M68000 Assembler

SSASM Users Manual

Version 1.032
August, 1993

Applix 1616 microcomputer project
Applix pty ltd

SSASM Users Manual

M68000 Assembler

Applix 1616 project

Even though Applix has tested the software and reviewed the documentation, Applix makes no warranty or representation, either express or implied, with respect to software, its quality, performance, merchantability, or fitness for a particular purpose. As a result this software is sold "as is," and you the purchaser are assuming the entire risk as to its quality and performance.

In no event will Applix be liable for direct, indirect, special, incidental, or consequential damages resulting from any defect in the software or its documentation.

The original version of this manual was written by Andrew Morton, based on David Farb's 68000 assembler.
Additional introductory and tutorial material by Eric Lindsay
Editorial and design consultant: Jean Hollis Weber

Comments about this manual or the software it describes should be sent to:

Applix Pty Limited
Lot 1, Kent Street,
Yerrinbool, 2575
N.S.W. Australia
(048) 839 372

Private BBS systems (ringback) on (02) 554 3114 and (02) 540 3595

ISBN 0 947341 ?? ?

1
Introduction

SSASM is an assembler for the Motorola 68000 series of 16/32 bit microprocessors. It accepts a file containing standard Motorola assembler mnemonics and directly produces executable transient programs. That is, it produces .exec files for the Applix 1616.

SSASM executes as either a built-in command, or as a normal transient program, under the 1616/OS operating system on an Applix 1616 computer system. Versions of 1616/OS later than 3.2b have SSASM, version 2.2, built into the operating system eproms. This is often very convenient, because you can edit and assemble programs without using a disk drive, if you wish.

SSASM supports all the instructions of the 68000 microprocessor, but does not support (at this time) the special instructions for the 68010 and 68020. These opcodes may be implemented as macro instructions. In addition, SSASM supports a variety of listing and pagination controls of use when examining the list file, data declaration statements for initialising constants, file inclusion and conditional assembly controls.

Two of the more interesting features of SSASM are

- structured programming constructs (which don't quite work)
- macro processing capability (which do work)

The structured programming constructs should include

- IF/THEN/ELSE
- FOR/ENDF
- WHILE/DO/ENDW
- REPEAT/UNTIL

all of which would be very pleasant, if they worked correctly under all circumstances. However you can easily get along without them, so don't pay much attention to the chapter on how they should work.

Macro processing permits conditional assembly inside macros, parameters to macros, automatic generation of unique labels and other features, all detailed in the chapter on macros. Very handy stuff, and essential once you get more familiar with the assembler.

1.1 References

The principle reference for the instructions on the 68000 is published by Prentice-Hall, Inc. of Englewood Cliffs NJ, 07632, titled *M68000 16/32 Bit Microprocessor Programmer's Reference Manual*, written by the staff of Motorola. Prentice-Hall, about 200 pages. This lists all the 68000 instructions, and is an essential reference tool, but will not tell you how to use the instructions, nor how to program in assembler. The current version is the fifth edition, and we recommend not bothering with the first, second or third editions.

The SSASM manual does not duplicate the content of the *Reference Manual*, but should be used in conjunction with it. The *Reference Manual* is available from Applix, and is recommended.

Other references which you may find helpful are the following:

The 68000: Principles and Programming, by Leo J. Scanlon, published by Howard W. Sams, 238 pages. The 1981 edition is now somewhat dated, but still includes much excellent introductory material (and I picked it up cheap). Like most other assembler manuals, it assumes you are familiar with the concept of assembler language, and merely want to add yet another version to your repertoire. This is historically naive, since everyone has to start somewhere.

68000 Microprocessor Handbook, by Gerry Kane, published by Wayne Green Books.

68000 Assembler Language Programming, by Gerry Kane, published by Wayne Green Books. I am not certain that Wayne Green Books are still available in Australia. A possible alternative, by the same author, is below.

68000 Assembly Language Programming by Gerry Kane, Doug Hawkins and Lance Leventhal, published by Osborne McGraw-Hill, about 300 pages covering concepts, simple problems at about the level of the Scanlon book, including over 50 program examples. Advanced topics cover parameter passing, subroutines, I/O and interrupts, and there is a useful major coverage of problem definition, program design, documentation, debugging and testing and so on. There are also about 150 pages of appendix on the instruction set, a summary, and object code formats.

