	18m	IFC	'2', ''
		19m	ENDC	
303C 0001		20m	MOVE	#1, D0

323C 0003	21m	MOVE	#3, D1
	22m	ENDM	
	23m	foo	4, <>, 6
argument 2 is null			
TRUE	24m	IFC	'', ''
4E71	25m	NOP	
	26m	ENDC	
303C 0001	27m	MOVE	#4, D0
323C 0003	28m	MOVE	#6, D1
	29m	ENDM	
	30m	foo	7, '', 9
argument 2 is null			
TRUE	31m	IFC	'', ''
4E71	32m	NOP	
	33m	ENDC	
303C 0001	34m	MOVE	#7, D0
323C 0003	35m	MOVE	#9, D1
	36m	ENDM	
	37		
103C 0009	38	MOVE.B	#9, D0
4E4F	39	TRAP	#15
Halt Simulator			
	40		
	41	END	START

Refer to the Examples folder for additional sample code.

Assembler Commands - Conditional Assembly

Conditional assembly is supported in all code areas including macros. The assembler recognizes two different conditional statements. One syntax compares the equality of two strings and has the form:

```
IFxx string1,string2
      code
ENDC
```

The condition xx is either C or NC. IFC means if compare (the strings are equal). IFNC means if not compare (the strings are not equal). If the condition is true the following code is included in the program.

Another syntax compares an expression against zero and has the form:

```
IFxx expression
      code
ENDC
```

The condition xx is either:

```
EQ (expression = 0)
NE (expression <> 0)
LT (expression < 0)
LE (expression <= 0)
GT (expression > 0)
GE (expression >= 0)
```

The expression is compared with 0. If the condition is true the following code is included in the program.

The expression must be absolute, no forward references are allowed. IFxx and ENDC directives may not be labeled.

Conditional assembly may be used to temporarily add or remove code. In the following example, debug code may be included during assembly by changing the 0 to a 1.

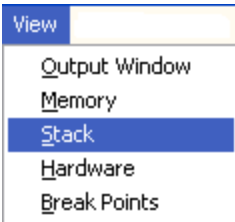
```
debug    equ    0                set to 1 to include
debug code

        ifne    debug
            debug code goes here
        endc
```

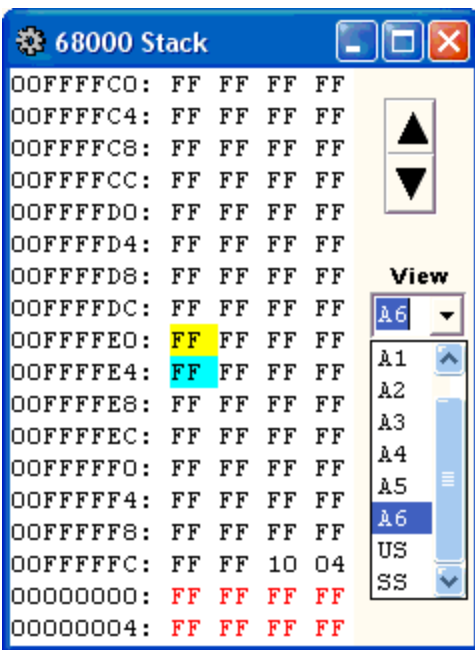
Refer to the Examples folder for additional sample code.

Sim68K - Stack Window

The 68000's stack space may be viewed by selecting Stack from the View menu.

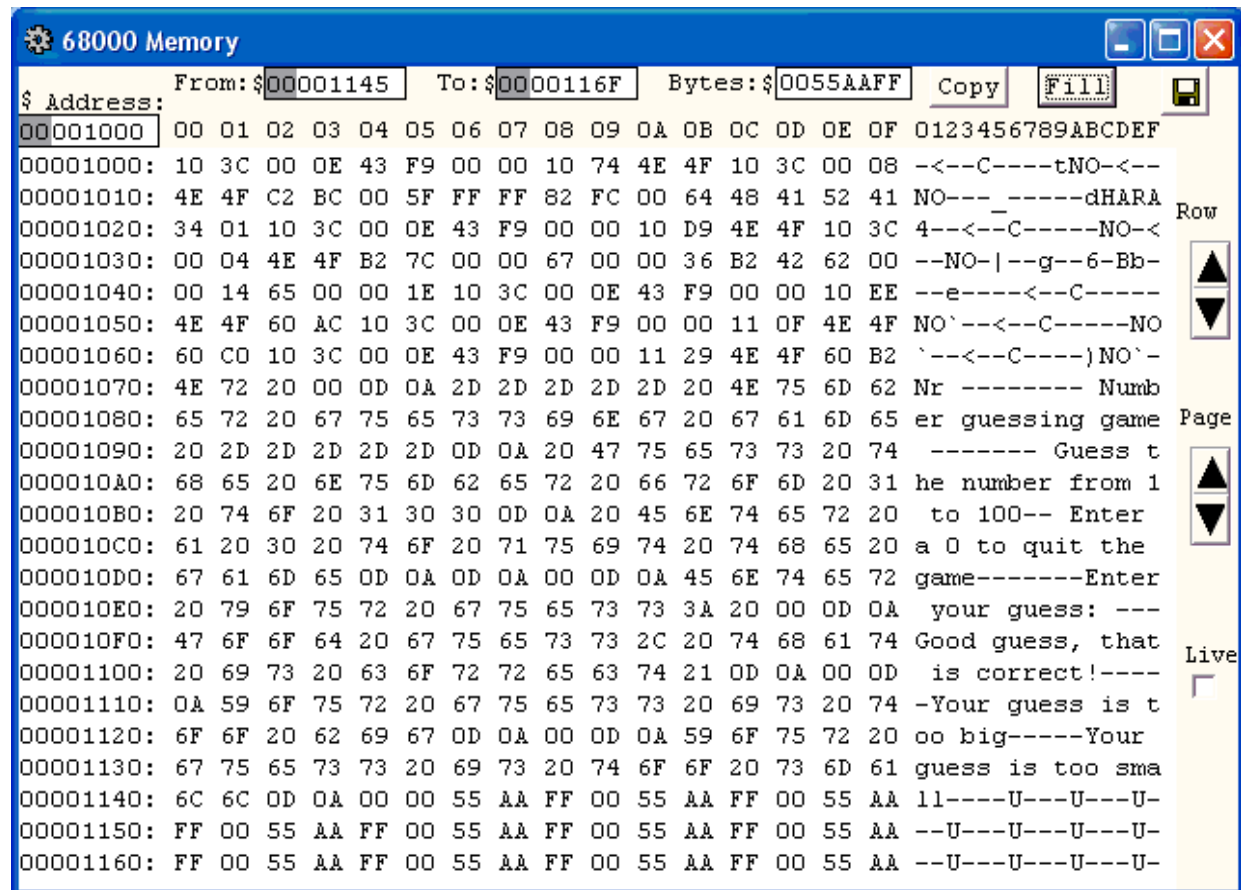
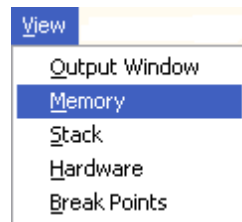


Address registers A0 - A6, User Stack (US) or System Stack (SS) may be selected as the stack pointer. The stack space is displayed for the currently selected register. The location of the selected register is highlighted in **blue** and the location of the current system stack is highlighted in **yellow**. Use the **Up/Down** buttons or scroll wheel on the mouse to move through stack memory. Memory ranges in the Invalid address space as specified on the [Hardware](#) window are displayed in red.



Sim68K - Memory Window

The simulator provides a full 16 Mega Byte 68000 address space from address \$00000000 - \$00FFFFFF. To view the 68000's memory select Memory from the View menu.



Each row of the memory window displays an address followed by 16 bytes of hexadecimal memory data, followed by the ASCII representation of the 16 bytes. The contents of memory may be changed by clicking on the desired location and entering a new value. Entries may be made in Hexadecimal or ASCII. Use the **Row** and **Page** buttons or the mouse wheel to scroll up or down through memory. To jump to a certain address, enter it in the **Address:** field.

Copy Blocks of memory may be copied. Enter the From and To address and the number of Bytes to copy and click the Copy button. Memory copy is useful when testing position independent code.

Fill

Blocks of memory may be filled. Enter the From and To address and the Bytes to fill with and click the Fill button.



Blocks of memory may be saved to a binary file. Enter the From and To address and click the Save button.

Live

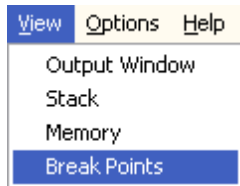
When checked the memory window presents a live view of 68000 memory.

* All numbers are entered in hexadecimal.

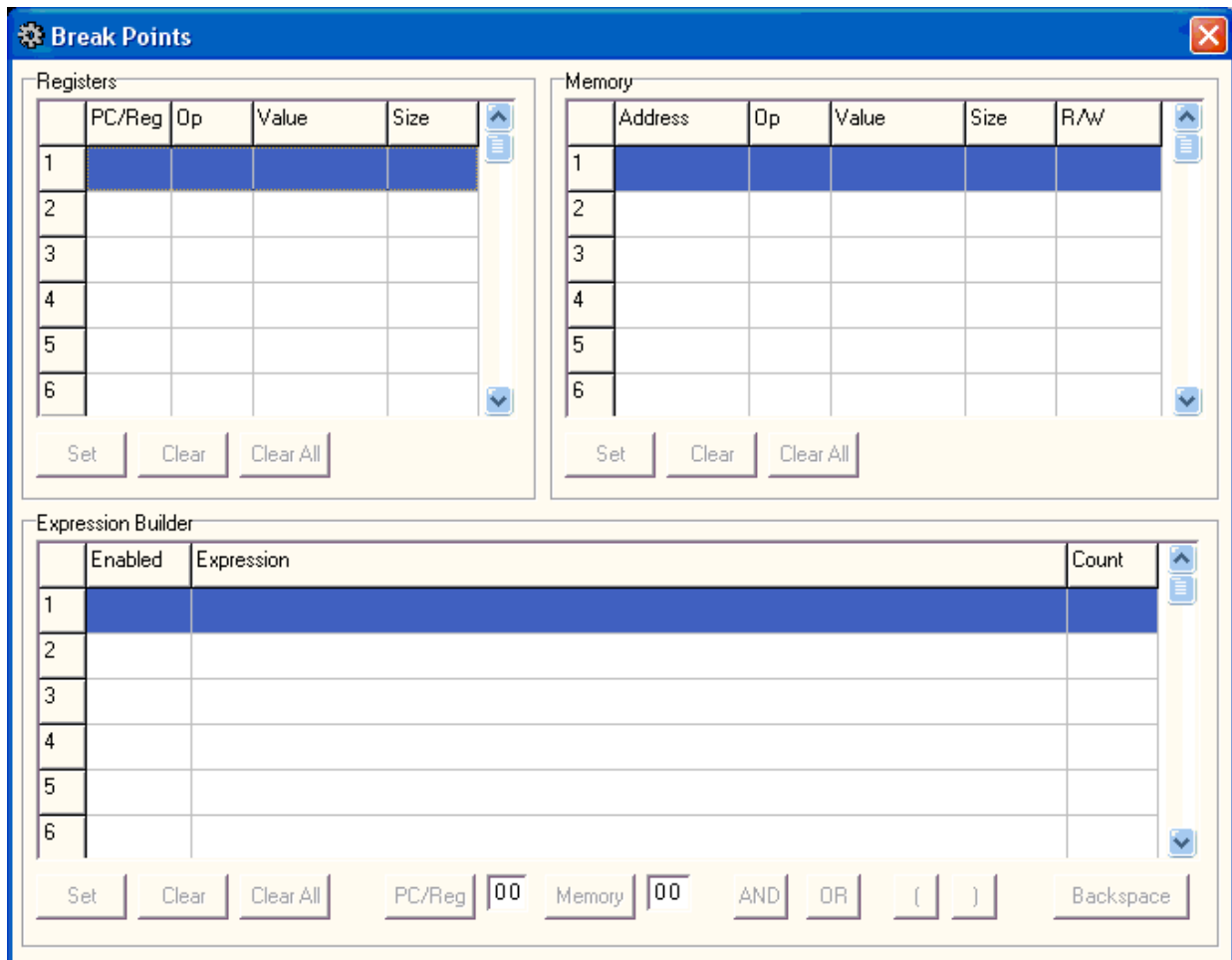
Sim68K - Advanced Breakpoints

To cause your program to Break on special conditions, use the Advanced Break Point features.

To access these features, click the View Menu and select Break Points.

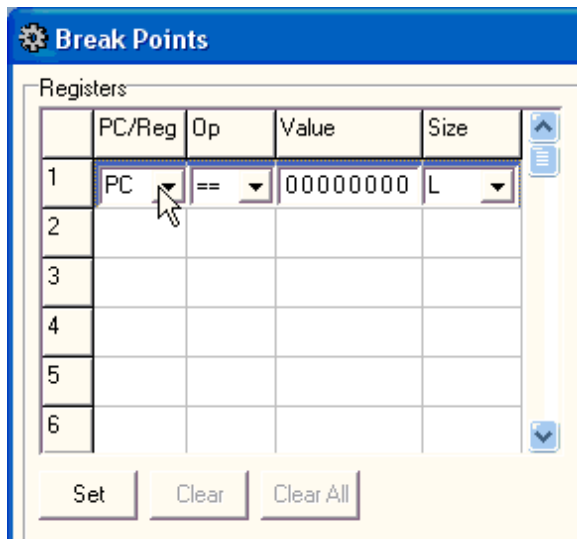


A window similar to the following will appear:



===Setting-up conditions===

To begin setting up a condition, double click on an empty line in either the **Registers** or **Memory** section.



Once clicked, the line will fill in with numerous drop-down and input boxes.

=Columns in the Registers section=

The **PC/Reg** column specifies the register to be tested.

The **Op** selection sets the type of comparison. (greater than, less than, equal to, etc.)

The **Value** selection sets the value to be compared to the register.

The **Size** specifies the size for the comparison. (byte, word, or longword)

=Columns in the Memory section=

The **Address** selection specifies address in memory to be tested.

The **Op** selection sets the type of comparison. (greater than, less than, equal to, etc.) NOTE: The **N/A** selection causes the comparison to always be true.

The **Value** selection sets the value to be compared to the location in memory.

The **Size** specifies the size for the comparison. (byte, word, or longword)

The **R/W** selection has four choices:

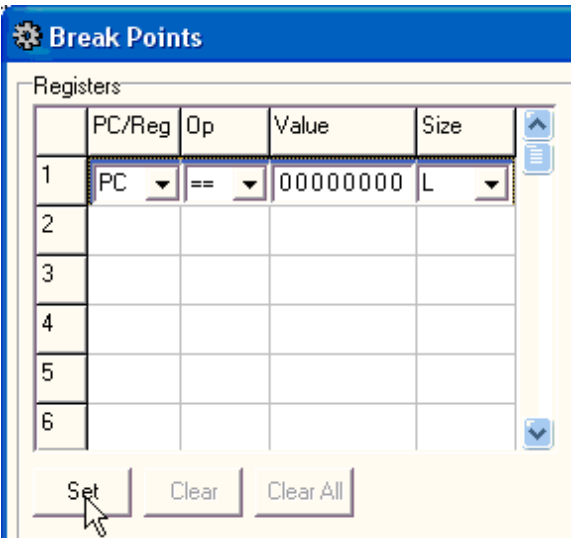
R/W - Break will only occur on read or write while the comparison is true.

Read - Break will only occur on read and when comparison is true.

Write - Break will only occur on write and when comparison is true.

N/A - Break will occur if comparison is true.

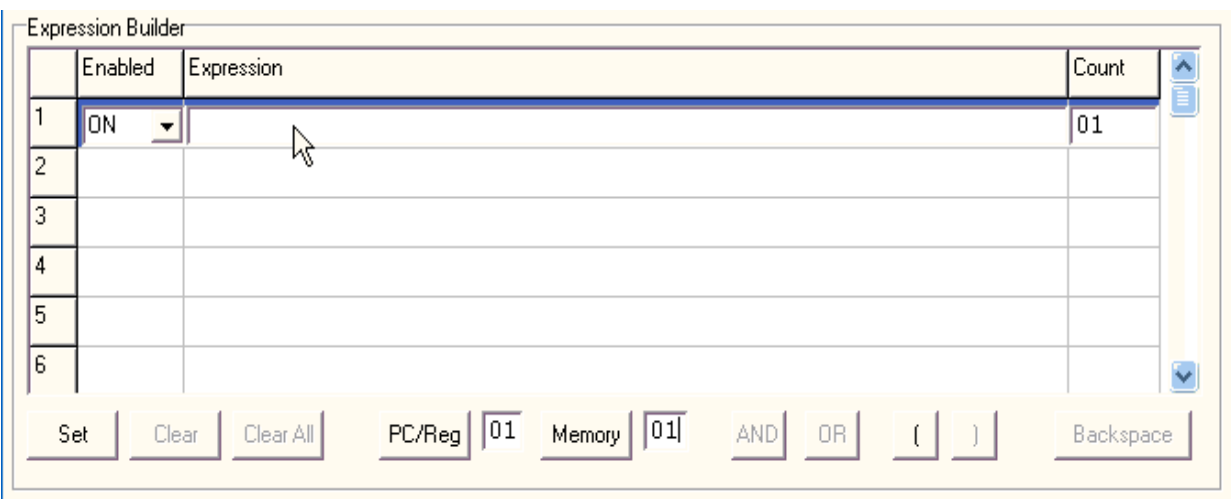
After you have finished inputting information, you must hit the SET button, or the line you just inputted will be deleted.



Also, once you have done these steps, your Break Points are **NOT** functional. You must use the conditions you have created in the Expression Builder to activate them.

===Using the Expression Builder===

To begin creating an expression, double-click on an empty row.



=Columns in the Expression Builder=

Set the **Enabled** column to **On** to activate the Break Point. Set it to **Off** to prevent the Break Point from occurring.

The **Expression** column stores an expression containing the conditions you created earlier along with optional logical operators. These conditions and optional logical operators are entered by clicking the **PC/Reg**, **Memory**, **And**, **Or**, **(**, and **)** buttons which are described below.

The **Count** column specifies how many times the expression must be true before breaking.

=Expression buttons=

PC/Reg - Inserts a condition defined earlier in the **Registers** section into the current position in the **Expression** column. The number of the condition inserted is specified by the digits to the right of the button.

Memory - Inserts a condition defined earlier in the **Memory** section into the current position in the **Expression** column. The number of the condition inserted is specified by the digits to the right of the button.

And / Or buttons - Inserts a logical operator into the **Expression** column.

(/) buttons - Groups part(s) of an expression together.

Backspace - Removes the last change made to the current **Expression**.

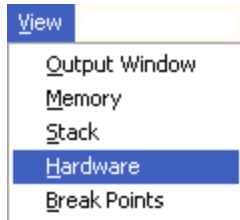
=Finalizing Expression Builder=

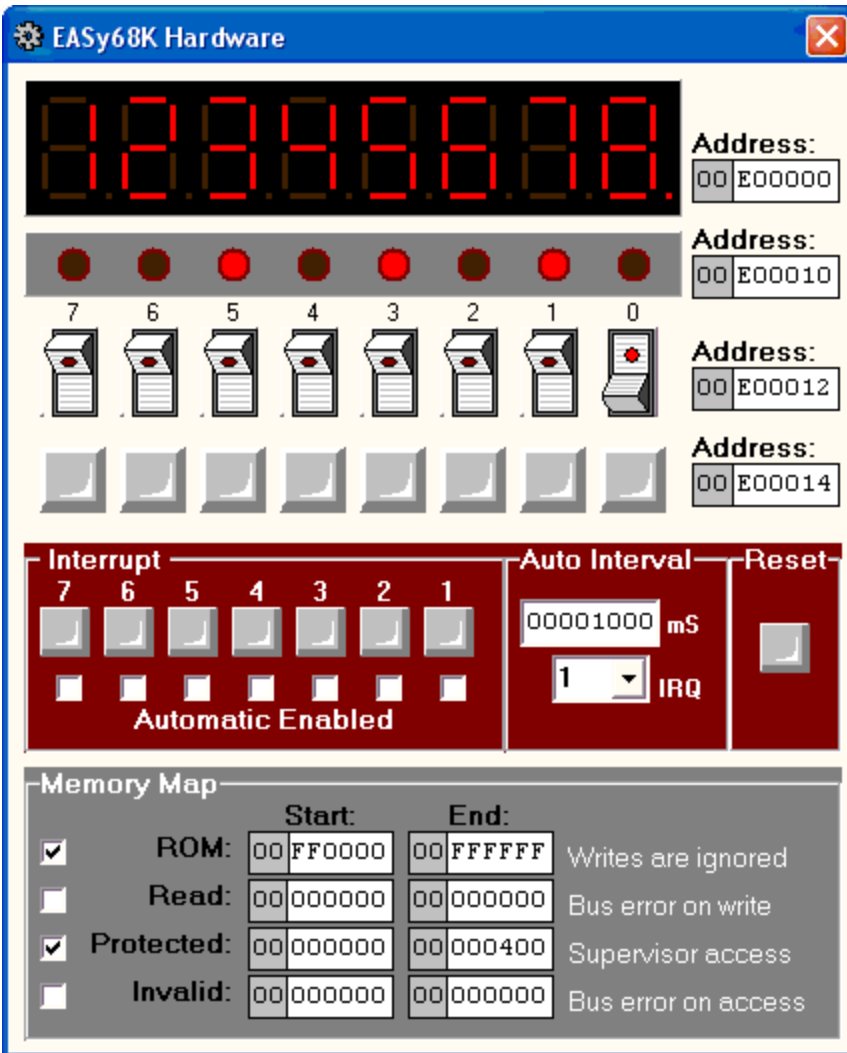
After you have finished inputting information, you must hit the **SET** button, or the line you just inputted will be deleted.

Now, your Break Points are active as long as you have set the **Enabled** column to **On**. You may change information in the **Registers**, **Memory**, or **Expression Builder** sections by double clicking on the line you want to change.

Sim68K - Hardware Window

Sim68K includes support for hardware simulation. To view the Hardware window, select 'Hardware' from the View menu.





The hardware window displays an 8 digit 7-segment display, a bank of 8 light emitting diodes (LED), a bank of 8 toggle switches and a bank of 8 push button switches. Each of these hardware items may be mapped to any valid 68000 address by entering the desired address in the corresponding **Address:** field. If the address entered causes a conflict with another device the color of the address text will change to red.

The 7-segment display has each digit mapped to a successive word address beginning with the left-most digit. For example, in the above picture the digits are mapped to memory as follows:

DIGIT	ADDRESS
1	E00000

2	E00002
3	E00004
4	E00006
5	E00008
6	E0000A
7	E0000C
8	E0000E

The toggle switches write a 1 to the corresponding bit in memory when the switch is on "Up" and write a 0 when the switch is off "Down".

The push button switches are normally high. They write a 0 to the corresponding bit in memory while the switch is pressed and write a 1 when the switch is released.

Interrupt

The Interrupt pushbuttons may be used to manually simulate an interrupt request. Pressing each button generates the corresponding interrupt request. For more on exception processing see [Enable Exceptions](#) under Simulator Options and [exception processing](#).

An interrupt may be set to automatically create interrupts by selecting the desired interrupt in the "Auto Interval" drop down and then entering the interval in milliseconds. Checking the checkbox under the interrupt pushbutton enables auto interrupts for the corresponding interrupt.

The Reset pushbutton simulates a hardware reset.

Memory Map

A 68000 microprocessor contains 32 bit registers but externally it only supports a 24bit memory space from \$00000000 through \$00FFFFFF. Any attempt to access memory beyond \$00FFFFFF

results in the upper 8 bits of the address being ignored. Therefore, attempting to access memory location \$nn002000, where nn is \$01 through \$FF, actually results in an access to \$00002000. EASy68K emulates this behavior.

Six different types of memory access are supported by EASy68K, they are: RAM, Hardware, ROM, Read, Protected and Invalid. RAM is assumed unless otherwise specified. The memory types:

RAM - Read and Write

Hardware - Any of the simulated hardware devices from the Hardware window above.

ROM - Read Only Memory, Writes are ignored.

Read - Read only, Writes result in a [Bus error](#).

Protected - Read and Write only if the Supervisor bit in the Status Register is set, otherwise a [Bus error](#) results.

Invalid - Any access results in a [Bus error](#).

The checkbox is used to enable/disable the corresponding memory type.

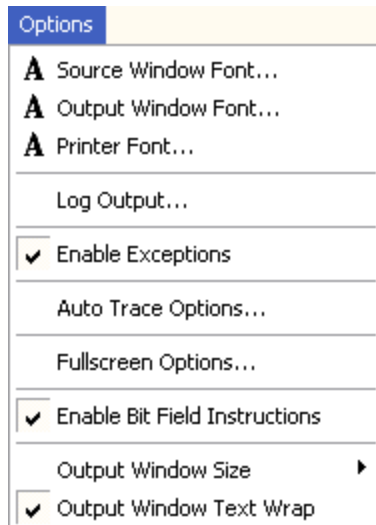
The simulator checks the memory map access in the following order: Hardware, Invalid, Protected, Read, ROM. Hardware accesses are always allowed.

The [Memory](#) assembler directive may be used to specify memory types in source code.

[Trap task 32](#) may be used to interact with the simulated hardware from a program.

* All Addresses are entered in hexadecimal, the Auto Interrupt timer is a decimal value.

Sim68K - Simulator Options



Source Window Font - Set the font used in the source code window.

Output Window Font - Set the font used in the output window.

Printer Font - Set the font used by the printer.

Log Output - Log Sim68K output to a text file. See Log Output below for more information.

Enable Exceptions - 68000 [exception processing](#). When checked any exceptions that occur during program execution will be directed to the appropriate 68000 exception vector. The 68K program being simulated is responsible for setting up the exception vectors and handling all exceptions.

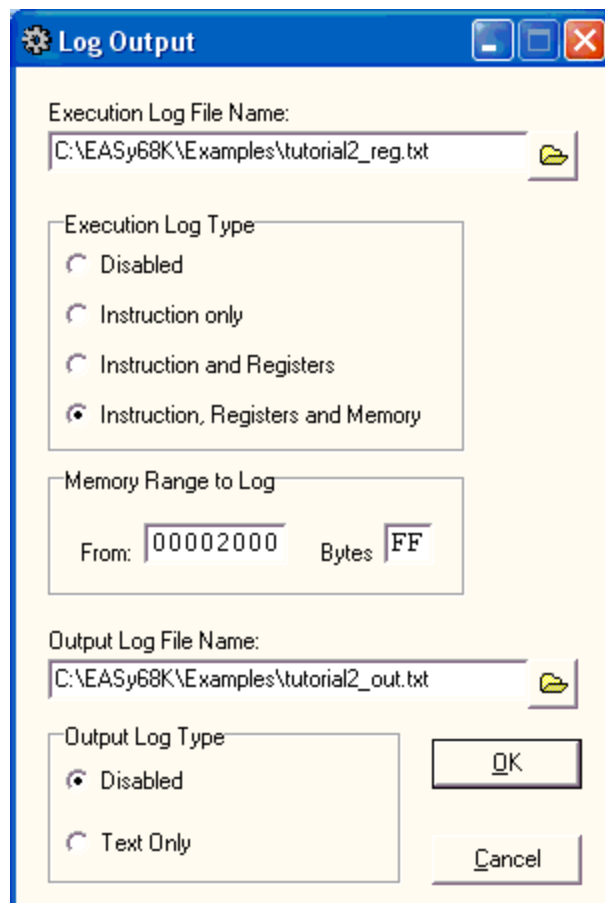
Auto Trace Options - Set the Auto Trace interval.

Full Screen Options - Set the screen to use for full screen output.

Enable Bit Field Instructions - The simulator will run a hybrid form of bit field instructions (Only 68000 addressing modes are supported). Bit field instructions are only present in 68020 and higher 68K processors and were not part of the 68000 instruction set. Support has been added to EASy68K for educational purposes. This option is automatically set if the .L68 file contains a comment in the form: `*[sim68k]bitfield` This comment is added to the .L68 file when the [OPT BIT](#) directive is assembled by EASy68K.

Output Window Size - Set the size of the output window.

Output Window Text Wrap - Long text output wraps at right edge of screen.



The image shows a Windows-style dialog box titled "Log Output". It contains several sections for configuring logging. The first section, "Execution Log File Name:", has a text field with the path "C:\EASy68K\Examples\tutorial2_reg.txt" and a folder icon button. The second section, "Execution Log Type", has four radio button options: "Disabled", "Instruction only", "Instruction and Registers", and "Instruction, Registers and Memory", with the last one selected. The third section, "Memory Range to Log", has two text fields: "From:" with the value "00002000" and "Bytes" with the value "FF". The fourth section, "Output Log File Name:", has a text field with the path "C:\EASy68K\Examples\tutorial2_out.txt" and a folder icon button. The fifth section, "Output Log Type", has two radio button options: "Disabled" (selected) and "Text Only". At the bottom right are "OK" and "Cancel" buttons.

Log Output

Execution Log File Name:
C:\EASy68K\Examples\tutorial2_reg.txt

Execution Log Type

- ☐ Disabled
- ☐ Instruction only
- ☐ Instruction and Registers
- ☒ Instruction, Registers and Memory

Memory Range to Log

From: 00002000 Bytes FF

Output Log File Name:
C:\EASy68K\Examples\tutorial2_out.txt

Output Log Type

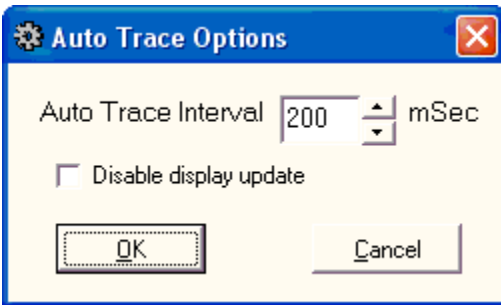
- ☒ Disabled
- ☐ Text Only

OK Cancel

- **Execution Log File Name:**
Select the name and location of the file to use when saving the instructions/registers of traced instructions. Click the Folder icon to browse. *Log data is only saved when tracing the program. No log data is saved when running the program at full speed.*
- **Execution Log Type**
Disabled - Logging is disabled.
Instruction only - Only the instruction is logged.
Instruction and Registers - The registers and the instruction are logged.
Instruction Registers and Memory - The registers, instruction and specified memory range are logged.
- **Memory Range to Log - Specify the range of memory to include in the log.**
From: Starting address to log (hexadecimal)
Bytes: Number of bytes to log (hexadecimal)
- **Output Log File Name:**
Select the name and location of the file to use when saving the output text from a program as it is traced. Click the Folder icon to browse. Output data is logged when tracing or running the program at full speed.
- **Output Log Type**
Disabled - Logging is disabled.
Text only - The text output from the traced program is logged.

Use  **Log Start** /  **Log Stop** from the Run menu or toolbar to start and stop logging. (See [Running A Program](#))

Auto Trace Options



Set the Auto Trace interval. The interval is in 1/1000's second so a value of 1000 is equivalent to 1 second.

When "Disable display update" is checked the source window is not updated after each trace. This results in a much faster trace operation and is intended for use when creating log files of traced code.

Sim68K - Exception Processing

Exception processing can begin in several different ways. Generally, there are three classes of exception conditions:

- Group 0: Reset, Address error, Bus error.
- Group 1: Trace, Interrupt, Illegal, Privilege violation.
- Group 2: TRAP, TRAPV, CHK, and Divide by zero.

A hardware reset may be generated by clicking the "Reset" button in the "Hardware" window.

Address error occurs when an `WORD_MASK` or long `WORD_MASK` is written or read from an odd `WORD_MASK` boundary. i.e. an odd memory address.

Bus error can occur in this simulator by attempting to read or write outside of the virtual memory space. The virtual memory for this simulator is from location 0 to location FFFFFFFF (hex).

Interrupt occurs when an external device interrupts the processor's operation. Interrupts may be created manually or automatically from the "Hardware" window..

Illegal exception occurs when an illegal instruction opcode is executed. It also occurs when the `ILLEGAL` instruction is executed.

Privilege violation occurs when a privileged instruction is attempted and the supervisor bit in the status register is not set.

TRAP exception occurs when a TRAP instruction is executed

TRAPV exception occurs when a TRAPV instruction is executed

CHK exception occurs when a CHK instruction is executed

Divide by zero exception occurs when a DIVU or DIVS instruction attempts a division by zero.

Exception processing begins by creating the appropriate exception stack frame for the particular exception group on the supervisor stack. Then, the supervisor mode is turned on and trace mode is turned off. After that, instruction execution resumes at the location referenced by the appropriate exception vector. The exception vector locations that can be used in this simulator are listed below:

Address

(Hex)	Assignment
000	Reset: Initial SSP
004	Reset: Initial PC
008	Bus error
00C	Address error
010	Illegal instruction
014	Divide by zero
018	CHK instruction
01C	TRAPV instruction
020	Privilege violation
024	Trace
028	Line 1010 emulator
02C	Line 1111 emulator
064	Level 1 Interrupt
068	Level 2 Interrupt
06C	Level 3 Interrupt
070	Level 4 Interrupt
074	Level 5 Interrupt
078	Level 6 Interrupt
07C	Level 7 Interrupt
080-0BC	TRAP instruction vectors

When the simulator starts up the supervisor bit is set on and the supervisor stack pointer is set to the value 1000000 (hex). Note that the stack grows downward, so the stack frame for any exceptions will grow from \$1000000 downward.

Sim68K Text I/O

TRAP #15 is used for I/O. Put the task number in D0.

Task

0	Display n characters of string at (A1), n is D1.W (stops on NULL or max 255) with CR, LF. (see task 13)
1	Display n characters of string at (A1), n is D1.W (max 255) without CR, LF. (see task 14)
2	Read string from keyboard and store at (A1), NULL (0) terminated, length returned in D1.W (max 80)
3	Display signed number in D1.L in decimal in smallest field. (see task 15 & 20)
4	Read a number from the keyboard into D1.L.
5	Read single ASCII character from the keyboard into D1.B.
6	Display single ASCII character in D1.B.
7	Check for keyboard input. Set D1.B to 1 if keyboard input is pending, otherwise set to 0. Use task 2 or 5 to read pending key.
8	Return time in hundredths of a second since midnight in D1.L.
9	Terminate the program. (Halts the simulator)
10	Print the NULL terminated string at (A1) to the default printer. (Always send a Form Feed character to end printing. See below.)
11	Cursor Position, Set/Get or Clear Screen Set cursor position: The high byte of D1.W holds the COL number (0-255), The low byte holds the ROW number (0-128). 0,0 is top left. Out of range coordinates are ignored. Get cursor position: Set D1.W to \$00FF Returns COL number, ROW number in high and low byte of D1.W respectively.

	<p>Clear Screen : Set D1.W to \$FF00. (Task 95 supports exact pixel placement of text)</p>
12	<p>Keyboard Echo. D1.B = 0 to turn off keyboard echo. D1.B = non zero to enable. (default). Echo is restored on 'Reset' or when a new file is loaded.</p>
13	<p>Display the NULL terminated string at (A1) with CR, LF.</p>
14	<p>Display the NULL terminated string pointed to by (A1). Example:</p> <pre> START ORG \$1000 Program load address. move #14,D0 D0 = task number 14. lea text,A1 A1 = address of string to display. trap #15 Activate input/output task. SIMHALT Halt program execution. text dc.b 'Hello World',0 Null terminated string. END START End of source with start address specified. </pre>
15	<p>Display the unsigned number in D1.L converted to number base (2 through 36) contained in D2.B. For example, to display D1.L in base16 put 16 in D2.B Values of D2.B outside the range 2 to 36 inclusive are ignored.</p>
16	<p>Adjust display properties D1.B = 0 to turn off the display of the input prompt. D1.B = 1 to turn on the display of the input prompt. (default) D1.B = 2 do not display a line feed when Enter pressed during Trap task #2 input D1.B = 3 display a line feed when Enter key pressed during Trap task #2 input (default) Other values of D1 reserved for future use. Input prompt display is enabled by default and by 'Reset' or when a new file is loaded.</p>

17	Combination of Trap codes 14 & 3. Display the NULL terminated string at (A1) without CR, LF then Display the decimal number in D1.L.
18	Combination of Trap codes 14 & 4. Display the NULL terminated string at (A1) without CR, LF then Read a number from the keyboard into D1.L.
19	<p>Returns current state of up to 4 specified keys or returns key scan code. Pre: D1.L = four 1-byte key_codes Post: D1.L contains four 1-byte Booleans. \$FF = corresponding key is pressed, \$00 = corresponding key not pressed. Pre: D1.L = \$00000000 Post: D1.L upper word contains key code of last key up. D1.L lower word contains key code of last key down.</p> <p>Example:</p> <pre> MOVE.B #19,D0 MOVE.L #'A'<<24+'S'<<16+'D'<<8+'F',D1 ; check for keypress (a,s,d,f) TRAP #15 BTST.L #24,D1 ; test for 'a' IF <NE> THEN ; if 'a' {a code} ENDI BTST.L #16,D1 ; test for 's' IF <NE> THEN ; if 's' ... etc </pre>
20	Display signed number in D1.L in decimal in field D2.B columns wide.
21	Set font properties where: D1.L is color as 0x00BBGGRR BB is amount of blue from 0x00 to 0xFF GG is amount of green from 0x00 to 0xFF RR is amount of red from 0x00 to 0xFF D2.L Low word is style by bits 0 = off, 1 = on

	<p>bit0 is Bold bit1 is Italic bit2 is Underline bit3 is StrikeOut</p> <p>High word (low byte) is Size in points (High word = 0, keep current font)</p> <p>8, 9, 10, 11, 12, 14, 16, 18 (not all sizes are valid for all fonts)</p> <p>Font sizes in multiples of valid sizes (size * n) results in a scaled appearance. For example: in Fixedsys font sizes of 9*2, 9*3, ..9*n will result in larger characters but the characters will have pixelated edges.</p> <p>High word (high byte) is Font</p> <p>1 - Fixedsys (size: 9) 2 - Courier (sizes: 10, 12, 15) 3 - Courier New (sizes 8,9,10,11,12,14,16,18) 4 - Lucida Console (sizes 8,9,10,11,12,14,16,18) 5 - Lucida Sans Typewriter (sizes 8,9,10,11,12,14,16,18) 6 - Consolas (sizes 8,9,10,11,12,14,16,18) 7 - Terminal (sizes 9,12,14)</p> <p>Example: D2.L = \$01090005 is Fixedsys, 9 point, Bold Underline</p>
22	<p>Read char at Row,Col of text screen.</p> <p>Pre: D1.L = High 16 bits = Row Low 16 bits = Column</p> <p>Post: D1.B contains ASCII code of character.</p>
23	<p>Delay n/100 of a second</p> <p>Pre: D1.L = n as unsigned number 0 through \$FFFFFFFF</p> <p>Delay releases CPU which reduces power consumption.</p>
24	<p>Text I/O control</p> <p>Pre: D1.L = 0, Enable simulator shortcut keys. (default) D1.L = 1, Disable simulator shortcut keys.</p> <p>All key codes are made available for 68000 program read using task 19. all other values reserved.</p> <p>Shortcuts are restored by Rewind or Reload.</p>
25	<p>Scroll Text Rectangle</p> <p>Pre: D1.L = High 16 bits = Top row (0 to 128) Low 16 bits = Left column (0 to 255)</p> <p>D2.L = High 16 bits = Height in rows (Top + Height max 128) Low 16 bits = Width in columns (Left + Width max 255)</p>

	D3.W = 0 scroll up D3.W = 1 scroll down D3.W = 2 scroll left D3.W = 3 scroll right all other values reserved
--	--

Task numbers 0 - 9 and 11 - 12 are compatible with the Teesside simulator.

The following control characters are supported. The labels shown (BEL, BS, HT, etc.) are not predefined in EASy68K. Placing the code in your program will equate the labels with the control characters.

BEL	EQU	\$07	Bell
BS	EQU	\$08	Backspace
HT	EQU	\$09	Tab (horizontal 5 characters)
LF	EQU	\$0A	Line Feed
VT	EQU	\$0B	Vertical tab (4 lines)
FF	EQU	\$0C	Form Feed (Always end printing with a Form Feed.)
CR	EQU	\$0D	Carriage Return

Sim68K Environment

TRAP #15 is used for I/O. Put the task number in D0.

Task

30	Clear the Cycle Counter
31	Return the Cycle Counter in D1.L Zero is returned if the cycle count exceeds 32 bits.
32	<p>Hardware/Simulator</p> <p>D1.B = 00, Display hardware window</p> <p>D1.B = 01, Return address of 7-segment display in D1.L</p> <p>D1.B = 02, Return address of LEDs in D1.L</p> <p>D1.B = 03, Return address of toggle switches in D1.L</p> <p>D1.B = 04, Return Sim68K version number in D1.L Version 3.9.10 is returned as 0003090A</p> <p>D1.B = 05, Enable exception processing. Exceptions will be directed to the appropriate 68000 exception vector.</p> <p>This has the same effect as checking Enable Exceptions in the Options menu.</p> <p>D1.B = 06, Set Auto IRQ</p> <p>D2.B = 00, disable all Auto IRQs or Bit 7 = 0, disable individual IRQ Bit 7 = 1, enable individual IRQ Bits 6-0, IRQ number 1 through 7</p> <p>D3.L, Auto Interval in milliseconds, 1000 = 1 second</p> <p>D1.B = 07, Return address of push button switches in D1.L</p>
33	<p>Get/Set Output Window</p> <p>D1.L High 16 bits = Set width in pixels, min = 640 Low 16 bits = Set height in pixels, min = 480</p> <p>D1.L = 0, Get current output window resolution in D1.L as High 16 bits = Width in pixels Low 16 bits = Height in pixels</p>

D1.L = 1, Set windowed mode
D1.L = 2, Set full screen mode

Example:

```
        MOVE.B #33,D0
        MOVE.L #1024*$10000+768,D1          Set screen to
1024 x 768
        TRAP    #15
```


Sim68K Serial I/O

TRAP #15 is used for I/O. Put the task number in lower 8 bits of D0.

The success of the Serial I/O calls is returned in D0.W as follows:

- 0 = Success
- 1 = Invalid port identifier (PID)
- 2 = Error
- 3 = Port not initialized (Tasks 41, 42, 43 only)
- 4 = Timeout (Tasks 42, 43 only)

A maximum of 16 communications ports are supported.

Task

40	<p>Initialize communications port. This must be called once for each port before any other serial I/O function.</p> <p>The port defaults to 9600 baud, 8 data bits, no parity, one stop bit.</p> <p>Pre:</p> <ul style="list-style-type: none">High 16 bits of D0 contain port identifier (PID), (0 through 15). The PID is used to identify the port in all other serial I/O tasks. It is not used to specify the COM port number, (i.e. A PID of 4 does not mean the same thing as COM4)(A1) address of serial port name as null terminated string (e.g. PORT DC.B 'COM4',0) <p>Post:</p> <ul style="list-style-type: none">D0.W is 0 on success, 1 on invalid PID, 2 on error <p>Example:</p> <pre style="background-color: #f0f0f0; padding: 10px;">; initialize serial port move.l #1<<16+40,d0 ; PID 1, task 40 lea PORT,A1 ; name of port trap #15</pre> <p>Visit www.EASy68K.com for examples.</p>
41	<p>Set port parameters.</p> <p>Pre:</p> <ul style="list-style-type: none">High 16 bits of D0 contain port identifier (PID)D1.L

	<p>Bits 0-7 (D1.B)</p> <p>Baud rate: 0=9600(default), 1=110, 2=300, 3=600, 4=1200, 5=2400, 6=4800, 7=9600</p> <p>8=19200, 9=38400, 10=56000, 11=57600, 12=115200, 13=128000, 14=256000.</p> <p>Bits 8-9</p> <p>Parity: 0=no, 1=odd, 2=even, 3=mark</p> <p>Bits 10-11</p> <p>Number of data bits: 0=8 bits, 1=7 bits, 2=6 bits</p> <p>Bit 12</p> <p>Stop bits: 0=1 stop bit, 1=2 stop bits</p> <p>The higher bits of D1.L are reserved for future use.</p> <p>Post:</p> <p>D0.W is 0 on success, 1 on invalid PID, 2 on error, 3 on port not initialized</p>
42	<p>Receive string.</p> <p>Pre:</p> <p>High 16 bits of D0 contain port identifier (PID)</p> <p>(A1) buffer address.</p> <p>D1.B max number of characters to receive. The port will wait sufficient time for the requested number of bytes to be received. A timeout will occur if the number of bytes received is less than the requested number. For faster response use a smaller number in D1.B.</p> <p>Post:</p> <p>D0.W is 0 on success, 1 on invalid PID, 2 on error, 3 on port not initialized, 4 on timeout</p> <p>D1.B number of characters received.</p> <p>(A1) null terminated string of characters received.</p>
43	<p>Send string.</p> <p>Pre:</p> <p>High 16 bits of D0 contain port identifier (PID)</p> <p>(A1) buffer address.</p> <p>D1.B number of characters to send. The port will wait sufficient time for the specified number of bytes to be sent. A timeout will occur if the number of bytes sent is less than the requested number.</p> <p>Post:</p> <p>D0.W is 0 on success, 1 on invalid PID, 2 on error, 3 on port not initialized, 4 on timeout</p> <p>D1.B number of characters sent.</p>

Sim68K File I/O

TRAP #15 is used for I/O. Put the task number in D0.

The success of the file handling calls is returned in D0.W as follows:

- 0 = success
- 1 = EOF encountered
- 2 = Error
- 3 = File is Read Only (Task 51 & 59 only)

A maximum of 8 files may be open at any one time.