680x0 Programming by Example by Stan Kelly-Bootle, published by Howard W. Sams, 482 pages. This one is a good introduction for those who have done minor work in another assembler. It contains brief, but very clear, summaries of 680x0 differences, addressing modes, and instructions, and then goes on to a number of commented code examples. The major example is a 68000 version of the Kermit file transfer protocol, which is a fairly involved piece of work. The beginner would be better served by the same author's *68000, 68010, 68020 Primer* (but since I don't have a copy, I can't review it).

68000 Assembly Language Techniques for Building Programs, by Donald Krantz and James Stanley, published by Addison-Wesley, 400 pages. After a brief overview of the 68000 instruction set (which contains some interesting viewpoints), this gets down to the main contents, namely implementing a text editor in 68000 assembler. Lots of commented code, with explanations of why things were done certain ways. While this is fine for the more experienced assembler language programmer, the beginner may have problems coming to grips with such a large project.

1.2 File Naming Conventions

`SSASM` uses the following file naming conventions

filename.s	source file (must include `.s`).
filename.exec	executable code.
filename.lst	listing output.
filename.xrel	relocatable code.

Although the source file must have a `.s` extension, you should not type the `.s` when invoking the assembler, or the `makexrel.shell` program for making `.xrel` files.

`SSASM` will automatically supply the `.exec` extension when it produces the final executable file, by removing the `.s` extension, and substituting the `.exec` extension. If you use the `-o` command line switch, you will override this automatic `.exec` naming feature.

To produce standard Applix 1616 relocatable programs (the `.xrel` files), you actually run the assembler twice, by using the `makexrel.shell` shell program that is explained later. This step is usually only done when you have a complete working program, with all the bugs and problems removed.

The listing file produced by the `-l` switch is an ASCII text file produced by the assembler. It includes the text of your `.s` source file, plus the expansion of any macros or include files, so you can see what was actually in them. It also contains a hexadecimal representation of the actual Motorola 68000 object code produced, and the location in memory they occupy. This lets you see the actual code produced by the assembler. The listing switches or options such as `-f -h -t` let you alter the page length and appearance of the listing, to more easily suit it to your printer or output device.

1.3 Invoking the Assembler

`SSASM` is a command executed under 1616/OS. You invoke it by typing the name of the command, followed by any switches (or parameters). For example

```
SSASM testcode
```

will assemble a file called `testcode.s` from the current directory.

The file `testcode.s` is a simple ASCII text file, usually produced using the `edit` inbuilt command. It contains lines of text, each usually a single Motorola 68000 command. The details of this are discussed in section 2.

Switches are specified by a leading dash **-**, immediately followed by a letter (upper or lower case), sometimes followed by a file name or other parameters. There must be at least one blank between the switch and the file name, but no blanks between the switch and parameters (see below).

The general form to invoke `SSASM` (where [ ] enclose optional switches) is

```
ssasm sourcefile [-flnprv] [-h pagelen] [-o objfile] [-t tabs]
```

The complete list of switches you may specify is

`-f`	use form feed between pages of the listing output.
`-h[nn]`	set listing page length to *nn* lines in total.
`-l`	produces listing output, to standard output. The `-h -f` and `-t` options vary the format of the listing file.
`-n`	no output of an executable file. Assemble only. Use to test your syntax, and so on.
`-o` [filename]	specifies the object filename for output. The default output file's name now ends with `.exec`. The `-o` option overrides this. If no `-o` option is specified, the object file's name is that of the source file, with the `.s` extension stripped off and `.exec` added.
`-p`	
`-r`	not working yet. Should produce an output ready for linking to a relocatable form directly, without needing to use `makexrel.shell`.
`-t[nn]`	set tab stops every *nn* columns
`-v`	verbose mode, prints sign on message, version number and status information as assembly proceeds.

Switches may be specified in any order, but file names must immediately follow their respective switches. Switches and file names may be entered in any combination of upper and lower case. All file names are translated to upper case for use by 1616/OS.