Task

50	Close all files. Finalizes file operations. Writes pending data to files. It is recommended to use this at the start of any program using file handling.
51	Open existing file. Pre: (A1) null terminated file name. Post: D1.L contains the File-ID (FID).
52	Open new file.. As above except the file is created if not found. If the file exists it will be overwritten.
53	Read file Pre: File-ID in D1.L as returned from 51 or 52, (A1) buffer address, D2.L number of bytes to read. Post: D2.L holds number of bytes actually read, EOF may cause a shortened read.
54	Write file As above except D2.L holds number of bytes to write (unaltered upon

	<p>return).</p> <p>The write operation is not completed until the file is closed using 50 or 56.</p>
55	<p>Position file</p> <p>Sets the file position where the next read/write will take place. Files begin at byte 0.</p> <p>Pre: File-ID in D1.L as returned from 51 or 52, D2.L the file position to be set.</p>
56	<p>Close file</p> <p>Finalizes file operations. Writes pending data to file.</p> <p>Pre: File-ID in D1.L as returned from 51 or 52.</p>
57	<p>Delete file.</p> <p>Pre: (A1) null terminated file name.</p>
58	<p>Display File Dialog.</p> <p>Pre: D1 = 0 to display 'Open' dialog D1 = 1 to display 'Save' dialog (A1) Points to a null terminated string that will be used as the request title string (max 256) or A1 = \$00000000 to use the default 'Open' or 'Save' depending on D1 (A2) Points to a null terminated string (max 256) containing semicolon separated list of file extensions for use in dialog, e.g. '*.txt;*.pcb',0 or A2 = \$00000000 for all file types. (A3) Points to a buffer of sufficient size (max 256) to hold the zero terminated full file path and name. On calling, if this contains a file path and name this will be used as the original file path and name. If the user exits via the cancel button this buffer will remain unchanged.</p> <p>Post: D1 = 0 if user cancelled or just closed D1 = 1 if user hit open/save</p>
59	<p>File operations.</p> <p>Pre: (A1) null terminated file name. D1.L = 0 does file exist?</p>

Post: D0.W 0 = file exists

2 = Error

3 = file exists and is Read Only

Other values of D1 reserved for future use.

Sim68K Peripheral I/O

TRAP #15 is used for I/O. Put the task number in D0.

The output screen resolution is adjustable via Task 33. The top left corner of the window is position 0,0

Task

60	<p>Enable/Disable mouse IRQ</p> <p>An IRQ is created when a mouse button is pressed or released in the output window.</p> <p>D1.W High Byte = IRQ level (1-7), 0 to turn off</p> <p>D1.W Low Byte = 1 in the corresponding bit to indicate which mouse event triggers IRQ where:</p> <p style="padding-left: 40px;">Bit2 = Move, Bit1 = button Up, Bit0 = button Down</p> <p>(Example D1.W = \$0107, Enable mouse IRQ level 1 for Move, button Up and button Down)</p> <p>(Example D1.W = \$0002, Disable mouse IRQ for button Up)</p>
61	<p>Mouse Read</p> <p>Pre: D1.B = 00 to read current state of mouse</p> <p style="padding-left: 40px;">= 01 to read mouse up state</p> <p style="padding-left: 40px;">= 02 to read mouse down state</p> <p>Post: The mouse data is contained in the following registers:</p> <p style="padding-left: 40px;">D0 as bits = Ctrl, Alt, Shift, Double, Middle, Right, Left</p> <p style="padding-left: 40px;">Left is Bit0, Right is Bit 1 etc. 1 = true, 0 = false</p> <p style="padding-left: 40px;">Shift, Alt, Ctrl represent the state of the corresponding keys.</p> <p style="padding-left: 40px;">D1.L = 16 bits Y, 16 bits X in pixel coordinates. (0,0 is top left)</p>
62	<p>Enable/Disable keyboard IRQ</p> <p>An IRQ is created when a key is pressed or released in the output window.</p> <p>D1.W High Byte = IRQ level (1-7), 0 to turn off</p> <p>D1.W Low Byte = 1 in the corresponding bit to indicate which keyboard event triggers IRQ where:</p> <p style="padding-left: 40px;">Bit1 = key Up, Bit0 = key Down</p> <p>(Example D1.W = \$0103, Enable keyboard IRQ level 1 for key Up and key Down)</p>

(Example D1.W = \$0002, Disable keyboard IRQ for key Up)
Read the last key down or key up with trap [task #19](#).

Sim68K Sound

TRAP #15 is used for I/O. Put the task number in D0.

Task

70	Play the WAV file using the standard player. Only one sound may be played at a time. Pre: (A1) null terminated path address. Invalid file names are ignored. Post: D0.W = 0 if player is busy, sound is not played. D0.W = non zero if sound played.
71	Load a WAV file into sound memory (not 68000 memory). Pre: (A1) null terminated path address. Invalid file names are ignored. D1.B reference number to use for sound 0-255. A maximum of 256 sounds may be loaded at any one time. Reusing a reference number will replace the current sound.
72	Play sound from sound memory loaded with task 71 using standard player. Only one sound may be played at a time. Pre: D1.B must contain sound reference number used in task 71. Post: D0.W = 0 if player is busy, sound is not played. D0.W = non zero if sound played.
73	Play the WAV file using DirectX player, if available. Multiple sounds may be played at the same time. Pre: (A1) null terminated path address. Invalid file names are ignored. Post: D0.W = 0 if DirectX player not available, sound is not played. D0.W = non zero if sound played.
74	Load a WAV file into DirectX sound memory (not 68000 memory). A maximum of 256 sounds may be loaded at any one time. Reusing a reference number will replace the current sound. Pre: (A1) null terminated path address. Invalid file names are ignored. D1.B reference number to use for sound 0-255.

	<p>Post: D0.W = 0 if DirectX player not available. D0.W = non zero if sound loaded.</p>
75	<p>Play sound from DirectX sound memory loaded with task 74. Pre: D1.B must contain sound reference number used in task 74. Post: D0.W = 0 if DirectX player not available, sound is not played. D0.W = non zero if sound played.</p>
76	<p>Control Standard player Sounds must be in memory loaded with task 71. Only one sound may be played at a time. Pre: D1.B contains sound reference number used in task 71. D2.L = 0, play sound once (this is the same as task 72) D2.L = 1, play sound in loop, returns error if sound currently playing. D2.L = 2, stop D1.B referenced sound, returns error on bad reference number D2.L = 3, stop all sounds, returns success (D1.B ignored) D2.L = other values reserved Post: D0.W = 0 on error. D0.W = non zero on success.</p>
77	<p>Control DirectX player, if available Sounds must be in DirectX memory loaded with task 74. Multiple sounds may be played at the same time. Pre: D1.B contains sound reference number used in task 74. D2.L = 0, play sound once (this is the same as task 75) D2.L = 1, play sound in loop. The same sound may be played multiple times. D2.L = 2, stop D1.B referenced sound, returns error on bad reference number D2.L = 3, stop all sounds (D1.B ignored) D2.L = other values reserved Post: D0.W = 0 if DirectX player not available. D0.W = non zero on success.</p>

Sim68K Graphics

TRAP #15 is used for I/O. Put the task number in D0.

The output screen resolution is adjustable via Task 33. The top left corner of the window is position 0,0

Drawing outside the screen area is ignored.

The screen may be cleared with task 11.

Task

80	Set pen color where D1.L is color as \$00BBGGRR BB is amount of blue from \$00 to \$FF GG is amount of green from \$00 to \$FF RR is amount of red from \$00 to \$FF
81	Set fill color where D1.L is color as \$00BBGGRR
82	Draw pixel in pen color at X,Y where X = D1.W & Y = D2.W The drawing mode is not used (see 92) . The drawing point is not moved (see 86).
83	Get pixel color at X,Y and place in D0.L where X = D1.W & Y = D2.W.
84	Draw line using pen color from X1,Y1 to X2,Y2 where X1 = D1.W, Y1 = D2.W, X2 = D3.W, Y2 = D4.W
85	Draw line using pen color to X,Y where X = D1.W & Y = D2.W
86	Move to X,Y where X = D1.W & Y = D2.W (Task 84 & 85 also move drawing point)
87	Draw rectangle defined by (Left X, Upper Y, Right X, Lower Y).

	<p>where $LX = D1.W$, $UY = D2.W$, $RX = D3.W$, $LY = D4.W$ The rectangle is drawn using pen color and filled using fill color.</p>
88	<p>Draw ellipse bounded by the rectangle (Left X, Upper Y, Right X, Lower Y) where $LX = D1.W$, $UY = D2.W$, $RX = D3.W$, $LY = D4.W$ The ellipse is drawn using pen color and filled using fill color. A circle is drawn if the bounding rectangle is a square.</p>
89	<p>Flood Fill the area at X, Y with the fill color where $X = D1.W$ & $Y = D2.W$</p>
90	<p>Draw unfilled rectangle defined by (Left X, Upper Y, Right X, Lower Y). where $LX = D1.W$, $UY = D2.W$, $RX = D3.W$, $LY = D4.W$ The rectangle is drawn using pen color and is not filled.</p>
91	<p>Draw unfilled ellipse bounded by the rectangle (Left X, Upper Y, Right X, Lower Y) where $LX = D1.W$, $UY = D2.W$, $RX = D3.W$, $LY = D4.W$ The ellipse is drawn using pen color and is not filled. A circle is drawn if the bounding rectangle is a square.</p>
92	<p>Set drawing mode which affects pen and fill colors. D1.B is mode number as:</p> <ul style="list-style-type: none"> 0 - Draw color is always black, ignores pen and fill colors 1 - Draw color is always white, ignores pen and fill colors 2 - Move cursor but do not draw 3 - NOT (background color) is drawn 4 - The draw color is drawn (default) 5 - NOT (draw color) is drawn 6 - NOT (background color) OR (draw color) is drawn 7 - NOT (background color) AND (draw color) is drawn 8 - (background color) OR NOT (draw color) is drawn 9 - (background color) AND NOT (draw color) is drawn 10 - (background color) OR (draw color) is drawn 11 - (background color) NOR (draw color) is drawn 12 - (background color) AND (draw color) is drawn 13 - (background color) NOT AND (draw color) is drawn 14 - (background color) XOR (draw color) is drawn

	15 - (background color) NOT XOR (draw color) is drawn 16 - Turn off double buffering (default) 17 - Enable double buffering. Draw on off screen buffer (Use Task 94 to display off screen buffer) * The background color is whatever is already there. * The draw color is the pen color or the fill color.
93	Set pen width where D1.B is width in pixels.
94	Repaint screen. Copies off screen buffer to visible screen. (Requires drawing mode 17 be set from task 92 above.)
95	Draw the NULL terminated string of text at (A1) to screen location X,Y where X = D1.W & Y = D2.W. The text is drawn as graphics and may not be read using trap task 22. Control characters are ignored. X,Y specifies the location of the top left corner of the text.
96	Get X,Y pen position where D1.W = X, D2.W = Y

Some color values:

```

BLACK    equ $00000000
MAROON   equ $00000080
GREEN    equ $00008000
OLIVE    equ $00008080
NAVY     equ $00800000
PURPLE   equ $00800080
TEAL     equ $00808000
GRAY     equ $00808080
RED      equ $000000FF
LIME     equ $0000FF00
YELLOW   equ $0000FFFF
BLUE     equ $00FF0000
FUCHSIA  equ $00FF00FF
AQUA     equ $00FFFF00
LTGRAY   equ $00C0C0C0

```

```
WHITE    equ $00FFFFFF
```

Sim68K Network I/O

EASy68K supports both TCP and UDP communications.

TCP

TCP (Transmission Control Protocol) provides reliable delivery of data. Large messages are automatically broken into smaller IP-sized packets, transmitted, received and reassembled in order by the receiver. Packets that are lost during transmission are automatically retransmitted. The reliability of TCP message delivery can result in long delays (on the order of seconds) if packets are lost and must be retransmitted. For this reason TCP is typically not considered suitable for the delivery of real-time data. TCP establishes a connection between the two devices. This connection is maintained until broken by one or both devices. In EASy68K only one TCP connection is permitted at a time. If the other host breaks the connection an error will result when attempting to send or receive data. Refer to the error codes below.

UDP

UDP (User Datagram Protocol) is a connectionless protocol that does not guarantee data delivery. Data packets may arrive out of order, be duplicated, or not delivered at all. UDP is typically used for time sensitive applications where timely delivery of data is more important than reliability. Because of it's connectionless nature a UDP server may receive transmissions from multiple UDP clients.

EASy68K uses non-blocking sockets which means all of the network trap tasks return immediately. Check the Post register contents to determine the results.

TRAP #15 is used for I/O. Put the task number in lower 8 bits of D0.

Task

100	Configure as network Client. Pre: D1.L {31.....16}{15.....8}{7.....0} Bits 0-7, 0 for UDP, 1 for TCP, all other values reserved Bits 8-15, Reserved for future use Bits 16-31, Port number Port numbers 0-1023 are used for well-known services. Port
-----	---

numbers 1024-65535 may be freely used.

(A2) IP address to connect to as null terminated string (e.g. '192.168.1.100',0) or

null terminated domain name (e.g. 'www.easy68k.com',0)

Post:

D0.L is 0 on success, non zero on error

Error codes:

Bits 0-15 Low word of D0

1 - general error

2 - network initialization failed

3 - invalid socket

4 - get host by name failed

5 - bind failed

6 - connect failed

7 - port already in use

8 - domain not found

All other values reserved

Bits 16-31 High word of D0, extended error code (see below)

(A2) IP address connected to as null terminated string

Example:

```
client      move.b    #100,d0                ; create network
            move.l    #(2048 << 16 + 0),d1    ; port 2048, UDP
            lea        serverIP,a2           ; IP to connect
            trap       #15
            ...
serverIP    dc.b      '192.168.1.101',0
```

Visit www.EASy68K.com for examples.

101 Configure as network Server.

May not be configured as Server and Client at the same time.

Pre:

D1.L {31.....16}{15.....8}{7.....0}

Bits 0-7, 0 for UDP, 1 for TCP, all other values reserved

Bits 8-15, Reserved for future use

Bits 16-31, Port number

Port numbers 0-1023 are used for well-known services. Port numbers 1024-65535 may be freely used.

	<p>Post:</p> <p>D0.L is 0 on success, non zero on error</p> <p>Error codes:</p> <p>Bits 0-15 Low word of D0</p> <ul style="list-style-type: none"> 1 - general error 2 - network initialization failed 3 - invalid socket 4 - get host by name failed 5 - bind failed 6 - connect failed 7 - port already in use 8 - domain not found <p>All other values reserved</p> <p>Bits 16-31 High word of D0, extended error code (see below)</p>
102	<p>Send data. (Deprecated, use 106)</p> <p>Pre:</p> <p>D1.L {31.....16}{15.....0}</p> <p>Bits 0-15, Number of bytes to send</p> <p>Bits 16-31, Reserved for future use</p> <p>(A1) data to send</p> <p>If server</p> <p>(A2) IP address of client as null terminated string (e.g. '192.168.1.100',0)</p> <p>Post:</p> <p>D0.L is 0 on success, non zero on error. Success does not indicate data was sent.</p> <p>Bits 0-15 Low word of D0</p> <ul style="list-style-type: none"> 1 - send failed <p>All other values reserved</p> <p>Bits 16-31 High word of D0, extended error code (see below)</p> <p>D1.L number of bytes sent. 0 if no data was sent. Unchanged on error.</p>
103	<p>Receive data. (Deprecated, use 107)</p> <p>Pre:</p> <p>D1.L {31.....16}{15.....0}</p> <p>Bits 0-15, Number of bytes to receive.</p> <p>Bits 16-31, Reserved for future use.</p> <p>(A1) received buffer, must be large enough to hold D1.W bytes.</p> <p>Post:</p> <p>D0.L is 0 on success, non zero on error. Success does not indicate data was received.</p> <p>Bits 0-15 Low word of D0</p> <ul style="list-style-type: none"> 1 - receive failed

	<p>All other values reserved</p> <p>Bits 16-31 High word of D0, extended error code (see below)</p> <p>D1.L number of bytes received. 0 if no data has been received.</p> <p>Unchanged on error.</p> <p>(A2) IP address of sender as null terminated string.</p>
104	<p>Close connections</p> <p>Pre:</p> <p> If TCP server</p> <p> D1.L IP address of connection to close, 0 to close all.</p> <p>Post:</p> <p> D0.L is 0 on success, non zero on error</p> <p> Bits 0-15 Low word of D0</p> <p> 1 - close failed</p> <p> All other values reserved</p> <p> Bits 16-31 High word of D0, extended error code (see below)</p>
105	<p>Get local IP address</p> <p>Pre:</p> <p> The network has been initialized with task 100 or 101</p> <p>Post:</p> <p> D0.L is 0 on success, non zero on error</p> <p> Bits 0-15 Low word of D0</p> <p> 1 - get local IP failed</p> <p> All other values reserved</p> <p> Bits 16-31 High word of D0, extended error code (see below)</p> <p> (A2) local IP address as null terminated string. Max size 16 characters including null.</p>
106	<p>Send data on specified port.</p> <p>Pre:</p> <p> D1.L {31.....16}{15.....0}</p> <p> Bits 0-15, Number of bytes to send</p> <p> Bits 16-31, Port number</p> <p> Port numbers 0-1023 are used for well-known services. Port numbers 1024-65535 may be freely used.</p> <p> (A1) Data to send</p> <p> If server</p> <p> (A2) IP address of client as null terminated string (e.g. '192.168.1.100',0)</p> <p>Post:</p> <p> D0.L {31.....16}{15.....0}</p> <p> Bits 0-15, 0 on success, non zero on error. Success does not indicate data was sent.</p> <p> 1 - Send failed</p>

	<p>All other values reserved</p> <p>Bits 16-31 Extended error code (see below)</p> <p>D1.L {31.....16}{15.....0}</p> <p>Bits 0-15, Number of bytes received. 0 if no data has been received.</p> <p>Unchanged on error.</p> <p>Bits 16-31, Reserved for future use</p>
107	<p>Receive data and port number</p> <p>Pre:</p> <p>D1.L {31.....16}{15.....0}</p> <p>Bits 0-15, Number of bytes to receive.</p> <p>Bits 16-31, Reserved for future use.</p> <p>(A1) received buffer, must be large enough to hold D1.W bytes.</p> <p>Post:</p> <p>D0.L {31.....16}{15.....0}</p> <p>Bits 0-15, 0 on success, non zero on error. Success does not indicate data was received.</p> <p>1 - receive failed</p> <p>All other values reserved</p> <p>Bits 16-31 High word of D0, extended error code (see below)</p> <p>D1.L {31.....16}{15.....0}</p> <p>Bits 0-15, Number of bytes received. 0 if no data has been received.</p> <p>Unchanged on error.</p> <p>Bits 16-31, Port number of received data</p> <p>(A2) IP address of sender as null terminated string.</p>

Extended Error Codes in high word of D0

\$2714 - A blocking operation was interrupted

\$271D - Socket access permission violation

\$2726 - Invalid argument

\$2728 - Too many open sockets

\$2735 - Operation in progress

\$2736 - Operation on non-socket

\$2737 - Address missing

\$2738 - Message bigger than buffer

\$273F - Address incompatible with protocol

\$2740 - Address is already in use

\$2741 - Address not valid in current context

\$2742 - Network is down

\$2743 - Network unreachable

\$2744 - Connection broken during operation

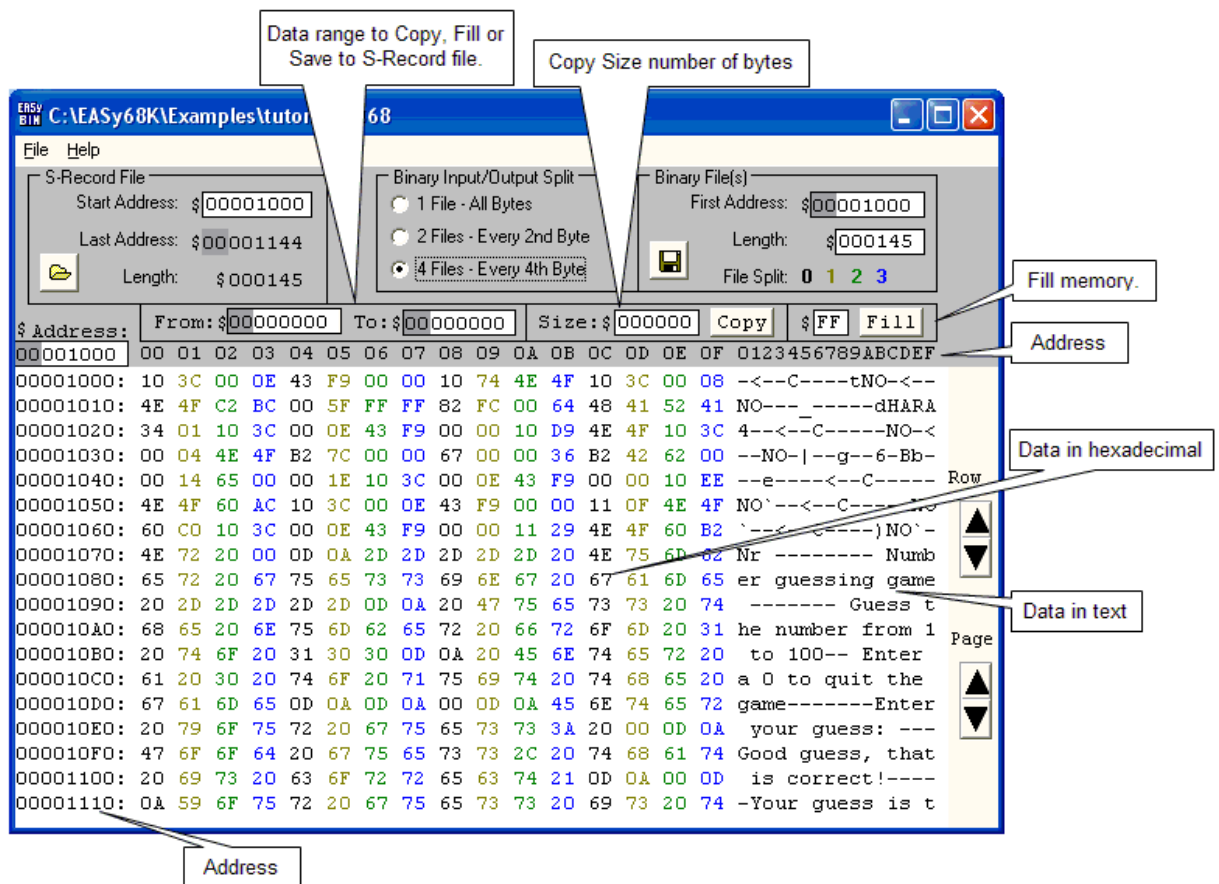
\$2745 - Connection aborted by host software

\$2746 - Connection reset by remote host
\$2747 - Insufficient buffer space
\$2748 - Connect request on already connected socket
\$2749 - Socket not connected or address not specified
\$274A - Socket already shut down
\$274C - Operation timed out
\$274D - Connection refused by target
\$274E - Cannot translate name
\$274F - Name too long
\$2750 - Destination host down
\$2751 - Host unreachable
\$276B - Network cannot initialize, system unavailable
\$276D - Network has not been initialized
\$2775 - Remote has disconnected

(Refer to Winsock file winerror.h for unlisted error codes)

EASyBIN - Help

EASyBIN is an S-Record and binary file utility designed for use with EASy68K. S-Record and binary files may be viewed, edited and created. The data is displayed in both hexadecimal and text and may be edited in either form. Files may be created from any portion of the data. EASyBIN supports editing files up to 16 MBytes in size, the full address space of the 68000 microprocessor.



Each row displays an address followed by 16 bytes of hexadecimal data, followed by the ASCII representation of the 16 bytes. The contents of memory may be changed by clicking on the desired location and entering a new value. Entries may be made in Hexadecimal or ASCII. Use the **Row** and **Page** buttons or the mouse

wheel to scroll up or down through memory. To jump to a certain address, enter it in the **Address:** field.

=== The Main Screen ===

(Note! All numbers are displayed in hexadecimal)

S-Record File

Start Address: \$00001000

(For Open) The starting execution address as specified in the S-Record file's S7, S8 or S9 record.

(For Save) The starting execution address saved to the S-Record file as an S7 (4 byte address) record.

Last Address: \$00001144

(For Open) The last address of the data loaded from the S-Record file.

Length: \$000145

(For Open) The number of bytes loaded from the S-Record file.

 Open S-Record

Binary Output Split (For Binary Open and Save)

1 File - All Bytes

(For Save) One binary file is created with the specified data.

(For Open) The binary file is loaded into consecutive memory locations.

2 Files - Every 2nd Byte

(For Save) Two binary files are created. The first data byte is written to file_0 the second data byte is written to file_1 the third data byte is written to file_0 the fourth data byte is written to

file_1 etc. until the end of the output data range is reached.
(For Open) The binary file is loaded into every 2nd byte in memory.

4 Files - Every 4th Byte

(For Save) Four binary files are created. The first data byte is written to file_0 the second data byte is written to file_1 the third data byte is written to file_2 the fourth data byte is written to file_3. The sequence repeats until the end of the output data range is reached.

(For Open) The binary file is loaded into every 4th byte in memory.

Binary File(s) (For Binary Open and Save)

First Address: \$00001000

(For Open) The address to begin loading the binary data.

(For Save) The address of the first byte to save.

Length: \$000145

(For Save) The number of bytes to save. All the data bytes outside the range specified by First Address and Length are grayed out to indicate the bytes that will be written to a binary file. The grayed out data may still be written to an S-Record file.

File Split: 0 1 2 3

(For Save) Indicates the number of binary files that will be created. The color of the digit and the color of the hexadecimal data reflects how the data will be split into each file.

 Save binary file(s).

From: To:

From: \$00000000 To: \$00000000

(For Copy) From: is the location of the source data. To: is the

location of the destination.

(For Fill) Specifies the memory range to fill.

(For S-Record Save) Specifies the memory range to save to the S-Record file.

Copy:

Size: \$000000

The number of bytes to copy.

Fill:

\$FF

The data to fill the specified memory range with.

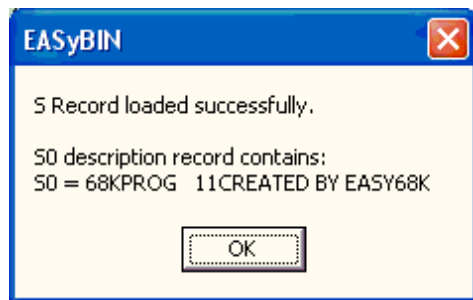
=== The Main Menu ===

Some menu items may be invoked with keyboard shortcuts. The keyboard shortcuts are displayed to the right of the command in the menu. In the File menu: below the keyboard shortcut for Open S-Record... is Ctrl+O. This means pressing the letter 'O' while holding the Ctrl (Control) key will perform the Open File command. Not all menu commands have a keyboard shortcut.

File menu:

New - Fills all memory with specified Fill byte.

Open S-Record... Ctrl+O - Loads an existing S-Record file. If the file loads successfully the contents of the S0 description record from the file are displayed in a dialog box.



Open Binary... Ctrl+B - Loads an existing Binary file. The file is loaded into memory beginning at First Address: as specified in the Binary File(s) area.

Save S-Record... Ctrl+R - Save the data specified by From: and To: addresses to an S-Record file.

Save Binary... Ctrl+B - Save Length: bytes of data starting at First Address to a Binary file. If Binary Output Split is used to split the data multiple file names are created by appending _0, _1, _2 or _3 to the specified file name.

Exit - Exits the program.

Motorola S-records in EASy68K

DESCRIPTION

An S-record file consists of a sequence of specially formatted ASCII character strings. An S-record will be less than or equal to 78 bytes in length.

The original Unix man page on S-records is the only place that a 78-byte limit on total record length or 64-byte limit on data length is documented. These values should not be trusted for the general case. Always assume that a record can be as long as 514 (decimal) characters in length ($255 * 2 = 510$, plus 4 characters for the type and count fields), plus any terminating character(s). Input buffers should be long enough to hold 515 chars, thus leaving room for the terminating null character. The order of S-records within a file is of no significance and no particular order may be assumed with the exception of the termination record at the end.

The general format of an S-record follows:

```
+-----//-----//---
-----+
| type | count | address | data
| checksum |
```

+-----//-----//---
-----+

type -- A char[2] field. These characters describe the type of record (S0, S1, S2, S3, S5, S7, S8, or S9).

count -- A char[2] field. These characters when paired and interpreted as a hexadecimal value, display the count of remaining character pairs in the record.

address -- A char[4,6, or 8] field. These characters grouped and interpreted as a hexadecimal value, display the address at which the data field is to be loaded into memory. The length of the field depends on the number of bytes necessary to hold the address. A 2-byte address uses 4 characters, a 3-byte address uses 6 characters, and a 4-byte address uses 8 characters.

data -- A char [0-64] field. These characters when paired and interpreted as hexadecimal values represent the memory loadable data or descriptive information.

checksum -- A char[2] field. These characters when paired and interpreted as a hexadecimal value display the least significant byte of the ones complement of the sum of the byte values represented by the pairs of

characters making up the count, the address, and the data fields.

Each record is terminated with a line feed. If any additional or different record terminator(s) or delay characters are needed during transmission to the target system it is the responsibility of the transmitting program to provide them.

S0 Record. The address field is unused and will be filled with zeros (0x0000). The code/data field contains descriptive information identifying the following block of S-records as having been created by EASy68K. Beginning with EASy68K v5.0.0 the memory map information generated with the MEMORY directive is saved in S0 records.

S1 Record. The address field is interpreted as a 2-byte address. The data field is composed of memory loadable data.

S2 Record. The address field is interpreted as a 3-byte address. The data field is composed of memory loadable data.

S3 Record. The address field is interpreted as a 4-byte address. The data field is composed of memory loadable data.

S5 Record. The address field is interpreted as a 2-byte value and contains the count of S1, S2, and S3 records previously transmitted. There is no data field. (EASy68K does not generate this record.)

S7 Record. Termination record. The address field contains the starting execution address and is interpreted as 4-byte address. There is no data field. This is the last record in the file. (EASyBIN always uses this termination record when creating S-Record files.)

S8 Record. Termination record. The address field contains the starting execution address and is interpreted as 3-byte address. There is no data field. This is the last record in the file. (EASy68K always uses this termination record when creating S-Record files.)

S9 Record. Termination record. The address field contains the starting execution address and is interpreted as 2-byte address. There is no data field. This is the last record in the file.

EXAMPLE

The following program was assembled with EASy68K. The .L68 file is shown. The .L68 file contains the address, machine code, line number and source code.

00001000 Starting Address
 Assembler used: EASy68K Editor v1.9.2
 Created On: 1/28/2004 4:51:58 PM

Address	Machine Code	Line	Source Code
00000000		1	*-----

00000000		2	* Program Number:
S-Record demo for EASy68k			
00000000		3	* Written by :
Chuck Kelly			
00000000		4	* Date Created :
Jan-28-2004			
00000000		5	* Description :
demonstrate the S-Record file format used by EASy68K.			
00000000		6	*
00000000		7	* This program is
public domain.			
00000000		8	*-----

00000000		9	
00001000		10	START ORG
\$1000 the program will load into address \$1000			
00001000 123C 0003		11	move.b
#3,d1 put 3 in low byte of data register D1			
00001004 5A01		12	add.b
#5,d1 add 5 to low byte of data register D1			
00001006		13	* Display string
textD1			
00001006 303C 000E		14	move
#14,d0 load task number into D0			
0000100A 43F9 00044446		15	lea
textD1,a1 load address of string to display into A1			

```

00001010 4E4F          16          trap
#15      trap #15 activates input/output task
00001012          17  * Display D1 as a
number
00001012 303C 0003      18          move
#3,d0      task number 3 into D0
00001016          19  * task number 3 is
used to display the contents of D1.L as a number
00001016 4E4F          20          trap
#15      display number in D1.L
00001018 4EB9 00022222  21          jsr
newLine
0000101E          22  * Stop execution
0000101E 4E72 2000      23          STOP
#$2000
00001022          24
00022222          25          org
$22222  this subroutine is located at address $22222
00022222          26  * Subroutine to
display Carriage Return and Line Feed
00022222          27  newLine:
00022222 48E7 8040      28          movem.l
d0/a1,-(a7) push d0 & a1
00022226 303C 000E      29          move
#14,d0      task number into D0
0002222A 43F9 00044462  30          lea
crlf,a1 address of string
00022230 4E4F          31          trap
#15      display return, linefeed
00022232 4CDF 0201      32          movem.l
(a7)+,d0/a1 restore d0 & a1
00022236 4E75          33          rts
return
00022238          34
00044444          35          org
$44444  this data is located at address $44444
00044444          36  sum2      ds.w
1      reserve word of memory for sum2
00044446 44 31 20 63 6F 6E ... 37  textD1  dc.b 'D1
contains: ',0      null terminated string
00044454 44 32 20 63 6F 6E ... 38  textD2  dc.b 'D2
contains: ',0      null terminated string
00044462 0D 0A 00      39  crlf      dc.b
$d,$a,0      carriage return & line feed, null
00044465          40

```

```

00044465                                41
*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
/\/\/\/\/\/\/\/\/\/\/\
00044465                                42  * E R R O R E R R O
R E R R O R E R R O R E R R O R
00044465                                43
*/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\/\
/\/\/\/\/\/\/\/\/\/\/\
00044465                                44  * EASy68K does not
support addresses beyond $FFFFFF or 16MBytes
00044465                                45  * Comment out the
following org directive if you want to run this
program.
12345678                                46                org
$12345678
12345678 69 6E 76 61 6C 69 ... 47  junk      dc.b
'invalid address'
12345687                                48
12345687                                49                END
START      end of program with start address specified

```

The S-record file created from the above program follows:.

```

S021000036384B50524F4720202032304352
4541544544204259204541535936384B6D
S1251000123C00035A01303C000E43F90004
44464E4F303C00034E4F4EB9000222224E72
200004
S21A02222248E78040303C000E43F9000444
624E4F4CDF02014E75C2
S223044446443120636F6E7461696E733A20
00443220636F6E7461696E733A200000D0A00

```

9A
S31412345678696E76616C69642061646472
657373EA
S804001000EB

The S0 Record contains the following information:

version number / -
module name / / -
revision number
checksum
S0 0000 6 8 K P R O G 2 0 C R
E A T E D B Y E A S Y 6 8 K /
S021000036384B50524F4720202032304352
4541544544204259204541535936384B6D

The version and revision number refer to the
EASy68K S-Record file type, not the product
version.

The file consists of one S0 record, one S1 record, two
S2 records, one S3 record and an S8 record.

The first S1 record is comprised as follows:

S1 25 1000

123C00035A01303C0000E43F9000444464E4F
303C00034E4F4EB90000222224E722000 04

- S1 S-record type, indicating it is a data record to be loaded at a 2-byte address.
- 25 Hexadecimal (decimal 37), indicating that thirty-seven character pairs, representing a 2 byte address, 34 bytes of binary data, and a 1 byte checksum, follow.
- 1000 Four character 2-byte address field; hexadecimal address \$1000, where the data which follows is to be loaded.
- 123C00035A01 . . . 34 character pairs representing the actual binary data.
- 04 The checksum.

The first S2 record is comprised as follows:

S2 1A 022222

48E78040303C0000E43F9000444624E4F4CDF
02014E75 C2

- S2 S-record type, indicating it is a data record to be loaded at a 3-byte address.
- 1A Hexadecimal (decimal 26), indicating that twenty-six character pairs, representing a 3 byte address, 22 bytes of binary data, and a 1 byte checksum, follow.

- 022222 Six character 3-byte address field; hexadecimal address \$022222, where the data which follows is to be loaded.
- 48E78040303C . . . 22 character pairs representing the actual binary data.
- C2 The checksum.

The second S2 record contains additional data which is to be loaded at address \$044446

The first S3 record is comprised as follows:

S3 14 12345678
696E76616C69642061646472657373 EA

- S3 S-record type, indicating it is a data record to be loaded at a 4-byte address.
- 14 Hexadecimal (decimal 20), indicating that twenty character pairs, representing a 4 byte address, 15 bytes of binary data, and a 1 byte checksum, follow.
- 12345678 Eight character 4-byte address field; hexadecimal address \$12345678, where the data which follows is to be loaded.
- 696E76616C69 . . . 15 character pairs representing the actual binary data.
- EA The checksum.

The S8 record is comprised as follows:

S8 04 001000 EB

- S8 S-record type, indicating it is a termination record with a 3-byte address.
- 04 Hexadecimal (decimal 4), indicating that four character pairs follow.
- 001000 The address field, hexadecimal address \$001000 indicating the starting execution address.
- EB The checksum.

EASy68K always uses the S8 record.

D	1101		CR	GS	-	=	M]	m
}									
E	1110		SO	RS	.	>	N	^	n
~									
F	1111		SI	US	/	?	O	_	o
DEL									
^	^								
(Hex)									

MSB = Most Significant Bits

LSB = Least Significant Bits

For example: The ASCII code for 'A' is 1000001 binary or 41 hexadecimal.

ASCII Symbol Names

NUL = NULL character

Escape

SOH = Start Of Heading

1

STX = Start Of Text

2

ETX = End Of Text

3

EOT = End Of Transmission

4

ENQ = Enquiry

Acknowledge

ACK = Acknowledge

Idle

BEL = Bell

Transmission Block

BS = Backspace

HT = Horizontal Tab

DLE = Data Link

DC1 = Device Control

DC2 = Device Control

DC3 = Device Control

DC4 = Device Control

NAK = Negative

SYN = Synchronous

ETB = End of

CAN = Cancel

EM = End of Medium

LF = Line Feed
VT = Vertical Tab
FF = Form Feed
CR = Carriage Return
Separator
SO = Shift Out
Separator
SI = Shift In

SP = Space
DEL = Delete

SUB = Substitute
ESC = Escape
FS = File Separator
GS = Group

RS = Record

US = Unit Separator

68000 Quick Reference Guide

The Quick Reference Guide is a (.pdf) file format for printing purposes. A (.pdf) file viewer is required to view this file.

[EASy68k Quick Reference Guide](#)

GNU General Public License

Version 2, June 1991

Copyright (C) 1989, 1991 [Free Software Foundation](#),
Inc. 59 Temple Place - Suite 330, Boston, MA 02111-
1307, USA

Everyone is permitted to copy and distribute verbatim
copies of this license document, but changing it is not
allowed.

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of [Section 1](#) above, provided that you also meet all of these conditions:
 1. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.
 2. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

3. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you;

rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under [Section 2](#)) in object code or executable form under the terms of Sections [1](#) and [2](#) above provided that you also do one of the following:
 1. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections [1](#) and [2](#) above on a medium customarily used for software interchange; or,
 2. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections [1](#) and [2](#) above on a medium customarily used for software interchange; or,

3. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with [Subsection b](#) above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software

distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version",

you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.
11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS

TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

EASy68K Credits

The authors of this software assumes no liability for its functionality and are not responsible for damages, losses or liabilities as a result of using this software. Use at your own risk. This application is "Open Source" and may be distributed under the [GNU General Public Use License](#).

Editor/Assembler GUI: Tim Larson

Original assembler & simulator code: Paul McKee

68000 Reference Guide: Jimmy Mårdell

Advanced simulator breakpoints: Eric Nelson

HTML Help: Aaron Curley, Curt Vickre & Jon Squires

Simulator GUI, modifications, new features and project lead: Charles Kelly

web: www.easy68k.com

Post reports or undocumented features, suggestions and comments on the EASy68K Forum.

Registers

The 68000 has 16 registers, 8 data registers (D0-D7) and 8 address registers (A0-A7) and they are all longword, i.e. 32 bit. No data or address register is different from any other. That means whenever you can use D0, you can always use D1-D7 also, the D0 register isn't a "main register", as the A register is on a Z80, or the registers on a x86 where all registers are made for different purposes (AX - accumulator, CX - counter, SI, DI - pointers etc). The only real difference between the registers is data registers vs. the address registers.

One important thing is that if you have a longword stored in a data register and you move a byte into it, the most significant three bytes are still there! This may cause problems when you later add or multiply the register, you MUST clear the most significant part!

The 68k also has a PC, Program Counter, as do most CPUs. Works the same way too (it is a 24 bit register). The stack pointer is actually the A7 register, so maybe you should say that A7 works differently compared to A0-A6, but it doesn't really. You could use A6 as stack pointer, except that all ROM routines and the CPU itself (see bsr and jsr) use A7 as the SP so it wouldn't be a good idea.

Flags

Then we have the flags. They're stored in the status register (SR), a 16 bit register. It's divided into two parts, the system byte (bit 15-8) and the user byte (or flag register) (bit 7-0). Here's a description over the whole SR register:

Bits 15 to 0 from most to least significant.

```
| T | - | S | - | - | I2,1,0 | - | - | - | X | N | Z | V | C | - the  
status register
```

* The System byte

Bit 15: T - The trace bit. If it's set an interrupt will be called after each instruction. Often used in debuggers.

Bit 13: S - Supervisor bit. When this bit is set, you have more "access" to some instructions and also to the system byte. The reason for this is that it prevents programs to disturb the OS with some instructions that you shouldn't use if you're not writing an OS. Is enabled when interrupts are generated.

Bit 8-10: The interrupt mask

The I0, I1 and I2 bits of the system register are used to set the interrupt mask: in fact, it means that they are set to an interrupt level: if the trap generated has a level higher than the interrupt mask, then the trap is executed. Otherwise it is ignored. (ignoring a trap generally means that another interrupt, with a higher priority is being treated)

Here is how these bits are set:

	I2	I1	I0	
level 0	0	0	0	-----> lowest priority
level 1	0	0	1	
level 2	0	1	0	
level 3	0	1	1	
level 4	1	0	0	
level 5	1	0	1	
level 6	1	1	0	
level 7	1	1	1	-----> highest priority

Note: #0=%000; #1=%001; #2=%010; etc.

One should refer to the \system.txt file and \lesson\lesson_3.txt for more on interrupts.

* The flag register

- C-flag (Carry). Works as carry is used to work. If you add two 8 bit numbers, the C-flag will be the 9th bit. Also used with shift and rotation.

- V-flag (oVerflow). Will be set if a result can't be represented. For example, when you add \$7F and \$01, \$80 can't be stored since that means -128 in two complement.

- Z-flag (Zero). Set if the result of an operation is zero.

- N-flag (Negative). If the highest bit in the result is set (in two complement it means the sign bit), N will be set.

- X-flag (eXtended). This flag is a copy of the carry-flag, but it won't be changed in all operations where C is changed. This allows you to first make a check (that will set C and X), then some other instructions that will change the C flag but not the X flag, and THEN

you can make the branch according to the flags, which means you can use the X flag.

How is Data Stored in Memory?

The 68000-processor supports two different data types: binary and BCD. Float numbers are not supported. One needs a coprocessor. I don't know whether one exists for the 68000 processor. Data is stored in high endian format: The Z80 and x86 stores binary numbers with the LSB in the first byte, i.e. \$12345678 is stored \$78,\$56,\$34,\$12 in the memory. The 68k stores it the other way around, \$12,\$34,\$56,\$78. That way to store numbers in the memory is called high endian.

Some Information about Instructions

Most instructions on the 68k have a suffix that shows how much data should be used in the instructions. This can be a byte (.b), a word (.w) or a longword (.l). Another very important thing is that all instructions are backwards (compared to most CPUs). With that I mean that the source of the instruction is the first argument and the destination is the last argument. For example: `MOVE.W D0,D1` moves (actually, copies) the first 16 bits (a word) from D0 to D1.

Effective Addressing

The 68000 has 14 different ways to address things in the memory. This example will use the instruction MOVE (LD on Z80, MOV on x86) to demonstrate how they work. Later, in the instruction summary, you'll see that when an instruction has <ea> (effective address) as an argument, it doesn't mean it can use all 14 ways to address. It almost always has restrictions.

1. DATA REGISTER DIRECT

Syntax: Dn (where n is 0-7)

Example: MOVE.L D1,D0 copies the contents of D1 to D0. When the instruction is executed, both registers will contain the same information. When moving a byte or a word, the upper part of the register will remain unchanged.

Instruction	Before	After
MOVE.B D1,D0	D0=FFFFFFFF	D0=FFFFFFF67
	D1=01234567	D1=01234567
MOVE.W D1,D0	D0=FFFFFFFF	D0=FFFF4567
	D1=01234567	D1=01234567

- **ADDRESS REGISTER DIRECT**

Syntax: An (n is 0-7)

Example: MOVE.L A1,D0 copies all of A1 to D0. After the instruction, both registers contain the same information. When transferring address registers, you must use word or longword. When a word is transferred TO an address register, bit 15 (the sign bit) will be copied through the whole upper word (bit 16-31). If it wasn't so, a negative number would become positive.

Instruction	Before	After
MOVE.W A1,D0	D0=FFFFFFFF	D0=FFFF4567
	A1=01234567	A1=01234567
MOVE.W D0,A1	D0=01234567	D0=01234567
	A1=FFFFFFFF	A1=00004567
MOVE.W D0,A1	D0=0000FFFF	D0=0000FFFF
	A1=00000000	A1=FFFFFFFF

- **ADDRESS REGISTER INDIRECT**

Syntax: (An) (n is 0-7)

Example: MOVE.L (A0),D0 copies the longword stored at address location A0 (you say A0 points to the longword). If you refer to a word or a longword, the address in the address register must be an even number. THIS CAN CAUSE BIG PROBLEMS AND UNEXPECTED ERRORS!

Instruction	Before	After
MOVE.L (A1),D0	D0=FFFFFFFF	D0=01234567

	A1=00001000	A1=00001000
	\$1000=01234567	\$1000=01234567

- **ADDRESS REGISTER INDIRECT WITH POST-INCREMENT**

Syntax: (An)+ (n is 0-7)

Description.: Works the same as the previous addressing mode, except that after the instruction, register An will be increased with the size of the operation. A special case is when you use A7 and transfer a byte, because A7 will be increased with 2 instead of 1, because A7, as the stack pointer, must be an even number.

Example: MOVE.L (A1)+,D0 copies the longword which A1 points to to D0, and increases A1 by 4 bytes.

Instruction	Before	After
MOVE.L (A1)+,D0	D0=FFFFFFFF	D0=01234567
	A1=00001000	A1=00001004
	\$1000=01234567	\$1000=01234567

- **ADDRESS REGISTER INDIRECT WITH PRE-DECREMENT**

Syntax: -(An) (n is 0-7)

Description.: Works the same as the previous addressing mode, except that register An will first be decreased with the operand size (with the exception of A7), then the data will be transferred.

Example: `MOVE.L -(A1),D0` first decreases A1 by 4 bytes, then copies the longword stored at A1 to D0.

Instruction	Before	After
MOVE.L -(A1),D0	D0=FFFFFFFF	D0=01234567
	A1=00001004	A1=00001000
	\$1000=01234567	\$1000=01234567

- **ADDRESS REGISTER INDIRECT WITH DISPLACEMENT**

Syntax: `x(An)` (x is 16 bit, n is 0-7)

Description: The location pointed at `x+An` is the one that will be copied.