A sample command such as

```
ssasm testcode -o objout -l
```

reads the source code from `testcode.s`, writes the object code to `objout` (note there is no `.exec` extension) and displays a listing file on the screen. The listing file may be redirected to a file or a character device by using the 1616/OS > redirection command. The listing file extension should be `.lst`,

If the **`-o`** switch is not specified, like this

```
ssasm testcode.s -l > sa:
```

`SSASM` reads the source code from `testcode.s`, writes the object code to `testcode.exec` and sends the listing file to serial port A.

If the source program that you are assembling contains more symbols than will fit in the memory of your computer, `SSASM` will automatically overflow the symbol table to disk. Provided there is sufficient space on your disk drives, `SSASM` will assemble any size source module.

The **`-f`** switch causes `SSASM` to use a form feed between pages in the listing file, while the **`-h`** switch sets the number of lines per page. The default number of lines is 66. To use form feeds and set the number of lines to 80, the command might be

```
ssasm testcode -f -h80
```

Normally `SSASM` simply copies horizontal tab characters in the source file to the listing file. The **`-t`** switch allows you to set tab stops at intervals in the listing file. `SSASM` will fill in with blanks to the next tab stop whenever a tab character is found in the input file. The command

```
ssasm testcode -t10
```

will set a tab stop every 10 columns in the listing.

1.4 Makexrel

Once a source file is producing working `.exec` files that produce the results you want, you should produce a final version of your executable code. The 1616 normally uses relocatable code. This means the code can be placed by 1616/OS anywhere in memory that contains sufficient space. This is an obvious advantage if you wish to run more than one program at once (with the `.exec` files, only one can fit in any specific set of memory locations).

The `makexrel.shell` is an easy way to produce an `.xrel` version, since it simply assembles your existing source code twice, at different locations in memory. It then locates the differences, and uses these to produce the desired `.xrel` version. Finally, it cleans up various temporary files after itself. Invoke it simply as

```
makexrel testfile
```

The `.makexrel.shell` is fairly simple, as you will see if you `type` it to your display. However it does depend upon another program, `genreloc.xrel`, being available in either the same directory, or in your normal xpath.

If you have the multitasking 1616/OS Version 4, you may find you have insufficient memory allocated to stack space to run `genreloc.xrel` correctly, the first time you attempt this. You correct this by using the `chmem.xrel` program supplied on your Version 4 user disk. Thus `chmem 16 genreloc.xrel` makes the default stack space for `genreloc.xrel` 16 kbytes. Don't forget to make sure `chmem` can find `genreloc.xrel` by making sure they are in the same directory, or in the xpath. This operation should only need to be done once.

1.5 Setting up your disk

Correct operation of the assembler depends upon it being able to find all the files it requires. When you are doing simple programs, this can be very easy. For example, if you put on the root directory of your disk `makexrel.shell`, `genreloc.xrel`, `chmem.xrel`, and any macro

files you have, such as `syscalls.mac` and `traps.mac`, then you can also put your source files in the same directory. If a source file has any `include` files, you merely have to state their name, and they will be found in the same directory as everything else. However, there are serious problems with this, in the long run.

The major problem is keeping track of where things should be, once you have written a number of programs. Most people use sub directories to group files of similar nature. If you have to assemble everything in the root (or any other specific directory), you also have to shuffle files in and out of it, as you fill it up. It is better to make some consistent location for the files needed by the assembler, and use the `xpath` and `assign` commands to keep everything working.

For example, make two directories, one called `bin`, and the other `include`. In your `autoexec.shell` startup file, have the commands `xpath /rd/bin /f0/bin` and `assign /include /f0/include` (and if you have a second disk drive, or a hard disk, add these also). Place all the executable programs you will normally need in `/bin`. This means `makexrel.shell`, `genreloc.xrel`, `chmem.xrel`, and so on. Place all your normal include source files, such as `syscalls.mac` and `traps.mac` in the `/include` directory. When writing your assembler programs, and using the `include` directive, you can type it as `include /include/traps.mac`, for example. If you do this consistently, it becomes very easy to change where the include source files actually reside, just by changing the `assign`, without changing any of your source files.

1.6 Error messages

If `SSASM` finds any errors (heaven forbid!) in your program, or fails to understand your intent (more likely) it produces a message on the console giving the line number of the source statement in the input file, and a reasonably detailed error message.

In addition, `SSASM` inserts a line in the listing file after the source statement giving the error message again. No more than two errors are ever reported on a single line.

`SSASM` returns an error code to 1616/OS if an assembly error occurs, so that shell files may terminate (if the `trap` command has been given).

Some errors are not associated with a source line number, such as a missing source file name, or insufficient space on the disk for another file. These messages appear only on the video display.

Appendix A contains a complete listing of all the error messages (and the error numbers produced by earlier versions of `SSASM`) and a description of their likely cause.

2
Coding Conventions

This chapter provides the basic information on coding and formatting SSASM source files. You may use the inbuilt 1616/OS editor, edit, to create these files. Source files are standard ASCII character files, with no special command codes.

Source statements may be coded in upper or lower case, but SSASM will convert all input (except literals and comments) to upper case before processing. This means that "label0" and "LabeL0" will be treated as the same symbol by SSASM. Any character entered by an editor is acceptable in comments and literals, but only alphabetic (A thru Z and a thru z), numeric (0 thru 9), and the special characters

+ - < > , . / ( ) ' á% $ # and !

may be used in the remainder of your program. See the sections below for more detailed discussions.

Source files are divided into lines, terminated by a carriage return-line feed combination. Each line is either a

- comment
- machine instruction
- assembler directive
- structured assembler operation
- conditional assembler operation
- macro definition.

2.1 Comments

Comment lines are ignored by SSASM, and may be freely used to document your program. Comment lines have an asterisk "*" in column 1, and any text you desire in the remainder of the line. The lines

```
*
* Main entry point.
*
```

are all comments and will be ignored by SSASM.

Comments can also be provided by using the semicolon ";" to indicate the start of the comment.

2.2 Instruction Formats

Most instructions can be divided into four fields.

label operation operands comments

The assembler determines which is which by their position within each line of the source file. Anything that starts in the first column of a line is a label. The space or tab character is used to indicate where the label field ends. The next item is the operation, and again the space or tab character indicates where this field ends, and so on for operands and comments. You can have as many space or tab characters as you like. Multiple spaces are treated the same as a single space for the purposes of indicating where a field ends. However, you must never put a space or tab where there should not be one, as you will totally confuse the assembler.

The label provides a name for the instruction and is usually optional. If there is no label, a space or tab is placed at the start of the line, and the next item in the line is treated as an operation.

The operation field determines the machine, macro, conditional assembly, or other operation to be performed. The operation field must be followed by one or more blanks or tab characters.

Operands are usually required (depending on the operation) and provide the operation with data. No blanks are permitted in the operand fields, except in structured operations.

The comment field is always optional and must be separated from the operands by at least one blank or a tab character. Note that the comment field does not need to be preceded by an asterisk or semicolon, however the source file will probably be more understandable if one is used.

The end of the line is indicated by a carriage return-line feed combination.

SSASM permits null lines (lines having no label operation or other fields) or lines which have only a label and none of the other fields. However, if there is a label, and you do not wish to have an operation on that line, the comments portion must also be omitted, since the assembler will attempt to interpret the comments as an operation.

Blanks or horizontal tab characters are used to separate the label, operation, operands and comment fields. If the label is terminated by a colon ":", no blank is required after the colon.

Labels of the form

```
START
            MOVE.L          D0-D7/A0-A7,-(SP)
```

are permitted, in fact encouraged. Having the label on a line separate from all other information is quite handy should you have to insert a new instruction between the label and the instruction.

In the instruction

```
LABELONE    MOVE.L D0,D3   SET RESULT
```

LABELONE is the label field, MOVE.L is the operation, D0,D3 are the operands, and SET RESULT is a comment.

2.3 Labels

Labels are at least one character long, must begin with an alphabetic character (A thru Z) or a period "." and may contain alphabetic and numeric characters and periods ".". Only the first 8 characters are significant. Labels may begin in column 1 and/or may be followed by a colon ":". If the label is followed by a colon, it need not begin in column 1 of the line. The colon is not part of the label.