Example: `MOVE.L 4(A1),D0` copies the longword stored at `A1+4` to D0. A1 will, after the instruction, remain unchanged. Note that if x is bigger than `$7FFF`, it means a negative value. It's because of the sign extension.

Instruction	Before	After
MOVE.L 4(A1),D0	D0=FFFFFFFF	D0=01234567
	A1=00001000	A1=00001000
	\$1004=01234567	\$1004=01234567

- **ADDRESS REGISTER INDIRECT WITH INDEX**

Syntax: `x(An,Dn.L)` (x is an 8 bit signed constant, n is 0-7)
`x(An,Dn.W)`

x(An,An.W)
x(An,An.L)

Description.: Works the same as the previous addressing mode, except that another register also will be added (if it's a word, a sign extension will be made before, so it will be a subtraction). When working with words or longwords, the generated address must be an even address.

Example: `MOVE.L 4(A1,A2.L),D0` copies the longword stored at $A1+A2+4$ to D0. A1 and A2 will after the instruction remain unchanged. Note that if x is bigger than \$7F, it means a negative value. It's because of the sign extension.

Instruction	Before	After
MOVE.L 4(A1,A2.L),D0	D0=FFFFFFFF	D0=01234567
	A1=00001000	A1=00001000
	A2=00001000	A2=00001000
	\$2004=01234567	\$2004=01234567

- **ABSOLUTE SHORT ADDRESS**

Syntax: x (x is a 16 bit signed constant)

Description: The address will be sign extended before it's used, but the MSB is ignored (don't bother about that). The sign extension means that near addressing can only be used on the first 32Kb.

Example: `MOVE.L $1000,D0` copies the longword stored at \$1000 to D0. Note that there is no parentheses! If you mean an immediate value, you put a # before the value (see below). However, adding the parentheses is not a bad idea: the

assembler will accept it and it will add in readability to the source.

Instruction	Before	After
MOVE.L \$1000,D0	D0=FFFFFFFF	D0=01234567
	\$1000=01234567	\$1000=01234567

Forcing Absolute Short Addressing: In EASy68K if an instruction such as: MOVE.L \$8000,D0 is assembled the assembler will automatically use absolute long addressing and encode the address as \$00008000. It is possible to force the assembler to use absolute short addressing by using .W as: MOVE.L \$8000.W,D0. Forcing short addressing will always result in a Warning message "Forcing SHORT addressing disables range checking of extension word".

Instruction	Before	After
MOVE.L \$8000,D0	D0=FFFFFFFF	D0=01234567
	\$00008000=01234567	\$00008000=01234567
MOVE.L \$8000.W,D0	D0=FFFFFFFF	D0=87654321
	\$00FF8000=87654321	\$00FF8000=87654321

- **ABSOLUTE LONG ADDRESS**

Syntax: x (x is a 32 bit constant)

Description: Works EXACTLY as the last one, except that x is a 32 bit value (the instruction is two bytes longer also).

Example: MOVE.L \$10000,D0 copies the longword stored at \$10000 to D0.

Instruction	Before	After
-------------	--------	-------

MOVE.L	D0=FFFFFFFF	D0=01234567
\$10000,D0	\$10000=01234567	\$10000=01234567

Forcing Absolute Long Addressing: In EASy68K if an instruction such as: MOVE.L \$1000,D0 is assembled the assembler will automatically use absolute short addressing and encode the address as \$1000. It is possible to force the assembler to use absolute long addressing by using .L as: MOVE.L \$1000.L,D0.

Instruction	Code
MOVE.L \$1000,D0	2038 1000
MOVE.L \$1000.L,D0	2039 00001000

- **PROGRAM COUNTER WITH DISPLACEMENT**

Syntax: x(PC) or (x,PC) (x is a 16 bit signed constant)

Description: The displacement word (x) is specified as an address relative to the current PC. The assembler calculates the relative offset to the displacement address and adds the resulting number to the program counter to determine the address. The displacement is a signed number (meaning that the limits are -32768 to +32767). This addressing mode may be used to write programs that are position independent. Position independent programs and their local data may be moved to any area of memory and will run correctly. This is possible if all references to local data are made with PC relative addressing.

Example: MOVE.L \$1102(PC),D0 copies the longword stored at PC+\$102 to D0 assuming the current PC is \$1000.

Instruction	Before	After

MOVE.L	D0=FFFFFFFF	D0=01234567
\$1102(PC),D0		
assuming PC=\$1000	\$1102=01234567	\$1102=01234567

- **PROGRAM COUNTER WITH INDEX**

Syntax: x(PC,Dn.L) or (x,PC,Dn.L) (x is an 8 bit signed constant, n is 0-7)

x(PC,Dn.W) or (x,PC,Dn.W)

x(PC,An.W) or (x,PC,An.W)

x(PC,An.L) or (x,PC,An.L)

Description: The displacement byte (x) is specified as an address relative to the current PC. The assembler calculates the relative offset to the displacement address and adds the resulting number the program counter and the sign extended index register to determine the address. The displacement is a signed number (meaning that the limits are -128 to +127). This addressing mode may be used to write programs that are position independent. Position independent programs and their local data may be moved to any area of memory and will run correctly. This is possible if all references to local data are made with PC relative addressing.

Example: MOVE.L \$1010(PC,A1.L),D0 copies the longword stored at PC+A1+\$10 to D0 assuming the current PC is \$1000.

Instruction	Before	After
MOVE.L	D0=FFFFFFFF	D0=01234567
\$1010(PC,A1.L),D0	\$2010=01234567	\$2010=01234567
assuming PC=\$1000	A1=00001000	A1=00001000

- **IMMEDIATE DATA**

Syntax: #x (x is 8, 16 or 32 bits)

Description: Uses the immediate value x.

Example: `MOVE.L #$10002000,D0` copies \$10002000 to D0.
Note that if you copy a word to an address register, the word will be sign extended.

Instruction	Before	After
<code>MOVE.L #\$10002000,D0</code>	<code>D0=01234567</code>	<code>D0=10002000</code>

- **ADDRESSING WITH THE STATUS REGISTER**

Syntax: SR
CCR

Description: The only instructions that are allowed to use this address mode are: `ANDI` (AND immediate), `EORI` (exclusive OR immediate), and `ORI` (OR immediate). If the length is a byte, the flag register is changed. If it's a word, both the flag register and the system byte are changed (but only if the supervisor bit is set) The assembler recognize both SR (that means both flags and system-byte) and CCR (only flag register), so you don't have to specify the length.

Example: The instruction `ORI #5,CCR` sets both the carry flag

(C) and the zero flag (Z). The other flags remains unchanged.
Note: #5=#%00000101

Instruction	Before	After
ORI #5,CCR	CCR=0000	CCR=0005

The Stack

As mentioned earlier, A7 functions as the stack pointer. The stack (user and system stack) is stored in the "system memory": you should not worry about the size: it should always be more than enough. You may have noticed, if you jumped to the instruction summary at first: there are no push/pop instructions! That's because they're not necessary, MOVE will do both jobs for us.

For example, if we want to push D0.W (with this I mean D0, the lower 16 bits) we just MOVE.W D0,-(A7). And to pop it back, just MOVE.W (A7)+,D0. Remember that when popping a word, the upper 16 bits in D0 are still there.

When you want to push more than one register at the same time, for example in the beginning of a subroutine, it may be a bit time consuming to type all the move instructions. There is a shortcut: MOVEM. With this instruction you can push (or just copy) many registers at once. If you want to push D0-D4 and A0-A2 for example, you just movem d0-d4/a0-a2,-(a7) and then movem (a7)+,d0-d4/a0-a2.

Another way to put things on the stack is with the PEA instruction. It pushes an effective address on the stack, used when pushing pointers. This will decrease A7 with 4 (the size of a pointer).

How to Address Variables

How do we address things? How do I copy the word stored at var1 to D0? It's very simple, just `move.w (var1),d0`. As usual, the upper 16 bits are unaffected. How do we make A0 point at str1? Also easy. Just use `lea str1(PC),a0`. You MUST type (PC) after the label, otherwise it will not work! This is just the way we get the address of a label (see [Effective Addressing](#)) After loading A0 with the pointer to str1, you can get the first byte with `move.b (a0)+,d0` This copies the first byte ('H' = 72) to D0 and increases A0 with one, thus pointing to the next character. Here you must use parentheses since else it would mean copy A0, not the byte stored at A0, to D0.

Common Mistakes

"If anything can go wrong, it will. If it can't, it will anyway."

You get some strange errors messages when assembling or running your program and you don't know what's wrong? Take it easy, it is quite common :->

But finding the errors in the source can be hard, VERY hard, sometimes. There are some common mistakes though if you've programmed other assembly languages before. Those include:

- * You MUST put a # before an immediate value!! There are exceptions, for example LEA 10(A7),A7, but when using MOVE, CMP and other instructions it can be a fatal error. For example, MOVE.B \$10,D0 means move the byte stored at address \$10 to D0! This can cause protected memory error or something like that. Always check so you've not missed any pound signs!
- * The first operand is the source, the second the destination! This is also a very common error in the beginning. We will not speak of what it could do, but instead of getting things from important places in the RAM (like, the interrupt vector), you store some trash there! Not good! Check all instructions with two operands! Most instructions have two operands.
- * Since the upper part of the register will remain there until you move another longword to the register, it may cause big problems! For example, when getting a byte from somewhere in the memory, and then you want to multiply it, you MUST be sure that the upper part of the byte is cleared out, since you don't specify any operation size (.B, .W or .L) when multiplying. To do this, use either CLR or EXT (see 2.0 Instruction summary).
- * All tables and other variables you will address with (An) must be at an even address. If you have defined it one byte before the table, it

will cause the table to start at an odd address, which will probably cause in a calculator crash. To avoid this, move all such tables/arrays at the beginning of the variable section (I think, or at least it seems so, all instructions have the size 2, 4, 6...). You may also use the "even" assembly directive which aligns the table on a longword boundary.

* Learn to use the debugging tools in the EASy68K simulator. Place breakpoints in your program and Trace individual instructions as you observe the contents of the 68000 registers. Make sure you fully understand the operation of each instruction and verify it's behavior as your trace through the code.

Be aware that when programming an assembly language, most of the time (>50%) you're sitting and correcting bugs.

Data Movement

These instructions move data from one place to another.

<u>EXG</u>	The contents of two registers will be exchanged.
<u>LEA</u>	Calculates a memory location and stores it into an address register.
<u>LINK</u>	Allocates a stack frame.
<u>MOVE</u>	Copies the contents in one register/memory location into another register/memory location.
<u>MOVEA</u>	Same as MOVE except that the destination is an address-register.
<u>MOVEM</u>	transfers many registers to or from the memory.
<u>MOVEP</u>	transfers data to or from an 8 bit peripheral unit.
<u>MOVEQ</u>	puts a constant in a data register.
<u>PEA</u>	calculates a memory address and stores it on the stack.
<u>SWAP</u>	Swaps the low and high words in a data register.
<u>UNLK</u>	removes a stack frame

Integer Arithmetic

These instructions perform simple two complement operations on binary data.

[ADD](#), [ADDA](#), [ADDI](#), [ADDQ](#),
[ADDX](#)

Different kinds of addition.

[CLR](#)

Clears an operand.

[CMP](#), [CMPA](#), [CMPI](#), [CMPM](#)

Compares two operands

[DIVS](#), [DIVU](#)

Integer division, signed and unsigned.

[EXT](#)

Makes a sign extension, byte to word or word to longword

[MULS](#), [MULU](#)

Multiplication, signed and unsigned.

[NEG](#), [NEGX](#)

Invokes the twos complement on a number.

[SUB](#), [SUBA](#), [SUBI](#), [SUBQ](#),
[SUBX](#)

Different kinds of subtraction.

[TAS](#)

used to synchronize more than one processor

[TST](#)

Compares an operand with 0.

Logical Operations

These operations perform logical operations on binary numbers. A logical operation is either "true" (1) or "false" (0).

[AND](#), [ANDI](#) Logical AND on two binary integers

[OR](#), [ORI](#) Logical OR

[EOR](#), [EORI](#) Exclusive OR (XOR)

[NOT](#) Returns the operands ones complement (0 -> 1, 1 -> 0)

Shift and Rotation

These instructions perform arithmetic and logical shift and rotation with or without extra carry.

[ASL](#), [ASR](#) Arithmetic shift left/right.
[LSL](#), [LSR](#) Logical shift left/right.
[ROL](#), [ROR](#) Rotation left/right without extra carry.
[ROXL](#), [ROXR](#) Rotation left/right through extra carry.

Bit Manipulation

These instructions affect single bits in a byte. All instructions test the bit before affecting it.

[BTST](#) Tests a bit

[BSET](#) Tests a bit, then sets it (1)

[BCLR](#) Tests a bit, then resets it (0)

[BCHG](#) Tests a bit, then inverts it (0 -> 1, 1 -> 0)

Bit field instructions allow manipulation of bits across a 32 bit field. Bit field instructions were introduced in the 68020 microprocessor and are not supported by the 68000 microprocessor. EASy68K is normally a 68000 only environment but does support a modified form of bit field instructions for educational purposes. In EASy68K the bit field instructions are restricted to using only 68000 addressing modes. An assembler [directive](#) or menu [option](#) in EASy68K enables or disables the assembly of bit field instructions.

[BFCHG](#) Test a bit field, then complement it

[BFCLR](#) Test a bit field, then clear it to 0

[BFEXTS](#) Test a bit field, then sign extend it to 32 bits and load it into the specified data register

[BFEXTU](#) Test a bit field, then zero extend it to 32 bits and load it into the specified data register

[BFFFO](#) Test a bit field, then put the bit number of the first bit that contains 1 in the specified data register

[BFINS](#) Insert the bit field from the specified data register into the destination, then set the condition codes

[BFSET](#) Test a bit field, then set it to 1

[BFTST](#) Test a bit field

BCD Instructions

These operations work with Binary Coded Decimal numbers.

[ABCD](#) BCD addition

[SBCD](#) BCD subtraction

[NBCD](#) BCD negate

Program Control

These instructions are used in branches, jumps, and calls.

<u>Bcc</u>	A group of 15 instruction that branches depending on the flags.
<u>DBcc</u>	15 instructions that perform loops.
<u>Scc</u>	16 instructions that will set/reset a byte depending on the flags.
<u>BSR</u> , <u>JSR</u>	Subroutine calls.
<u>RTS</u>	Return from a subroutine.
<u>JMP</u>	Absolute jumps.
<u>RTR</u>	Pops the PC and flags from the stack.

System Control

These instructions change the state of the 68000 processor. They require that the supervisor bit is set.

[MOVE](#) Gives a program in supervisor mode access to the stack
[USP](#) pointer in user mode.

[RESET](#) Restores the peripheral units.

[RTE](#) Returns from an interrupt.

[STOP](#) Stops the execution until a given interrupt occurs.

[CHK,](#)
[TRAPV](#) Finds fatal program errors.

[TRAP](#) 16 instructions that give a program in user mode the possibility to call another program in supervisor mode.

Other Instructions

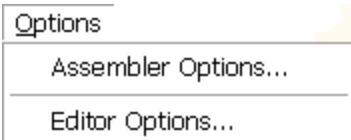
ILLEGAL Causes an interrupt (Illegal instruction)

NOP Does nothing for one processor cycle (not very long wait)

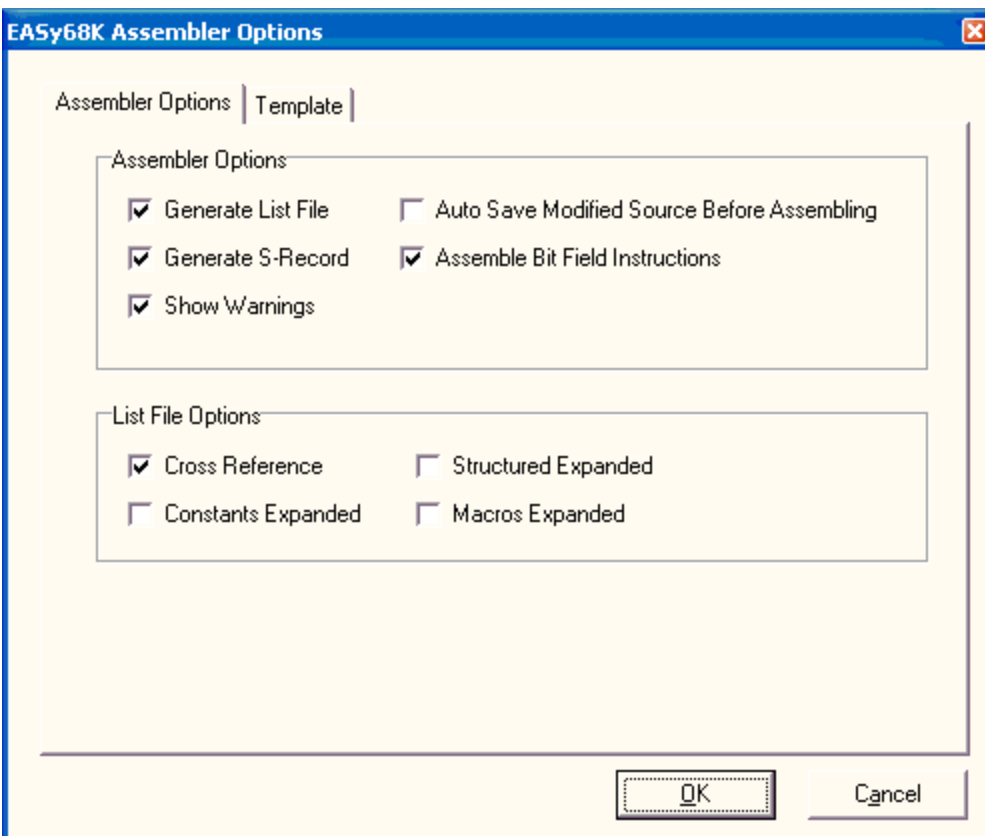
-

Edit68K - Options

Assembler and editor options may be changed by selecting the desired item under the Options menu item.



Assembler Options...



Assembler Options

- **Generate List File**

The assembler will create a list file when checked. The list file is used by the EASy68K simulator to provide source level debugging. The list file will have the same name as the source file with an .L68 file extension. The list file contains the original program source along with the machine code generated by the assembler and error messages. The list file may also include instructions added by the assembler for structured control statements or macros. The List File Options described below along with some assembler directives control how much information is included in the list file.

- **Generate S-Record**

The assembler will create an S-Record file when checked. The S-Record file is required by the EASy68K simulator. The S-Record file will have the same name as the source file with an .S68 file extension. The S-Record file contains the machine code generated by the assembler in a text format compatible with the Motorola S-Record format. See the included description of the [EASy68K S-Record format](#) for details. S-Record files may be converted to binary files using the [EASyBIN](#) utility.

- **Show Warnings**

The assembler will display warning messages during assembly when checked. Error messages are always displayed.

- **Auto Save Modified Source Before Assembling**

When checked, the assembler will save all modified source files before assembling.

- **Assemble Bit Field Instructions**

When checked, assembly of bit field instructions is supported. Only 68000 addressing modes are supported. [Bit field](#)

instructions are available in 68020 and higher 68K processors and were not part of the 68000 instruction set. Support has been added to EASy68K for educational purposes. EASy68K does not include support for any other non 68000 instructions.

Template

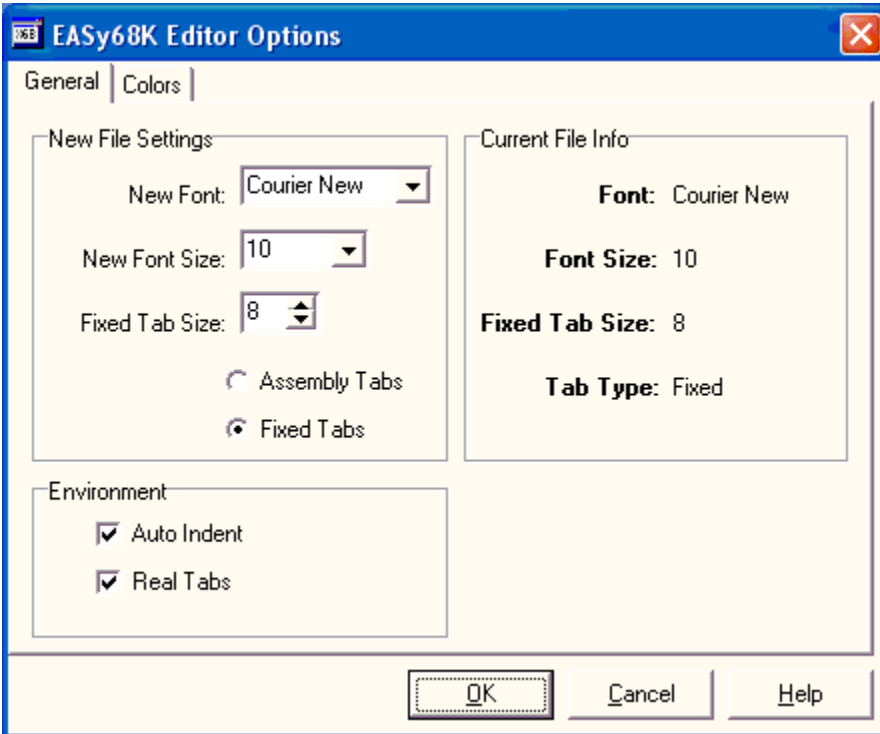
- The template is used to create a new source file.

List File Options

- Cross Reference
When checked the list file will include a cross reference at the bottom of the file of all labels used along with their respective values.
 - Constants Expanded
When checked the list file will contain all of the data created by the assembler from constants defined in the source file.
 - Structured Expanded
When checked the list file will contain all of the code added by the assembler to implement the structured assembly statements contained in the source file.
 - Macros Expanded
When checked the list file will contain a complete expansion of each macro call.
-

Editor Options...

General



This window may be used to change the appearance of text in the currently selected window. The Editor Options are also used as the default settings for new source files. The font and tab characteristics apply to the entire window.

New File Settings

- New Font
Used to select a new font.
- New Font Size
Used to select a new font size.

- **Fixed Tab Size**
The size, in characters, of each tab.
- **Assembly Tabs or Fixed Tabs**
Select the desired tab type. Assembly Tabs position the tab stops at predefined positions that work best for traditional formatting of assembly source. Fixed Tabs uses the tab size specified in the Fixed Tab Size field. Fixed tabs are best for writing Structured Assembly code.