The colon allows structured programming with indented lines of code and labels. Allowing the label to be indented requires that the label have some special character which denotes a label; otherwise it would be confused with the operation field.

START and LOOPAGN are labels

```
START       SUBQ.L          #3,D3
            LOOPAGN: CHK    (A2)+,D3
```

2.4 Operation Field

The operation field gives the name of the assembler operation which is being performed. It is always composed of alphabetic characters, but only the first five are used. The basic operation code is followed (usually) by a period and an operation qualifier.

In the instruction

```
            BTST.B          D6,OFFSET(A3)
```

`BTST` is the operation code, and `B` is the qualifier. Normal qualifiers are

B - byte
W - word (2 bytes)
L - long (2 words, 4 bytes)

and

S - short (for branching).
L - Long (for branching).

but others may be used.

Qualifiers usually specify the length of the data that an operation will use; for branch instructions they determine the length of the displacement (one byte or two).

2.5 Operands

Operands are required for most operations in `SSASM`. Quite frequently there are two operands, which are separated by a comma. No blanks are permitted in the operand portion of an instruction, except the structured programming operations.

For 68000 machine operations there are many formats of operands, and these are covered in Chapter 3. Each assembler directive also has a particular format for operands which is covered in Chapter 4.

2.6 Comments

Comments may be placed after the operand field of a machine, data definition, conditional assembly, assembler directive, structured operation, or macro invocation. Comments of this form must be preceded by at least one blank or tab. The instruction

```
ADD.L          D0,D0          Multiply value by 2
```

has the comment `Multiply value by 2` after it.

It is suggested that you put an asterisk "*" or semicolon ";" in front of comments such as these for both readability and compatability with other assemblers.

2.7 Expressions

Expressions allow you to compute arithmetic results in your source code during assembly. `SSASM` supports a comprehensive set of operations in expressions.

The operations are

- addition **+**
- subtraction **-**
- multiplication *****
- division / (truncating)
- shift left <<
- shift right >>
- logical and **&**
- logical or **!**
- parenthesis **(** and **)** for order of evaluation.

Addition and subtraction are performed as

[expression1] + [expression2]
[expression1] - [expression2]

Multiplication and division are performed as

[multiplier] ***** [multiplicand]
[dividend] **/** [divisor]

The divisor must not be zero. The result of the division is a truncated integer, thus 1/2 is 0.

Shifting is performed by

[expression] << [shift count]
[expression] >> [shift count]

In the first line, the expression is shifted LEFT (<<) the number of bits specified by the shift count (an expression). In the second line the expression is shifted RIGHT (>>) the number of bits specified in the shift count.

The logical operations of AND and OR (inclusive) are performed by

[expression 1] **&** [expression 2]
[expression 1] **!** [expression 2]

In the first line, the bits of expression 1 are logically ANDed with the bits of expresssion 2. In the second line, the bits of expression 1 are logically inclusive ORed with the bits of expression 2.

The order of evaluation is logical operations first, then shifts, then multiplication and division, then addition and subtraction. Parenthesis can be used to modify the order of evaluation of operations.

A*(B+C)

means add the value of B to the value of C, and multiply the result by the value of A, while

A*B+C

means multiply the value of A times the value of B and add the value of C.

In addition to labels, constants of several forms can be used. A constant is either a numeric or a literal (a series of characters) which defines a unique value.

Numeric constants can be entered in four forms, decimal, binary, octal and hexadecimal. The formats are given by

@01234567 - Octal, preceded by "@", followed by one or more octal digits (0-7).

%01 - Binary, preceded by "%", followed by one or more binary digits (0 and 1).

$0123456789ABCDEF - Hexadecimal, preceded by "$", followed by one or more hexadecimal digits (0-9, A-F).

0123456789 - Decimal, a string of decimal digits.

Literals are strings of characters enclosed in apostrophes (’), or double quotes ("), which are used as operands in instructions.