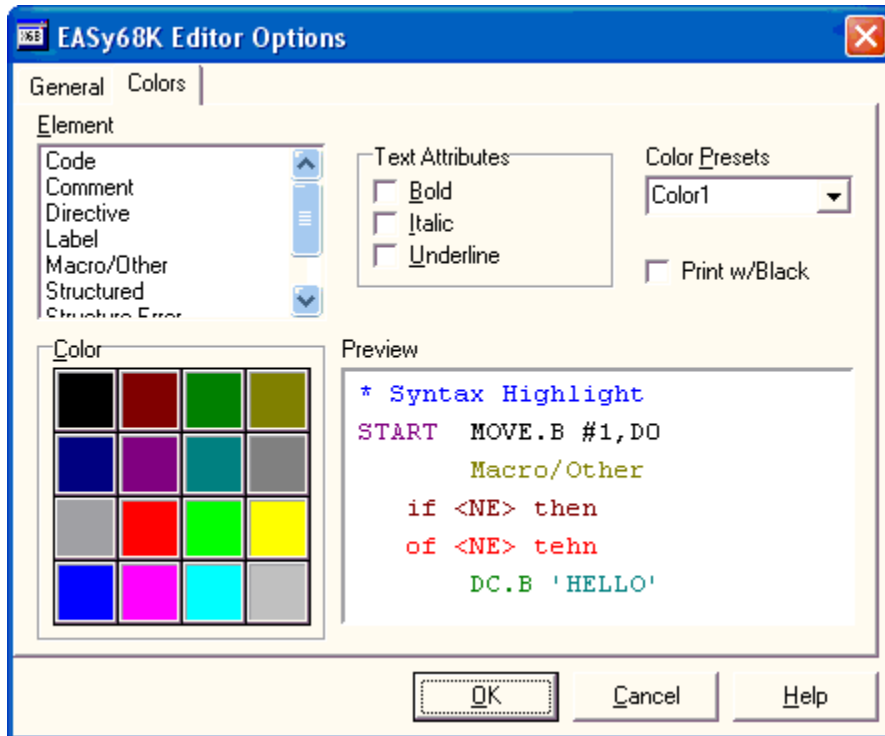
Current File Info

- **Font**
The name of the current font.
- **Font Size**
The size of the current font.
- **Fixed Tab Size**
The size of the fixed tab in characters.
- **Tab Type**
The type of tabs currently in use.

Environment

- **Auto Indent**
When checked the editor will indent new lines to the same level as the preceding line.
- **Real Tabs**
When checked the editor uses real tab characters. When unchecked the editor uses spaces in place of tab characters.

Colors



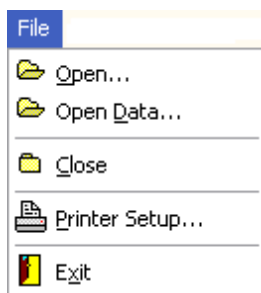
This window may be used to change the editors colors and text attributes used for syntax highlighting. The Preview window displays the style changes. Clicking OK applies any changes to every open file in the editor.

- Element
Code elements may be individually selected so their color and/or attributes may be changed.
- Text Attributes
The attributes of the currently selected Element are displayed and may be changed.

- Color Presets
Selecting a color preset will change the colors and attributes of all elements to match the preset. Syntax highlighting may be disabled by selecting Disabled.
- Print w/Black
Printing will not use colors when checked.
- Color
The color of the currently selected Element may be selected by clicking on the desired color.
- Preview
The colors and styles that will be applied to the editor when the OK button is clicked. Click Cancel to exit without making any changes.

Sim68K - Basic Operation

A 68000 program can be executed by starting the simulator manually and loading a 68000 S-Record (.S68) file or by clicking the "Execute" button after assembling a 68000 Source file in Edit68K. A batch file may also be used to automatically load and run a 68000 .S68 file. See [Running a Program From a Batch File](#) below.



Open - Loads a 68000 program from S-Record (.S68) file into the simulator's 68000 memory space and initializes the simulator environment.

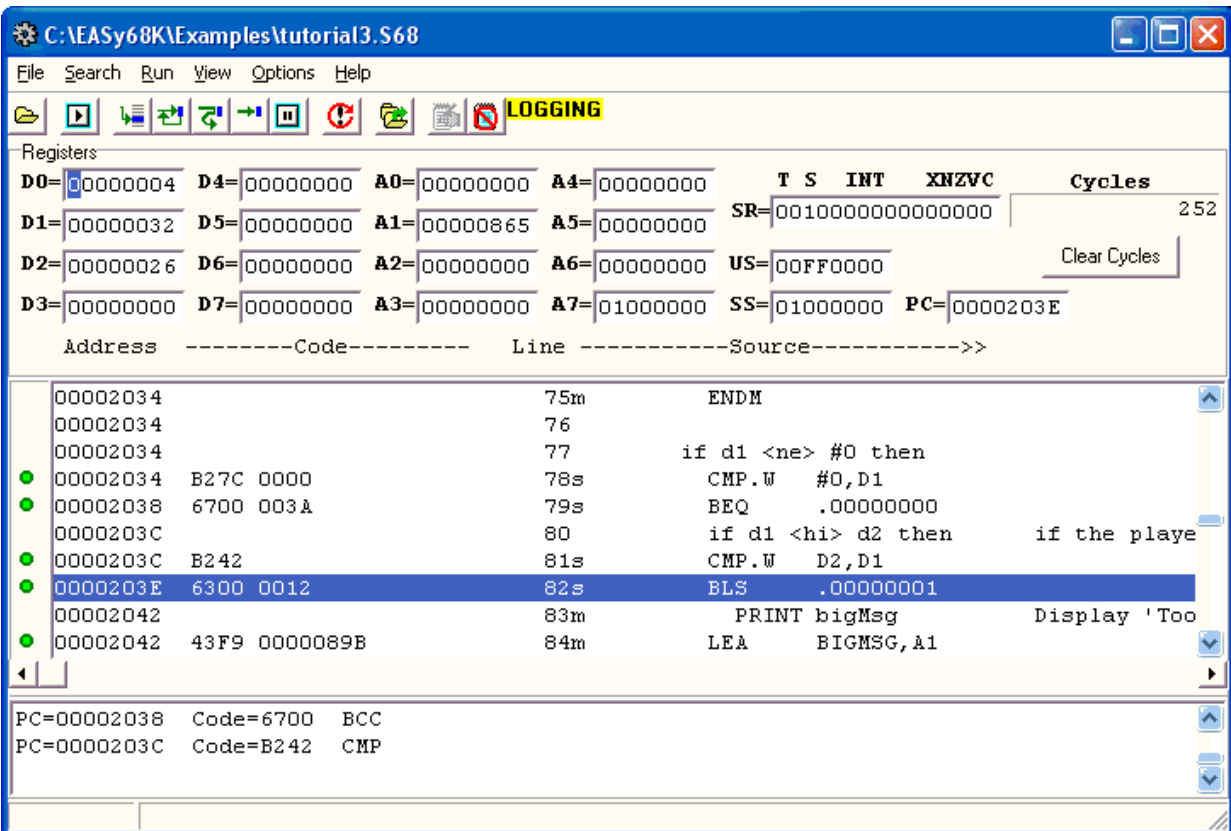
Open Data - Loads a 68000 data from S-Record (.S68) file into the simulator's 68000 memory space without changing the simulator settings.

Close - Clears the current program from the display.

Printer Setup... - Setup the printer used for simulator output. See help for [Simulator I/O task 10](#) for more information.

Once you have loaded a program, the simulator will look similar to the following window:

Note: If no matching "Listing" (.L68) file is available, no code will display in the code window.



Set the Program Counter (PC) to the desired starting address by manually entering the address into the PC field or by double clicking on the desired line of program code. By default the PC is set to the starting address of the program. You may also change any of the 68000's other registers at any time while the simulator is in Stop or Pause mode.

All numbers in the registers are displayed using Hexadecimal notation except the Status Register which is in Binary.

The registers displayed are as follows:

D0= ... D7= Data registers

A0= ... A7= Address registers

SR= Status register

US= User Stack (The user stack is the same as A7 when the S bit in the Status Register is set to 0).

SS= System Stack (The system stack is the same as A7 when the S

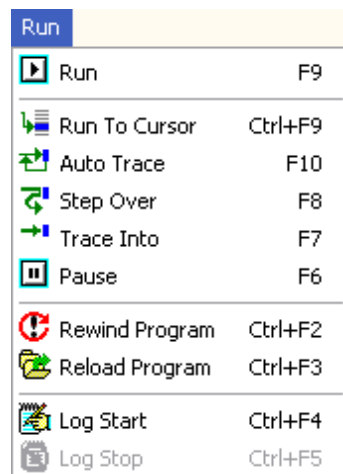
bit in the Status Register is set to 1).

PC= Program Counter

Cycles displays the estimated cycle time required for the previously executed instructions. The cycle counter does not take into account pipeline efficiencies. The cycle number displayed tends to be a worst case value and is only intended for relative comparison purposes.

[Trap tasks 30 and 31](#) may be used to clear and read the cycle counter from a program.

=== Running a Program ===



Run

To run the program, select Run from the Run Menu, press F9 or click the Run button on the toolbar. Sim68K begins executing the 68000 program at the current Program Counter location. Program execution will continue until one of the following occurs:

- The program reaches a STOP instruction.
- The program reaches a user placed Break-Point
- The user Pauses the program.
- The user Resets the simulator.
- An exception occurs.



Run To Cursor - Program execution will continue until the Program Counter reaches the Cursor line (highlighted line) or until any of the stop conditions listed above in the Run command occurs. Run To Cursor may be selected from the Run Menu or by pressing Ctrl-F9 or by clicking the Run To Cursor button.



Auto Trace - Automatically activates a Trace Into at the specified time interval. The time interval may be adjusted by selecting Auto Trace Options in the Options menu. To start Auto Trace, select Auto Trace from the Run Menu, press F10 or click the Auto Trace button.



Step Over - Executes the current instruction and positions the Program Counter at the instruction in the next line. If the current instruction is a JSR or BSR the subroutine is completely executed and the Program Counter is placed at the instruction following the JSR or BSR. To Step through a program, select Step Over from the Run Menu, press F8 or click the Step Over button.



Trace Into - Executes the current instruction and positions the Program Counter at the next instruction to be executed. If the current instruction is a JSR or BSR the program counter is placed at the first instruction of the subroutine. To Trace through a program, select Trace Into from the Run Menu, press F7 or click the Trace Into button.

When Tracing or Stepping through a program, the next line to be executed is highlighted as shown above.

The Auto Trace, Trace and Step buttons will be disabled if the program is waiting for input.



Pause

Pauses program execution and enables the menus. To Pause a running program select Pause from the Run Menu, press F6 or click the Pause button.



Rewind Program

Clears the Output Window, clears the 68000 registers and places the Program Counter at the beginning of the program. To Rewind a program select Rewind Program from the Run Menu, press Ctrl+F2 or click the Rewind Program button.



Reload Program

Reloads the last program into the simulator, clears the Output Window, clears the 68000 registers and places the Program Counter at the beginning of the program. To Reload a program select Reload Program from the Run Menu, press Ctrl+F3 or click the Reload Program button.



Log Start

Starts logging. See [Options](#) for help on configuring the log types.



LOGGING

is displayed on the toolbar when logging is in progress.



Log Stop

Stops logging. See [Options](#) for help on configuring the log types.

Running a Program From a Batch File

Batch files may be used to start Sim68K, load a 68000 .S68 program and set several simulator options. The following command line arguments are recognized by Sim68K for this purpose:

The first argument must be the name of the 68000 .S68 file to load followed by the optional arguments:

/f Full Screen

/e Enable Exceptions

/r Run

/b Enable Bit Field Instructions

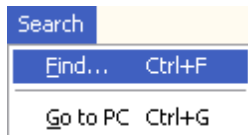
For example, if a file named easyzone.bat is created that contains the following text:

```
"c:\easy68k\SIM68K" easyzone.s68 /r /f /e
```

The 68000 program easyzone.s68 would be loaded into Sim68K and

run in full screen mode with exceptions enabled. This example assumes SIM68K is located in "C:\easy68K" and also assumes easyzone.s68 is located in the same folder as the batch file.

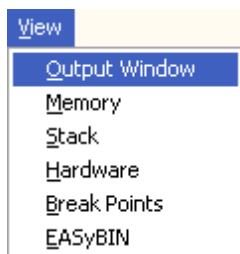
=== Search Menu ===



Press Ctrl+F or select Find... from the Search menu to open a find dialog. Find searches the code window for the specified text. The search is not case sensitive and will match partial words. The corresponding line in the code window will be highlighted if the search text is found. Press Ctrl+G or select Go to PC from the Search menu to return to the Program Counter location in the code window.

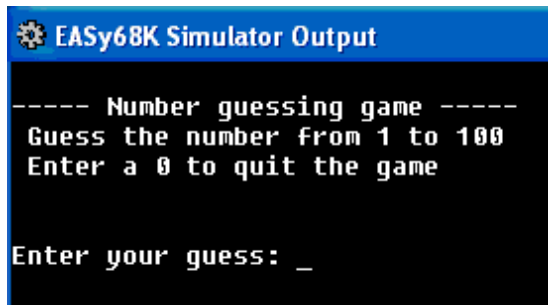
=== View Menu ===

Use the View menu to select between the different Sim68K windows. You may also press Ctrl+Tab to cycle between the currently active simulator windows. The EASyBIN selection will start the EASyBIN application if the EASyBIN.exe file is located in the same folder as Sim68K.



=== Output Window ===

To see what your program has displayed, select the View menu and then click "Output Window"

A screenshot of the 'EASy68K Simulator Output' window. The window has a blue title bar with a gear icon and the text 'EASy68K Simulator Output'. The main area is black with white text. It displays a 'Number guessing game' with instructions to guess a number from 1 to 100 and to enter 0 to quit. It prompts the user to 'Enter your guess:' followed by a cursor.

The output window also serves as a way to enter data into your program. See [Text I/O](#)

=== Simple breakpoints ===

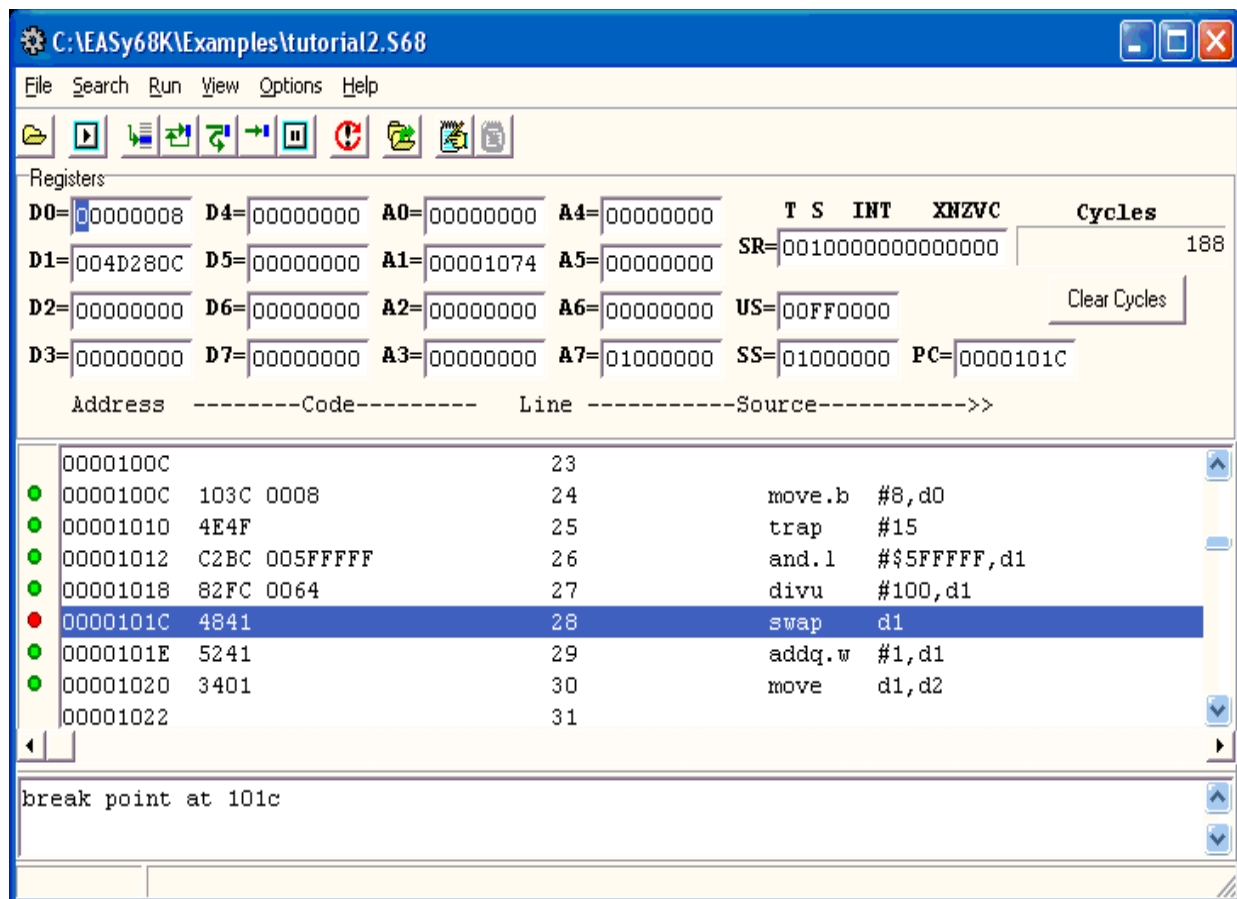
A breakpoint is used to halt a program. You can place a simple breakpoint in the your program by clicking the green dot next to the line of code you want to break at.

Once set, the dot will turn red as shown:



You may clear the breakpoint by clicking the dot again.

When the program reaches the breakpoint it will halt prior to running the instruction at the breakpoint. The contents of the 68000 registers are displayed and may be modified. The program may be resumed using the Run button or using Trace and Step. The breakpoint remains active until removed.



=== Embedding breakpoints in the source ===

Breakpoints may be embedded in the 68000 source file using the following form:

*[sim68k]break

Any source line that begins with the above text will automatically set a PC breakpoint on the following line of code when loaded into Sim68K.

Key Codes:

All letter keys correspond to the ASCII code for the capital form of the letter.

All number keys on the top row correspond to their ASCII code.

Modifier keys (Shift, Alt, Ctrl) do not change the key code returned.

Backspace = \$8
keypad 5 = \$C
Enter or keypad Enter = \$D
Shift = \$10
Control = \$11
Alt = \$12
Pause = \$13
Caps Lock = \$14
Esc = \$1B
Space = \$20
Page Up or keypad 9 = \$21
Page Down or keypad 3 = \$22
End or keypad 1 = \$23
Home or keypad 7 = \$24
Left Arrow or keypad 4 = \$25
Up Arrow or keypad 8 = \$26
Right Arrow or keypad 6 = \$27
Down Arrow or keypad 2 = \$28
Insert or keypad 0 = \$2D
Delete or keypad . = \$2E
keypad * = \$6A
keypad + = \$6B
keypad - = \$6D
keypad / = \$6F
F1 = \$70 used by Sim68K ¹
F2 = \$71
F3 = \$72
F4 = \$73

F5 = \$74
F6 = \$75 used by Sim68K ¹
F7 = \$76 used by Sim68K ¹
F8 = \$77 used by Sim68K ¹
F9 = \$78 used by Sim68K ¹
F10 = \$79 used by Sim68K ¹
F11 = \$7A
F12 = \$7B
Num Lock = \$90
Scroll Lock = \$91
; = \$BA
'=' = \$BB
, = \$BC
- = \$BD
. = \$BE
/ = \$BF
` = \$C0
[= \$DB
\ = \$DC
] = \$DD
' = \$DE

¹ - use trap task #24 to disable simulator shortcut keys

EXG Instruction

Lets two data or address registers swap contents with each other.
You can swap a data register with an address register.

ADDRESS METHODS: Dn, An

DATA LENGTH: Longword

FLAGS: Unaffected

SYNTAX: EXG Rx,Ry

EXAMPLE CODE:

```
EXG    D0,D1    *exchanges the contents of D0 with D1
```


LEA Instruction

Loads an effective address into an address register. LEA is often used when writing code that must be independent of the position in the memory (which all Fargo programs are). It's often used with the address methods x(PC) or x(PC,xr.s).

LEA also adds a constant to an address register without changing the flags, and/or also an index with x(An,xr.s).

ADDRESS METHODS: (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Longword

FLAGS: Unaffected

SYNTAX: LEA <ea>,An

EXAMPLE CODE:

```
LEA    memorylocation,A1    *loads the address of a memory
location into A1. This is commonly used in the trap statement when
you are outputting strings to the screen
```

LINK Instruction

The instruction LINK creates a stack frame, a temporarily allocated piece of memory on the stack. The instruction is often used in high-level languages when allocating memory for local variables in procedures. When the procedure is deactivated, the variables disappear which saves memory. LINK has two operands. The first is an address register and the second is a two's complement value, which is the size of the frame (often negative since the stack grows backwards). When the instruction is executed, the address register is pushed onto the stack and the newly updated A7 is copied to the address register. The immediate value is added to the Stack register (A7).

The local memory is accessed by negative shifting the address register. That way you can use local variables without worrying about other data that is pushed or popped from the stack. This is provided you keep in the range you specified when you called the LINK opcode. The instruction UNLK removes the stack frame and restores the stack pointer.

ADDRESS METHODS: None

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: LINK An, #<shifting>

Note that the use of LINK allows Reentrant programs.

MOVE Instruction

The instruction MOVE copies a byte, word, or longword from one effective address to another. The flags are set according to the data moved.

ADDRESS METHODS	Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l
(source):	x(PC), x(PC,xr.s), #x

ADDRESS METHODS (for the destination): Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

The address method An can only be used when the data length is a word or a longword.

FLAGS: X - U
N - S
Z - S
C - 0
V - 0

SYNTAX: MOVE <ea>,<ea>

The instruction MOVEA moves data to an address register (An is, as you may have noticed missing in address methods for the destination). Most assemblers choose MOVEA if you have an address register as an operand.

EXAMPLE CODE:

```
MOVE.B  D0,D1    moves the lower 8 bits of D0 to D1, does
not change the upper 24 bits of D0 or D1
MOVE.W  D0,D1    moves the lower 16 bits of D0 to D1, does
```

not change the upper 16 bits of D0 or D1

MOVE.L D0,D1 moves all 32 bits of D0 to D1

MOVE to CCR

If you specify CCR as the destination, the lower byte of a word is copied to the flag register (CCR). The flags do are not affected by the result, i.e. if you clear the flag register the Z flag won't be set.

ADDRESS Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l,x(PC),
METHODS: x(PC,xr.s), #x

DATA LENGTH: Word

FLAGS: Set according to the bits of the byte you moved to CCR.

SYNTAX: MOVE <ea>,CCR

EXAMPLE CODE:

```
MOVE    D0,CCR    moves the lower 16 bits of D0 into the CCR
```

MOVE to SR

If you specify SR as the destination, a word is moved to the status register (i.e. the system byte and the flag byte). The instruction requires that the supervisor bit is set. The instruction can be used to change the T-bit (trace), S-bit (supervisor), the interrupt mask and the flags. Note that with Fargo II, to get into supervisor mode, you should use the trap #1.

If you only want to change the flags, you should use MOVE to CCR instead, which also works if you are in user mode.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s), #x

DATA LENGTH: Word

FLAGS: Set according to the lower bits in the word you moved to SR.

SYNTAX: MOVE <ea>,SR

EXAMPLE CODE:

```
MOVE.W D0,SR *Moves the lower word in D0 to the SR
```

MOVE from SR

This instruction copies the whole status register to an operand with the size of a word. It requires that you are in supervisor mode (the S-bit in SR must be set).

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Word

FLAGS: Unaffected

SYNTAX: MOVE SR,<ea>

EXAMPLE CODE:

```
MOVE.W  SR,D0    *Moves the SR into the lower word of D0
```

MOVEA Instruction

The instruction MOVEA copies an operand given by an effective address to an address register. Only words and longwords are used, and all 32 bits in the destination are affected (sign extended if word).

ADDRESS Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC),
METHODS: x(PC,xr.s), #x

DATA LENGTH: Longword

FLAGS: Unaffected

SYNTAX: MOVEA <ea>,An

Most assemblers choose MOVEA if you use MOVE with an address register as the destination.

EXAMPLE CODE:

```
MOVEA  D0,A1  moves the contents of D0 into A1
```


MOVEM Instruction

The instruction MOVEM (MOVE Multiple) makes it possible to fast transfer a group of registers to or from memory. The operation only works with words and longwords. When you move words to registers, the words are sign extended so all 32 bits are affected. The instruction is most often used to push registers onto the stack, so you can temporarily use those register for other purposes, and later restore the original values. Very useful in the beginning and the end of subroutines that change a lot of registers.

ADDRESS METHODS: (An), -(An), x(An), x(An,xr.s), x.w, x.l

ADDRESS METHODS: (An), (An)+, x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Word, longword

FLAGS: Unaffected

SYNTAX: MOVEM <register list>,<ea>
 MOVEM <ea>,<register list>

The register list is a series of registers separated by a slash ("/"). You can also use one of many intervals (shown with a "-"). For example, D0-D5/A0-A2 means the registers D0, D1, D2, D3, D4, D5, A0, A1, A2.

EXAMPLE CODE:

```
MOVEM  D0-D7/A1-A4,list  moves all of the D resisters and 1
through 4 of the address resisters to memory
```

MOVEP Instruction

The instruction MOVEP (MOVE Peripheral) transfers a word or longword in a data register to or from every second memory address. The MSB in the data register transfers to or from the address $x(An)$, - the only address method - the next byte to or from $x+2(An)$ and so on. If the first address was odd all MOVEP transfers will use the lowest byte in the 68000's databus. Even addresses use the highest byte. This instruction is mainly used when you use 8 bit data buses connected with the 16 bit databus in the 68k. Since this is never done on the TI-92 this instruction won't be used that way.

ADDRESS METHODS: $x(An)$

DATA LENGTH: Word, longword

FLAGS: Unaffected

SYNTAX: MOVEP Dn, $x(An)$
MOVEP $x(An)$,Dn

MOVEQ Instruction

The instruction MOVEQ (MOVE Quick) is used to put small immediate data in a data register. The instruction is two bytes long and can handle constants in the range -128 and +127 (decimal). All 32 bits in the register are affected. If you used MOVE.L the instruction would take 6 bytes.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s), #x

DATA LENGTH: Longword

FLAGS: X - U
N - S
Z - S
C - 0
V - 0

SYNTAX: MOVEQ #<data>,Dn

Many assemblers automatically change a MOVE to a MOVEQ, if possible.

EXAMPLE CODE:

```
MOVEQ.L  #3,D0  *puts 3 into D0, but affects the entire
longword, quicker than move
```

PEA Instruction

The instruction PEA calculates an effective address and pushes it onto the stack. The address is always a longword.

ADDRESS METHODS: (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Longword

FLAGS: Unaffected

SYNTAX: PEA <ea>

EXAMPLE CODE:

PEA num1 similar to LEA except this pushes the address to the stack, rather than a register

SWAP Instruction

The instruction SWAP swaps the upper word with the lower word in a data register (bit 31-16 is exchanged with bit 15-0).

ADDRESS METHODS: Dn

DATA LENGTH: Word

FLAGS: X - U

N - Same as bit 31 in the data register

Z - Set if all 32 bits are zero, else cleared

C - 0

V - 0

SYNTAX: SWAP Dn

EXAMPLE CODE:

SWAP D0 if D0 contains FFFF0000, it would become 0000FFFF, which is useful for packing and unpacking BCDs

UNLK Instruction

The instruction UNLK (UNLink) removes a stack frame that was set earlier by the instruction LINK (see LINK). It works like this, the given address register (usually the frame pointer) is stored in the stack pointer. Then the original state of the frame pointer is restored by getting the first longword on the stack. This is the opposite of what LINK does. UNLK works correctly no matter what has been pushed or popped on the stack between the instructions LINK and UNLK but it requires that you popped your variables from the frame pointer or updated the frame pointer to its initial value (by adding the size of the variables to the frame pointer for example)

ADDRESS METHODS: N/A

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: UNLK An

ADD Instruction

Adds two binary operands and stores the result in the destination operand.

Two different methods are allowed:

1. Add an effective address to a data register.
2. Add a data register to an effective address.

ADDRESS 1) Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC),
METHODS: x(PC,xr.s), #x

ADDRESS METHODS: 2) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

When using an address register as destination, byte is not allowed.

FLAGS: X - Set if carry from the most significant bit,
 else cleared.
 N - S
 Z - S
 C - Same as X
 V - S

SYNTAX: ADD Dn,<ea>
 ADD <ea>,Dn

EXAMPLE CODE:

```
ADD.B  D0,D1    adds the lower 8 bits of D0 to D1, does not  
change the upper 24 bits of D0 or D1  
ADD.W  D0,D1    adds the lower 16 bits of D0 to D1, does not  
change the upper 16 bits of D0 or D1  
ADD.L  D0,D1    adds all 32 bits of D0 to D1
```


ADDA Instruction

Binary addition to an address register. To make it possible to mix address operations with data operations, this instruction will not affect any flags.

ADDRESS METHODS: D_n , A_n , (A_n) , $(A_n)+$, $-(A_n)$, $x(A_n)$, $x(A_n, x.r.s)$, $x.w$, $x.l$, $x(PC)$, $x(PC, x.r.s)$, $\#x$

The effective address must be the source.

DATA LENGTH: Word, longword

ADDA always affects all 32 bits in the destination address register.

FLAGS: Unaffected

SYNTAX: ADDA <ea>, A_n

EXAMPLE CODE:

```
ADDA  A1,A2  *adds the address in A1 to A2
```

ADDI Instruction

Adds a constant to an effective address. The source operand must be immediate.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - Set if carry from the most significant bit, otherwise it is cleared.

N - S

Z - S

C - Same as X

V - S

SYNTAX: ADDI #x,<ea>

Most assemblers automatically choose ADDI if the source operand to an ADD instruction is immediate.

ADDQ Instruction

This instruction adds a three bit immediate value to an effective address. The instruction is very quick and much shorter than the usual ADD.

ADDRESS METHODS: Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

When using an address register as destination, byte is not allowed.

FLAGS: X - Set if carry from the most significant bit,
otherwise it is cleared.
N - S
Z - S
C - Same as X
V - S

No flags are affected if the destination operand is an address register.

SYNTAX: ADDQ #<data>,<ea>

#<data> is a constant between 1 and 8.

ADDX Instruction

The instruction ADDX (ADD eXtended) works as ADD but the X flag is also added. This makes it possible to add big numbers stored in many bytes.

The instruction has two methods:

1. Add a data register to a data register.
2. Add a memory location to another memory location. You must use -(An) on both operands then.

ADDRESS METHODS: Dn, -(An)

DATA LENGTH: Byte, word, longword

FLAGS: X - Set if carry from the most significant bit,
otherwise it is cleared.
N - S
Z - S
C - Same as X
V - S

The Z flag works in another way now, making it possible to check if a big number (much bigger than 32 bits) is zero. You must set the zero flag before making the addition though, shorter than comparing a register with itself.

SYNTAX: ADDX Dy,Dx
 ADDX -(Ay),-(Ax)

EXAMPLE CODE:

ADDX D0,D1 Adds D0 and D1 + X bit. then clears the X bit

CLR Instruction

This instruction (CLear) clears an operand specified with an effective address.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - U
N - 0
Z - 1
C - 0
V - 0

SYNTAX: CLR <ea>

EXAMPLE CODE:

```
CLR    D0    clear the low word of D0 to zero
```

CMP Instruction

CMP (CoMPare) compares a data register with an effective address. The flags are affected the same way as if the effective address was subtracted from the data register. None of the operands are changed. Often used with the Bcc instruction. An example:

```
CMP D0,D1  
BGT X1
```

The program will branch to X1 if D1 is greater than D0, else the program will continue.

ADDRESS Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC),
METHODS: x(PC,xr.s), #x

DATA LENGTH: Byte, word, longword

Byte can't be used when comparing to an address register.

FLAGS: X - U
 N - S
 Z - S
 C - Set if a borrow was needed when subtracting,
 otherwise it is cleared.
 V - S

SYNTAX: CMP <ea>,Dn

There are four CMP instructions, CMP, CMPA, CMPI and CMPM. The compiler often chooses the right instruction, so you can write CMP all the time if you want.

```
CMP    #2,D0    compares D0 to #2 and sets CCR flags  
accordingly
```

CMPA Instruction

CMPA (CoMPare Address) compares an address register with an effective address. The flags are affected the same way as if the effective address was subtracted from the data register. None of the operands are changed. Often used with the Bcc instruction. An example:

```
CMP A0,A1  
BGT X1
```

The program will branch to X1 if A1 is greater than A0, otherwise the program will continue.

ADDRESS Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC),
METHODS: x(PC,xr.s), #x

DATA LENGTH: Word, longword

FLAGS: X - U
 N - S
 Z - S
 C - Set if a borrow was needed when subtracting,
 otherwise it is cleared.
 V - S

SYNTAX: CMPA <ea>,An

There are four CMP instructions, CMP, CMPA, CMPI and CMPM. The compiler often chooses the right instruction, so you can write CMP all the time if you want.

CMPI Instruction

CMPI (CoMPare Immediate) compares an immediate value with an effective address. The flags are affected the same way as if the effective address was subtracted from the data register. None of the operands are changed. Often used with the Bcc instruction. An example:

```
CMP #<data>,D0  
BGT X1
```

The program will branch to X1 if D0 is greater than the immediate value.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - U

N - S

Z - S

C - Set if a borrow was needed when subtracting,
otherwise it is cleared.

V - S

SYNTAX: CMPI <ea>,Dn

There are four CMP instructions, CMP, CMPA, CMPI and CMPM. The compiler often chooses the right instruction, so you can write CMP all the time if you want.

CMPM Instruction

CMPM (CoMPare Memory) compares two memory locations with each other with after-increment. The flags are affected the same way as if the source was subtracted from the destination. None of the operands are changed.

ADDRESS METHODS: (An)+

DATA LENGTH: Byte, word, longword

FLAGS: X - U

N - S

Z - S

C - Set if a borrow was needed when subtracting,
otherwise it is cleared.

V - S

SYNTAX: CMPM (Ay)+,(Ax)+

There are four CMP instructions, CMP, CMPA, CMPI and CMPM. The compiler often chooses the right instruction, so you can write CMP all the time if you want.

DIVS Instruction

DIVide Signed. The instruction divides a signed (two complement) data register with an operand specified as an effective address. The data register is 32 bit and the effective address is a word. The result is stored in the lower 16 bits of the data register and the remainder in the upper 16 bits. Two errors can arise with DIVS:

1. If you try to divide with 0 an interrupt will occur.
2. If the result doesn't fit in 16 bits the data register will remain unchanged and the V-flag will be set (overflows).

ADDRESS Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s),
METHODS: #x

DATA LENGTH: Word

FLAGS: X - U
 N - S, undef if overflow
 Z - S, undef if overflow
 C - 0
 V - S

SYNTAX: DIVS <ea>,Dn

EXAMPLE CODE:

DIVS #4,D0 if D0 contained 9, D0 would now contain 00010002, 1 remainder, 2 from division

DIVU Instruction

DIVide Unsigned. The instruction divides an unsigned data register with an operand specified as an effective address. The data register is 32 bit and the effective address is a word. The result is stored in the lowest 16 bits of the data register and the remainder in the upper 16 bits. Two errors can arise with DIVU:

1. If you try to divide with 0 an interrupt will occur.
2. If the result doesn't fit in 16 bits the data register will remain unchanged and the V-flag will be set (overflows).

ADDRESS Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s),
METHODS: #x

DATA LENGTH: Word

FLAGS: X - U
 N - S, undef if overflow
 Z - S, undef if overflow
 C - 0
 V - S

SYNTAX: DIVU <ea>,Dn

EXAMPLE CODE:

```
DIVU  #4,D0  if D0 contained 9, D0 would now contain
00010002, 1 remainder, 2 from division
```


EXT Instruction

The instruction EXT (Sign EXTend) makes a sign extension, byte to a word or a word to a longword. If you extend a byte to a word, bit 7 is copied to bit 15-8. If you extend a word to a longword, bit 15 is copied to bit 31-16.

ADDRESS METHODS: Dn

DATA LENGTH: Word, Longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: EXT Dn

EXAMPLE CODE:

```
EXT.L D0    if D0 contained 0000FFF5, it would change D0 to  
FFFFFFFF5
```

MULS Instruction

The instruction MULS (MULTiply Signed) multiplies a 16 bit operand in a data register with a 16 bits operand specified as an effective address. The result is a 32 bit value that is stored in the data register. All operands are signed (twos complement).

ADDRESS Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s),
METHODS: #x

DATA LENGTH: Word

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: MULS <ea>,Dn

EXAMPLE CODE:

```
MULS   D1,D2   multiplies D1 and D2, and stores it in D2
```

MULU Instruction

The instruction MULU (MULTiply Unsigned) multiplies a 16 bit operand in a data register with a 16 bit operand specified as an effective address. The result is a 32 bit value that is stored in the data register. All operands are unsigned.

ADDRESS Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s),
METHODS: #x

DATA LENGTH: Word

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: MULU <ea>,Dn

EXAMPLE CODE:

```
MULU   D1,D2   multiplies D1 and D2, and stores it in D2
```


NEG Instruction

The instruction NEG returns the twos complement of an operand given as an effective address.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - Clear if the result is zero, otherwise it is set.
 N - S
 Z - S
 C - Same as X
 V - S

SYNTAX: NEG <ea>

EXAMPLE CODE:

```
NEG    D0.W    if D0 contained 2, D0 would become 0000FFFE
```

NEGX Instruction

The instruction NEGX (NEGate with eXtend) returns the twos complement from a binary number with multi-precision.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - Set if loan, else cleared
 N - S
 Z - Set if the result is not zero, else
 unaffected
 C - Same as X
 V - S

SYNTAX: NEGX <ea>

EXAMPLE CODE:

```
NEGX D0.W   if D0 contained 2, X = 1, D0 would become 0000FFFD
```

SUB Instruction

Subtracts two binary operands and stores the result in the destination operand.

Two different methods are allowed:

1. Subtract an effective address from a data register.
2. Subtract a data register from an effective address.

ADDRESS METHODS: 1) Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s), #x

ADDRESS METHODS: 2) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

When using an address register as destination, byte is not allowed.

FLAGS: X - Set if a loan was required from the most
 significant bit, otherwise 0.
 N - S
 Z - S
 C - Same as X
 V - S

SYNTAX: SUB Dn,<ea>
 SUB <ea>,Dn

When an address register is the destination, you use SUBA. Many assemblers will automatically choose SUBA if you write SUB with an address register as destination.

EXAMPLE CODE:

```
SUB.B  D0,D1    subtracts the lower 8 bits of D0 from D1, does
not change the upper 24 bits of D0 or D1
SUBW   D0,D1    subtracts the lower 16 bits of D0 from D1,
does not change the upper 16 bits of D0 or D1
SUB.L   D0,D1    subtracts all 32 bits of D0 from D1
```

SUBA Instruction

Binary subtraction from an address register. To make it possible to mix address operations with data operations, this instruction will not affect any flags.

ADDRESS Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC),
METHODS: x(PC,xr.s), #x

The effective address must be the source.

DATA LENGTH: Word, longword

SUBA affects always all 32 bits in the destination address register.

FLAGS: Unaffected

SYNTAX: SUBA <ea>,An

EXAMPLE CODE:

```
SUBA  A1,A2  subtracts A1 from A2
```

SUBI Instruction

Subtracts a constant from an effective address.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - Set if a loan was required from the most
 significant bit, otherwise 0.
 N - S
 Z - S
 C - Same as X
 V - S

SYNTAX: SUBI #x,<ea>

Most assemblers automatically choose SUBI if the source operand to an SUB instruction is a constant.

SUBQ Instruction

This instruction subtracts a three bit constant from an effective address. The instruction is very quick and much shorter than the usual SUB.

ADDRESS METHODS: Dn, An, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

When using an address register as destination, byte is not allowed.

FLAGS: X - Set if a loan was required from the most
 significant bit, otherwise 0.
 N - S
 Z - S
 C - Same as X
 V - S

No flags are affected if the destination operand is an address register.

SYNTAX: SUBQ #<data>,<ea>

#<data> is a constant between 1 and 8.

SUBX Instruction

The instruction SUBX (SUBtract eXtended) works the same as SUB but the X flag is also subtracted. This makes it possible to add big numbers stored in many bytes (multi-precision).

This instruction has two methods:

1. Subtract a data register from a data register.
2. Subtract a memory location to another memory location. You must use -(An) on both operands then.

ADDRESS METHODS: Dn, -(An)

DATA LENGTH: Byte, word, longword

FLAGS: X - Set if a loan was required from the most significant bit, else 0.
N - S
Z - Cleared if the result is not zero, else unaffected
C - Same as X
V - S

The Z flag works in another way now, making it possible to check if a big number (much bigger than 32 bits) is zero. You must set the zero flag before making the addition though, shorter than comparing a register with itself.

SYNTAX: SUBX Dy,Dx
SUBX -(Ay),-(Ax)

EXAMPLE CODE:

```
SUBX.B D0,D1 D1 = D1 - D0 - X
```


TAS Instruction

The instruction TAS (Test And Set) examines a specified byte with an effective address. The most significant bit in the byte is set. The N- and Z-flags are set according to the bytes value before the operation. The instruction reads, modifies and sets and can't be interrupted. The instruction is used to synchronize if two or more MPU use the same memory area. Since TAS can't be interrupted a MPU can mark a location in the memory to be "busy" before another processor can examine it. If the operation was interruptible, two processor could examine the same byte at the same time, and both processor would think the byte was free to use, which would lead to an error. TAS guaranties that one processor wins and the other loses. TAS is used by high level languages to implement "Semaphores"

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

You can use TAS with a data register, but then it has nothing to do with synchronization.

DATA LENGTH: Byte

FLAGS: X - U
 N - S (before the operation)
 Z - S (before the operation)
 C - 0
 V - 0

SYNTAX: TAS <ea>

TST Instruction

The instruction TST examines an operand specified as an effective address, and finds out if it's zero or negative. The flags are set depending on the result.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: TST <ea>

AND Instruction

The instruction AND performs the logical operation "AND", bit for bit.

1. The source is and effective address, the destination is a data register.
2. The source is the data register and the destination is the effective address.

ADDRESS 1) Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC),
METHODS: x(PC,xr.s), #x
 2) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: AND <ea>,Dn
 AND Dn,<ea>

EXAMPLE CODE:

```
AND #01111000,D0    If D0 contained 00111100, it would then contain  
00110000
```

Use AND to selectively clear bits in the destination. Each bit in the source operand that is 0 will cause the corresponding bit in the destination to be 0. Each bit in the source that is 1 will leave the corresponding bit in the destination unchanged. For example:

SOURCE	0000	0101
DESTINATION	0011	0011
-----	----	----
RESULT	0000	0001

ANDI Instruction

Same as AND except that the source is an immediate value.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, SR, CCR

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: ANDI #<data>,<ea>

OR Instruction

The instruction OR performs the logical operation "OR", bit for bit. There are two ways to do this:

1. The source is an effective address, the destination is a data register.
2. The source is the data register and the destination is the effective address

ADDRESS 1) Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC),
METHODS: x(PC,xr.s), #x
 2) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: OR <ea>,Dn
 OR Dn,<ea>

EXAMPLE CODE:

```
OR    #00001111,D0    If D0 contained 11001100,after the or it
would contain 11001111
```

Use OR to selectively set bits in the destination. Each bit in the source operand that is 1 will cause the corresponding bit in the

destination to be set to 1. Each bit in the source that is 0 will leave the corresponding bit in the destination unchanged. For example:

SOURCE	1111	0101
DESTINATION	0011	0011
-----	----	----
RESULT	1111	0111

ORI Instruction

Same as OR except that the source is an immediate value.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, SR, CCR

When using SR, the Supervisor bit must be set.

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: ORI #<data>,<ea>

EOR Instruction

EOR performs an exclusive OR between a data register and the memory. EOR gives the result (binary) 1 if one, and only one, of the operators are 1 (compare with OR). The data register must be the source and the effective address the destination.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: EOR Dn,<ea>

EXAMPLE CODE:

```
EOR    #%11110000,D0    if D0 contains 01010101, then after the
EOR    it would have 10100101
```

Use EOR to selectively invert bits in the destination. Each bit in the source operand that is 1 will cause the corresponding bit in the destination to be inverted. Each bit in the source that is 0 will leave the corresponding bit in the destination unchanged. For example:

SOURCE	1111	0101
DESTINATION	0011	0011

RESULT

1100

0110

EORI Instruction

EORI performs an exclusive OR between an immediate value and an effective address. EORI gives the result (binary) 1 if one, and only one, of the operators are 1 (compare with OR). The effective address must be the destination of course.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, SR, CCR

Operations that uses the status register (SR) and the flag register (CCR) can only work with word and byte. If you try to change SR you must be in supervisor mode, else an interrupt occurs. (use trap #1 to change the mode)

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

If the status register or the flag register is the destination, the flags are set the same way as any other effective address. If an instruction clears all bits in the flag register, the Z flag won't be set (as it should since the result was 0).

SYNTAX: EORI #<data>,<ea>

NOT Instruction

Returns the ones complement of an operand specified with an effective address. The ones complement is the same as changing all bits in the operand.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - U
 N - S
 Z - S
 C - 0
 V - 0

SYNTAX: NOT <ea>

EXAMPLE CODE:

NOT D0 if D0 contained 01010101, then after the not, it would contain 10101010

ASL Instruction

Performs an arithmetic shift to the right on a data register or a memory location. There are three possibilities:

1. Shift a data register to the left. Number of steps is stored in another data register.
2. Shift a data register to the left. Number of steps is an immediate value. Shift range 1-8 bits.
3. Shift a word in the memory one bit to the left.

ADDRESS METHODS: 3) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - The last bit that was sent out of the operand. Unaffected if number of steps is 0.

N - S

Z - S

C - Same as X

V - Set if the most significant bit was changed during the operation, otherwise it is cleared.

ASL Dx,Dy

SYNTAX: ASL #<data>,Dy

ASL <ea>

EXAMPLE CODE:

```
ASL.B #1,D0 if D0.B contained 00001111, it would now be 00011110
```


ASR Instruction

Performs an arithmetic shift to the right on a data register or a memory location. There are three possibilities:

1. Shift a data register to the right. Number of steps is stored in another data register.
2. Shift a data register to the right. Number of steps is an immediate value. Shift range 1-8 bits.
3. Shift a word in the memory one bit to the right.

ADDRESS METHODS: 3) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - The last bit that was sent out of the operand. Unaffected if number of steps is 0.

N - S

Z - S

C - Same as X

V - Set if the most significant bit was changed during the operation, otherwise it is cleared.

ASR Dx,Dy

SYNTAX: ASR #<data>,Dy

ASR <ea>

EXAMPLE CODE:

```
ASR.B  #1,D0  if D0.B contained 00011110, it would now be 00001111
```

LSL Instruction

Performs a logical shift to the left on a data register or a memory location. There are three possibilities:

1. Shift a data register to the left. Number of steps is stored in another data register.
2. Shift a data register to the left. Number of steps is an immediate value. Shift range 1-8 bits.
3. Shift a word in the memory one bit to the left.