’A-Z, etc,’ - ASCII string, one or more characters, enclosed in apostrophes ’. Use two apostrophes in a row to represent one, ’’ represents a single apostrophe.

for example

```
        CMPI        #’A’,D0      Check for ’A’.
```

2.8 Location Counter

The current location counter in the assembly can be used in expressions by using the special character "*****". "*****" represents an absolute value which is the value of the location counter.

```
            BRA.S       *           Loop forever.
```

or

```
LENGTH      EQU         *-START     Compute length of data.
```

2.9 Special Names

SSASM reserves certain names for internal 68000 registers and other uses. The most common are the data and address registers, named D0 thru D7, and A0 thru A7. A7 may also be referred to as SP for stack pointer.

Other special names, used only in certain instructions, are USP for the User Stack Pointer, CCR for the Condition Code Register, SR for the Status Register, and PC for the Program Counter.

You should not define labels with these names since they are reserved by SSASM for these CPU registers. In addition, avoid the names VBR for Vector Base Register, SFC for Source Function Code, and DFC for Destination Function Code. These names will be used for the 68010 and 68020.

The name NARG is reserved for macro processing. It defines the number of arguments to the current macro expansion. See Chapter 6.

2.10 Code Generation

The 68000 has a wide variety of instructions that can accomplish the same operation, but some may be more efficient than others. Where possible SSASM will chose the shortest form of an instruction.

It is always possible to explicitly override the choices that SSASM makes by coding the proper form of the instruction. Usually this means putting the proper qualifier on the operation code.

SSASM will assemble

```
            BRA         BACK
```

in 2 bytes if the label BACK is previously defined, and can be reached with a displacement from the PC between 0 and -128. Otherwise it will assemble 4 bytes for the instruction.

```
            BRA.L       BACK
```

will always be assembled in 4 bytes, and

```
            BRA.S       BACK
```

will always be assembled in 2 bytes (which may cause an error if the displacement from the PC is less than -128 or greater than 127).

3
Operand Formats

This chapter explains the formats allowed on all machine instruction operands. The 68000 has a wide variety of operand addressing modes.

Operands may be located in a data or address register, or they may be indirectly refered to by an address register. The contents of an address register, used for indirect references, may be automatically incremented or decremented. Other modes are available. Each of these modes requires differing operand formats which are explained below.

3.1 Source and Destination Fields

The instruction format of the 68000 was designed for regularity, and most 68000 instructions have two operands. The first operand is always considered the source operand, and the second is always considered the destination operand. Data movement is always from left to right.

In an ADD instruction

```
ADD.L           #400,D5
```

the immediate value 400 (decimal) is added to the contents of data register 5 and the result stored back in data register 5.

The MOVE instruction

```
MOVE.L          FROMDATA(A3),TODATA(A6)
```

moves one long word from `FROMDATA` displaced from A3 to `TODATA` displaced from A6.

The CMP (compare) instruction

```
CMP.L           #400,D3
```

subtracts the immediate value 400 from the contents of D3 (just as the SUB instruction), sets the condition codes, but does not store the result in D3. D3 is left unchanged.

CMP instructions are usually followed immediately by Bcc instructions, of which there are about 15 possible condition codes.

```
CMP.L           #32,D4          COMPUTE D4 - 32
BLT.S           D4LT32          BRANCH IF D4 < 32
```

Unfortunately this is backwards from the way people usually think, and it may take some getting used to.

3.2 Data Register Direct

The simplest operand format is the data register, which is specified by one of the names D0 thru D7. Some examples of its use are

```
ADDQ.L          #3,D0
CLR.L           D5
```

In the ADDQ (add quick) instruction, D0 refers to data register 0, D5 in the CLR (clear) instruction refers to data register 5.

References to data registers are of the form Dn, where n is a digit between 0 and 7.

3.3 Address Register Direct

Address registers, when the data they contain is to be manipulated, are simply refered to by their names of A0 thru A7. A7 may also be referred to as SP (Stack Pointer). Remember that A7 is a double register, depending upon whether the 68000 is in User or Supervisor mode. SP may be referred to as USP or SSP.

```
LEA.L          FIELD1(A3),A4
MOVE.L         A3,USP
MOVEA.W        LABEL2,A6
```

The LEA instruction loads the effective address of `FIELD1(A3)` into address register 4 (A4). The MOVE instruction moves the contents of address register 3 (A3) to the user stack pointer (USP).