ADDRESS METHODS: 3) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - The last bit that was sent out of the operand. Unaffected if number of steps is 0.

N - S

Z - S

C - Same as X

V - Set if the most significant bit was changed during the operation, otherwise it is cleared.

LSL Dx,Dy

SYNTAX: LSL #<data>,Dy

LSL <ea>

EXAMPLE CODE:

```
LSL.B  #1,D0  if D0.B contained 00001111, it would now be 00011110
```


LSR Instruction

Performs a logical shift to the right on a data register or a memory location. There are three possibilities:

1. Shift a data register to the right. Number of steps is stored in another data register.
2. Shift a data register to the right. Number of steps is an immediate value. Shift range 1-8 bits.
3. Shift a word in the memory one bit to the right.

ADDRESS METHODS: 3) (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, word, longword

FLAGS: X - The last bit that was sent out of the operand. Unaffected if number of steps is 0.

N - S

Z - S

C - Same as X

V - Set if the most significant bit was changed during the operation, otherwise it is cleared.

LSR Dx,Dy

SYNTAX: LSR #<data>,Dy

LSR <ea>

EXAMPLE CODE:

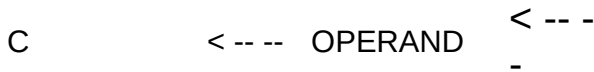
LSR.B #1,D0 if D0.B contained 00011110, it would now be 00001111

ROL Instruction

The instruction ROL rotates a data register or a memory operand to the left. There are three ways to do this:

1. Rotate a data register to the left. Number of steps is a constant.
2. Rotate a data register to the left. Number of steps is stored in another data register. You can rotate 1-8 bits this way.
3. Rotate a word in the memory one bit to the left.

The rotation is done without an extra bit (i.e. a 8, 16 or 32 bit rotation). The bit that is rotated from the highest position to the lowest will also be sent to the carry flag.



ADDRESS METHODS: (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l
(only when rotating a word in the memory)

DATA LENGTH: Byte, word, longword

When rotating in the memory, you can only use word.

FLAGS: X - U

N - S

Z - S

C - Equal to the bit that was last moved from the operand. If number of steps is zero, the flag is cleared.

$$V = 0$$

SYNTAX: ROL #<steps>,Dy

ROL Dx,Dy
ROL <ea>

EXAMPLE CODE:

```
ROL.B  #1,D0  if D0.B contained 11110000, it would now be 11100001
```

ROR Instruction

The instruction ROR rotates a data register or a memory operand to the right. There are three ways to do this:

1. Rotate a data register to the right. Number of steps is a constant.
2. Rotate a data register to the right. Number of steps is stored in another data register. You can rotate 1-8 bits this way.
3. Rotate a word in the memory one bit to the right.

The rotation is done without an extra bit (i.e. a 8, 16 or 32 bit rotation). The bit that is rotated from the lowest position to the highest will also be send to the carry flag.

-- -- > OPERAND -- -- > C

ADDRESS METHODS: (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l
(only when rotating a word in the memory)

DATA LENGTH: Byte, word, longword

When rotating in the memory, you can only use word.

FLAGS: X - U

N - S

Z - S

C - Equal to the bit that was last moved from the operand. If number of steps is zero, the flag is cleared.

V - 0

ROL #<steps>,Dy

SYNTAX: ROL Dx,Dy

ROL <ea>

EXAMPLE CODE:

```
ROL.B  #1,D0  if D0.B contained 00001111, it would now be 10000111
```

ROXL Instruction

The instruction ROXL (ROtate Left with eXtend) rotates a data register or a memory operand to the left. There are three ways to do this:

1. Rotate a data register to the left. Number of steps is a constant.
2. Rotate a data register to the left. Number of steps is stored in another data register. You can rotate 1-8 bits this way.
3. Rotate a word in the memory one bit to the left.

The rotation is done with an extra bit (i.e. a 9, 17 or 33 bit rotation). The most significant bit is rotated to the carry flag and to the extra flag. The bit at the extra flag will be rotated to the least significant bit in the operand.

-- -- > C -- -- >
OPERAND < -- -- X < -- --

ADDRESS METHODS: (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l
(only when rotating a word in the memory)

DATA LENGTH: Byte, word, longword

When rotating in the memory, you can only use word.

FLAGS: X - The last bit that was rotated from the operand. Unaffected if rotation step was zero.

N - S

Z - S

C - Same as X

V - 0

SYNTAX: ROXL #<steps>,Dy
ROXL Dx,Dy
ROXL <ea>

EXAMPLE CODE:

```
ROXL.B  #1,D0  if D0.B contained 11110000, X = 1 it would now be 11100001  
if X = 0 then 11100000
```


ROXR Instruction

The instruction ROXR (ROtate Left with eXtend) rotates a data register or a memory operand to the right. There are three ways to do this:

1. Rotate a data register to the right. Number of steps is a constant.
2. Rotate a data register to the right. Number of steps is stored in another data register. You can rotate 1-8 bits this way.
3. Rotate a word in the memory one bit to the right.

The rotation is done with an extra bit (i.e. a 9, 17 or 33 bit rotation). The least significant bit is rotated to the carry flag and to the extra flag. The bit at the extra flag will be rotated to the most significant bit in the operand.

-- -- >	OPERAND	-- -- >	C
< -- --		X	< -- --

ADDRESS METHODS: (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l
(only when rotating a word in the memory)

DATA LENGTH: Byte, word, longword

When rotating in the memory, you can only use word.

FLAGS: X - The last bit that was rotated from the operand. Unaffected if rotation step was zero.

N - S

Z - S

C - Same as X

V - 0

SYNTAX: ROXL #<steps>,Dy
ROXL Dx,Dy
ROXL <ea>

EXAMPLE CODE:

```
ROXR.B #1,D0 if D0.B contained 00001111, X = 1 it would now be 10000111  
If X = 0 then 00000111
```

BTST Instruction

This instruction will test a bit in an operand specified by an effective address. The Z-flag (the only flag affected) will be set as the bit was before the change! The bit number is stored in a data register or is an immediate address.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Byte, longword

FLAGS: Z - Set if the bit was 0 before the change, else cleared. All other flags unaffected.

SYNTAX: BTST Dn,<ea>
 BTST #<data>,<ea>

BSET Instruction

This instruction will set a bit in an operand specified by an effective address. The Z-flag (the only flag affected) will be set as the bit was before the change! The bit number is stored in a data register or is an immediate address.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, longword

FLAGS: Z - Set if the bit was 0 before the change, else cleared. All other flags unaffected.

SYNTAX: BSET Dn,<ea>
 BSET #<data>,<ea>

BCLR Instruction

This instruction will clear a bit in an operand specified by an effective address. The Z-flag (the only flag affected) will be set as the bit was before the change! The bit number is stored in a data register or is an immediate address.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, longword

FLAGS: Z - Set if the bit was 0 before the operation, else cleared. All other flags unaffected.

SYNTAX: BCLR Dn,<ea>
 BCLR #<data>,<ea>

BCHG Instruction

This instruction will change a bit in an operand specified by an effective address. The Z-flag (the only flag affected) will be set as the bit was before the change. The bit number is stored in a data register or is an immediate address.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte, longword

FLAGS: Z - Set if the bit was 0 before the change, else cleared. All other flags unaffected.

SYNTAX: BCHG Dn,<ea>
 BCHG #<data>,<ea>

BFCHG Instruction

This instruction will complement a bit field in an operand specified by an effective address. The condition codes are set according to the value in the field before the change.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared
otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFCHG <ea>{offset:width}

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

BFCHG \$2001{D0,5} complements the bits
from \$2001 bit D0 through \$2001 bit (D0 + 5).

BFCLR Instruction

This instruction will clear a bit field to 0 in an operand specified by an effective address. The condition codes are set according to the value in the field before the change.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared
otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFCLR <ea>{offset:width}

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

BFCLR \$2000{1,5} clears the bits from
\$2000 bit 1 through \$2000 bit 6

BFEXTS Instruction

This instruction will extract a bit field from the specified effective address, sign extend to 32 bits, and load the result into the destination data register. The condition codes are set according to the value in the field before the change.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared
otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFEXTS <ea>{offset:width},Dn

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If

the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

```
BFEXTS    $1000{D0,D1},D2    extracts the bits
from $1000 bit D0 through $1000 bit (D0 + D1) and
loads them into D2
```

BFEXTU Instruction

This instruction will extract a bit field from the specified effective address, zero extend to 32 bits, and load the result into the destination data register. The condition codes are set according to the value in the field before the change.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared
otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFEXTU <ea>{offset:width},Dn

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If

the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

```
BFEXTU    $1000{D0,D1},D2    extracts the bits
from $1000 bit D0 through $1000 bit (D0 + D1) and
loads them into D2
```

BFFFO Instruction

This instruction searches for the most significant bit position that contains a 1. The bit position (the original bit offset plus the offset of the first 1 bit) of that bit is placed in Dn. If no bit of the bit field is 1, the value placed in Dn is the field offset plus field width. The condition codes are set according to the value in the field.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFFFO <ea>{offset:width},Dn

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If

the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

```
BFFF0    $1000{2,10},D1    finds the first 1
bit in the bits from $1000 bit 2 through $1000 bit
12 and put in D1
```


BFINS Instruction

This instruction inserts a bit field from the low order bits of the specified data register to a bit field at the specified effective address location. The condition codes are set according to the inserted value.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared
otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFINS Dn,<ea>{offset:width}

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

```
BFINS    D1,$1000{2,10}    insert bits from D1
to the bits from $1000 bit 2 through $1000 bit 12.
```

BFSET Instruction

This instruction sets all bits of a bit field at the specified effective address location. The condition codes are set according to the value in the field before it is set.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared
otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFSET <ea>{offset:width}

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

```
BFSET    $1000{2,10}    set bits from $1000 bit  
2 through $1000 bit 12.
```

BFTST Instruction

This instruction sets the condition codes according to the value in the bit field.

ADDRESS METHODS: Dn, (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: Unsized

X - Not affected

N - Set if the most significant bit of the field is set.
Cleared otherwise.

FLAGS: Z - Set if all bits of the field are zero. Cleared
otherwise.

V - Always cleared.

C - Always cleared.

SYNTAX: BFTST <ea>{offset:width}

The field selection is specified by a field offset and field width. The field offset denotes the starting bit of the field.. The field width determines the number of bits to be included in the field. The offset and width fields may be specified with an immediate operand ('#' is optional) or a data register. If the offset is an immediate operand the value must be in the range 0-31. If the offset is a data register the value is in the range -2^{31} to $2^{31}-1$. If the width is an immediate operand the value is in the range 1-31 or 0 to specify a width of 32. If the width is a data register the value is taken modulo 32, with values 1-31 or 0 to specify a width of 32. Bit 0 is the most significant bit.

```
LEA    $1000,A2
MOVE   #4,D1
BFTST  (A2){D1,5}  set the condition codes with
bit field from $1000 bit 4 through bit 9.
```

ABCD Instruction

Adds two bytes in BCD-form (Binary Coded Decimal). The destination operand is replaced with the sum of the source and the destination operand.

ADDRESS METHODS: Dn, -(An)

Only two methods are allowed:

1. Add data register to another data register (Dn to Dn)
2. Add memory to memory. This is used when you add BCD numbers stored in many bytes. You must start at the highest address (the least significant byte in the BCD number) and go upwards. The X flag is set if the addition results in a carry, which is added to the next byte.

DATA LENGTH: Byte

FLAGS: X - Set if carry from the most significant BCD digit.
N - undef
Z - Cleared if the result is NOT zero. Unaffected else.
C - Same as X
V - undef

SYNTAX: ABCD Dx,Dy
 ABCD -(Ax),-(Ay)

EXAMPLE CODE:

ABCD.B D0,D1 Adds the 2 BCD numbers in D0 and D1 and stores the answer in D1

SBCD Instruction

The instruction SBCD (Subtract BCD with extend) subtracts two bytes in BCD-form. The difference (destination - source - X flag) is stored in the destination register.

ADDRESS METHODS: Dn, -(An)

There are two ways to use this instruction:

1. Subtract a data register from a data register (address method Dn). The lower byte in the source register is subtracted from the lower byte in the destination register, where the answer is stored.
2. Subtract memory from memory. This way, you can subtract big numbers stored in many bytes. Since you only can use -(An) you must start on the highest byte (the least significant digits in the BCD number) and work down. If there is a carry, the X bit is set, which will be subtracted from the next byte.

DATA LENGTH: Byte

FLAGS: X - Set if a loan was required when subtracting,
 else cleared
 N - undef
 Z - Cleared if the result was 0, else unaffected
 C - Same as X
 V - undef

The Z flag is cleared if the result is not 0. This way, you can see if the answer, after a series of subtractions, is zero or not. First, you have to set the Z flag (done by comparing a register with itself). Then do the subtraction, and if the Z flag is set, the BCD number is zero.

SYNTAX: SBCD Dx,Dy

SBCD -(Ax),-(Ay)

EXAMPLE CODE:

```
SBCD.B  D0,D1  Subtracts the 2 BCD numbers in D0 and D1 and stores the  
answer in D1
```

NBCD Instruction

The instruction NBCD (Negate BCD) negates a BCD-number. The method used for tens complement. The tens complement to 01 is 99 ($1+99=100$), to 26 is 74 ($26+74=100$) and so on. The X flag is added to the tens complement which is the loan from the previous BCD calculation (multi-precision). A normal series of BCD instructions starts with the X flag cleared and the Z flag set.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte

FLAGS: X - Set if a loan was required when subtracting,
 else cleared
 N - undef
 Z - Cleared if the result was 0, else unaffected
 C - Same as X
 V - undef

SYNTAX: NBCD <ea>

EXAMPLE CODE:

```
NBCD.B  D0  Negates the BCD number in D0
```

Bcc Instruction

This instruction will cause a branch in the program if certain flags are set. There are fifteen ways of checking the flags. Each of them has a symbol of two letters which will replace the "cc" in "Bcc".

BCC	Branch Carry Clear - Branch if the C-flag is 0.
BCS	Branch Carry Set - Branch if the C-flag is 1.
BEQ	Branch EQual - Branch if the Z-flag is 1.
BNE	Branch Not Equal - Branch if the Z-flag is 0.
BGE	Branch Greater or Equal - Branch if N and V are equal.
BGT	Branch Greater Than - Branch if N and V are equal and Z=0.
BHI	Branch HIgher than - Branch if both C and Z are 0.
BLE	Branch Less or Equal - Branch if Z=1 or if N and V are different.
BLS	Branch Lower or Same - Branch if C=1 or Z=1.
BLT	Branch Less Than - Branch if N and V are different.
BMI	Branch MInus - Branch if N=1.
BPL	Branch PLus - Branch if N=0.
BVC	Branch V Clear - Branch if V=0
BVS	Branch V Set - Branch if V=1.
BRA	BRanch Always

Some conditions are pretty similar. BGE, BGT, BLE, BLT should be used when using signed integers and BHI and BLS when using unsigned integers.

ADDRESS METHODS: No special. You specify a label, which the compiler will change to a relative address, byte or word depending on how far you will jump.

DATA LENGTH: Short, Long

FLAGS: Unaffected

SYNTAX: Bcc.S <label>
 Bcc.L <label>

You do not have to add .S or .L, the compiler will choose the best syntax.

EASy68K will accept .B or .S to force 1-byte offsets and .W or .L to force 2-byte offsets.

DBcc Instruction

DBcc is an instruction that quits loops. The instruction is very similar to Bcc (same conditions are used, see above for the different conditions) except that the first operand is a data register that will be decreased with one until it reaches -1, then the loop stops. The loop can also quit if the flags are set correctly (specified with the condition). You often use DBRA, which will quit the loop when the data register has reached -1. If you want the loop to be looped 10 times, you should set a data register to 9 (since it ends at -1, not 0).

ADDRESS METHODS: No real address method.

DATA LENGTH: Word

FLAGS: Unaffected

SYNTAX: DBcc Dn,<label>

EXAMPLE CODE:

```
DBRA    D0,branch    subtracts 1 from D0, will skip line if D0 = -1
```

Scc Instruction

This instruction sets all bits in a byte (effective address) if a condition is true, else all bits are cleared.

The conditions can be

CC,CS,EQ,GE,GT,HI,LE,LS,LT,MI,NE,PL,VC,VS (for these, check the Bcc instruction for what they mean), SF (always false), ST (always true).

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x.l

DATA LENGTH: Byte

FLAGS: Unaffected

SYNTAX: Scc <ea>

EXAMPLE CODE:

```
S** D0 *if ** is true, then all bits in D0.B are set to 1, else to 0
```

BSR Instruction

Branch to SubRoutine. This instruction will push the address to the next instruction on the stack, then branch to the label specified in the instruction (a relative address). Used to call subroutines in your own programs.

ADDRESS METHODS: No real address method. The label is a relative address.

DATA LENGTH: Short (1 byte offset), Long (2 byte offset)

FLAGS: Unaffected

SYNTAX: BSR <label>
 BSR.S <label>
 BSR.L <label>

EXAMPLE CODE:

```
BSR   subroutine
```

EASy68K will accept .B or .S to force 1-byte offsets and .W or .L to force 2-byte offsets.

JSR Instruction

JSR (Jump to SubRoutine) works as JMP except that before the jump is made, the address to the instruction after JSR is pushed to the stack, so you can return with the instruction RTS.

ADDRESS METHODS: (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: JSR <ea>

EXAMPLE CODE:

JSR subroutine jumps to the subroutine, use RTS to return to the next instruction

RTS Instruction

The instruction RTS (ReTurn from Subroutine) does the opposite to the instructions BSR (Branch to SubRoutine) and JSR (Jump to SubRoutine). The longword on top of the stack is stored in the program counter. The instruction is used when ending a subroutine. The execution will return to the instruction that follows the last JSR- or BSR- instruction.

ADDRESS METHODS: None

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: RTS

EXAMPLE CODE:

```
RTS    returns to the address stored on the top of the stack
```

JMP Instruction

JMP (JuMP) is used to move the program control to an effective address. It really works as `MOVE.L xxx,PC`, since it changes the program counter to an effective address (calculated).

ADDRESS METHODS: (An), x(An), x(An,xr.s), x.w, x.l, x(PC), x(PC,xr.s)

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: `JMP <ea>`

EXAMPLE CODE:

```
JMP    address
```

RTR Instruction

The instruction RTR (ReTurn and Restore) pops the flags and the program counter from the stack. First a word is popped from the stack and the lower byte of that word is stored in the flag register. The higher byte is ignored. Then a longword is popped into the program counter. The stack pointer will be increased with six.

ADDRESS METHODS: None

DATA LENGTH: N/A

FLAGS: The flags is set according to the first word that is popped from the stack.

SYNTAX: RTR

MOVE USP Instruction

The instruction MOVE USP transfer the user mode stack pointer to or from an address register. This instruction requires that you are in supervisor mode. Since the 68k processor has two stack pointers, this instruction is necessary when a supervisor program wants to access the user mode stack pointer.

ADDRESS METHODS: An

DATA LENGTH: Longword

FLAGS: Unaffected

SYNTAX: MOVE USP,An
MOVE An,USP

RESET Instruction

This instruction restores all external units.

ADDRESS METHODS: None

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: RESET

RTE Instruction

The instruction RTE (ReTurn from Exception) is used to put data in both the status register and the program counter with the same instruction. This instruction is necessary when an operating system that is working in supervisor mode must leave the control to an application in user mode. The new contents in the status register and the program counter is popped from the stack. First, a word is popped into the status register, then a longword is popped into the program counter. Thus, the stack pointer is increased by six. This instruction requires that the S bit is set in the status register when the instruction is executed. Since the status register is changed after the instruction, it's possible that the processor will be in user mode then.

ADDRESS METHODS: None

DATA LENGTH: -

FLAGS: All flags are popped from the stack.

SYNTAX: RTE

STOP Instruction

This instruction enables interrupts and waits for an interrupt. This instruction requires that the processor works in supervisor mode. An immediate 16 bit data value is stored in the status register. Bit 13 in that data must be set (that will be the S bit after the instruction). Otherwise, an interrupt will occur since you're in user mode (you must be in supervisor mode before and after this instruction).

ADDRESS METHODS: N/A

DATA LENGTH: N/A

FLAGS: The flags will be bit 0-5 in the immediate data

SYNTAX: STOP #<data>

CHK Instruction

CHeck register against bounds. Is often used in high level languages to check if variables are in range. The lowest 16 bits of a data register are compared with an effective address. If the result is less than 0 (if bit 15 is 1) or greater than the limit, the result will be a CHK-interrupt.

ADDRESS METHODS: Dn, (An), (An)+, -(An), x(An), x(An,xr.s), x.w, x(PC), x(PC,xr.s), #x

DATA LENGTH: Word

FLAGS: X - U

N - Set if the data register is less than zero, cleared if the data register is greater than the higher limit. Else undefined.

Z - U

C - U

V - U

SYNTAX: CHK <ea>,Dn

TRAPV Instruction

The instruction TRAPV examines if an overflow has arisen. If it hasn't (the V flag is 0), nothing happens. If it has, the program counter and the status register is pushed on the stack, and the program counter will be given a new address, stored at the absolute address \$1C. The MPU is set to supervisor mode. The instruction is often used by high-level languages.

ADDRESS METHODS: N/A

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: TRAPV

TRAP Instruction

The instruction TRAP pushes the program counter and the status register on the supervisor stack, switches to supervisor mode and the program counter is given a new value taken from one of the sixteen vectors, given by a four bit data value.

The instruction is used in applications to call a supervisor program (an OS for example) without knowing exactly where in the memory the OS is.

ADDRESS METHODS: N/A

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: TRAP #<vector>

The vector used by TRAP is stored at $\$80 + 4 * \text{vector}$ (absolute address).

EXAMPLE CODE:

```
TRAP #15      EASy68K uses TRAP #15 for simulator control.
```

ILLEGAL Instruction

It will cause an interrupt (Illegal instruction) on all 68k CPUs. Often used as a breakpoint in debuggers.

ADDRESS METHODS: None

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: ILLEGAL

NOP Instruction

The instruction NOP (No OPeration) makes nothing during one instruction. Nothing happens except that the program counter is set to the next instruction. NOP is used to make small delays (VERY small delays, to let hardware parts to act) and to create empty space in the program which can later be used for changes.

ADDRESS METHODS: None

DATA LENGTH: N/A

FLAGS: Unaffected

SYNTAX: NOP

EXAMPLE CODE:

LOOP	NOP		this instruction does nothing
	BRA	LOOP	let's stick with what we're good at

"And at warp 9 we're going nowhere mighty fast!" Scotty

Psalm 23

The Lord is my shepherd; I shall not want. He maketh me to lie down in green pastures: he leadeth me beside the still waters. He restoreth my soul: he leadeth me in the paths of righteousness for his name's sake. Yea, though I walk through the valley of the shadow of death, I will fear no evil: for thou art with me; thy rod and thy staff they comfort me. Thou preparest a table before me in the presence of mine enemies: thou anointest my head with oil; my cup runneth over. Surely goodness and mercy shall follow me all the days of my life: and I will dwell in the house of the Lord for ever.