3.4 Address Register Indirect

To use the contents of an address register as a pointer to the data for the instruction, use the name of the register enclosed in parenthesis.

```
ADD.L          (A5),D5
MOVE.L         D3,(A0)
```

The ADD instruction finds the long word (.L) which is located at the address CONTAINED in address register 5 ((A5)) and adds it to the contents of data register 5 (D5). The MOVE instruction places the entire contents of data register 3 (D3) in the location CONTAINED in address register 0 ((A0)).

3.5 Address Register Indirect with Postincrement

This form of addressing uses the contents of an address register as a pointer to the data for the instruction. Once the data has been accessed, the contents of the address register are incremented by an appropriate length.

The length is determined by the qualifier on the operation: if the qualifier is byte (.B), the increment is 1, if the qualifier is word (.W) the increment is 2, and if the qualifier is long (.L) the increment is 4.

To specify this form of addressing, enclose the address register in parenthesis, and follow it immediately with a plus sign

```
CMP.B          (A3)+,D0
CLR.L          (A7)+
```

The CMP instruction will subtract the contents of the byte pointed to by A3 from D0, set the condition codes accordingly and increment A3 by 1.

The CLR instruction will increment the contents of A7 by four after setting the long word at that address to zero.

3.6 Address Register Indirect With Predecrement

This form of indirect addressing decrements the contents of the address register BEFORE it is used to access the data. It is specified by enclosing the register name in parenthesis, and immediately PRECEDING it with a dash

```
MOVEM.L        D0-D3/D6/D7,A0-A5,-(SP)
SBCD.B         -(A3),-(A5)
```

The MOVEM will store the contents of the indicated registers at the address pointed to by the current stack pointer (SP, the same as A7) -4, -8, -12, and so on.

Note again that the address register is decremented BEFORE it is used as an address.

The value of the decrement is determined by the length code of the instruction. L will decrement by 4, W by 2 and B by 1.

3.7 Address Register Indirect with Displacement

This form of addressing allows you to use an address register as a pointer to a group of data and reference data within the group using ABSOLUTE displacements. One method of coding this mode (not recommended) is

```
        SUBQ.L      #3,12(A4)
```

If you change the data structure that A4 is pointing to, you will have to change all the absolute numbers used to reference that data structure. You may miss a few. A better way uses labels to define offsets into the data area

```
DATA
        FIELD0      DS.L        2
        FIELD1      DS.L        4
```

and

```
        SUBQ.L      #3,FIELD1-DATA(A4)
```

This way, to change the data structure, simply change the label definitions and reassemble the program; `SSASM` will compute the proper value for FIELD1 and use it in the instruction.

For this mode of the instruction the displacement is coded as an expression which yields an ABSOLUTE result. Any valid expression can be used,

```
        TAS.B       FIELD22+16*4(A3)
```

It is usually better to assign a symbolic name to every data field than to use absolute numeric expressions because of problems associated with changing the data structure.

3.8 Address Register Indirect with Index

This mode adds the contents of an address register to an index register using either long or word arithmetic, and then adds a displacement (sign extended) to the result. It is specified using the following format

```
        ADD.W       DISPLACE(A3,D2),D0
        ADD.W       DISPLACE(A3,D2.W),D3
        ADD.W       DISPLACE(A3,D2.L),D3
```

In the first ADD instruction the contents of A3 are added to the contents of D2 (D2 is sign extended from a word to a long word), the value of DISPLACE (a single byte, sign extended) is then added to the result.

The second ADD instruction is identical to the first except the word extension of D2 is coded explicitly (D2.W).

The third ADD instruction specifies that all of register D2 is to be added to register A3, then the sign extended byte is added to the result.

Note that the displacement is a single byte, and must have a value between -128 and +127.

3.9 Absolute Short Addressing

This form of addressing uses a two byte, absolute address, to refer to the data. This permits you to access either the first 32K, or the last 32K of memory (address' $000000 thru $007FFF and $FF8000 thru $FFFFFF). This mode is obtained by coding an ABSOLUTE expression as the operand.

```
            JMP         $400
SUBONE      EQU         $1202
            JSR         SUBONE
DEVONE      EQU         $200
            BSET.B      #4,DEVONE
```

If the value of the expression is defined at the time the instruction is assembled (ie. it is a backward reference), SSASM will choose the short mode for the reference.

If the value of the reference is NOT defined at the time the instruction is assembled (ie. it is a forward or external reference), SSASM will choose the long mode (next section) for the reference.

However you can force all forward references to be short form by using the OPT FRS instruction (see Chapter 4).

3.10 Absolute Long

SSASM will use the PC relative form of addressing in some cases, if the expression is a backward reference.

The form for requesting the long form is the same as the short form above.

```
            MOVE.L      D0,DATASTOR
DATASTOR    DS.L        1
```

3.11 Program Counter with Displacement

This form of addressing uses the present location in the Program Counter register (which is usually pointing to the byte AFTER the first two bytes of the instruction) and adds a displacement to it. The displacement is usually the first two bytes after the instruction.

The high order bit of the displacement is treated as a two's complement sign for the displacement, so it is possible to reference forward and backward from the current instruction.

The form for requesting this mode of addressing is

```
            BCLR.B      #3,ABSFRWD(PC)
```

The value of the program counter is subtracted from the value of the expression, and the result is placed in the displacement.

3.12 Program Counter Relative with Index

This mode allows you to specify an index register and a displacement which will be used with the program counter to form an address. You may specify whether the index register is to be used as a word, or a long word. The following forms are possible.

```
            CHK.W       WORD(PC,D3.L),D4
            CHK.W       WORD(PC,A4.W),D0
            CHK.W       WORD(A4.W),D0
```

The last two instructions above are identical, if `WORD` is defined in the same relocatable section as the instruction. Note that the value of the displacement must be between -128 and +127 or an error will result.

3.13 Immediate Data

Immediate data always begins with a hash mark "#" which must be followed by an expression. The size of the result of the expression must fit in the size of the immediate data area or an error will result. This size usually depends on the instruction qualifier.

```
ADD.W          #SIZE1+SIZE2,D0
```

`SIZE1+SIZE2` must be an absolute value, and must be in the range -32768, +32767, since the operation qualifier is W for word.

3.14 Bcc, BRA, BSR and DBcc Instructions

These instructions all use addressing which is displaced from the program counter. However, the program counter is not explicitly coded. They must all refer to expressions which are locations in the source file.

The relative distance to the label, from the branch instruction, must be in the range - 32768 to +32767.

If the relative distance to the label is in the range -128 to +127 then the short form of the Bcc, BRA, and BSR instructions may be used. These are two byte instructions where the displacement is in the second byte.

The DBcc instruction has only the longer form (4 bytes). Some examples are

```
BHI.S          LABEL
BRA.L          LOOP
BSR.S          SUBONE
DBLT.L         COUNTLP
```

3.15 MOVEM Register Specification

The MOVEM instruction moves multiple registers between storage and the CPU registers. The format for the register specification allows you to specify individual registers, ranges of registers, or any combination of these.

```
MOVEM.L        D0-D3/D6/A0/A3/A6,-(SP)
```

moves registers D0 thru D3 (D0, D1, D2 and D3), D6, A0, A3 and A6 to the stack.

Ranges of registers are separated by a dash "-" and the range must specify the same type of register, either all data or all address. Ranges and individual registers are separated by slashes.

For another method of specifying registers, see the REG directive in Chapter 4.

4
Assembler Directives

This chapter discusses the various assembler directives which you may use in your programs. Assembler directives give instructions to the assembler (not the 68000 chip which will execute your program) on how to assemble your program.

These directives may be grouped into the following categories.

Miscellaneous	Listing Control
OPT INCLUDE FAIL END	PAGE LIST NOLIST SPC TTL
Section Control	
ORG	
Symbol Definition	Data Definition
EQU SET	DS DC

The following sections explain each operation in detail. They are organized alphabetically for easy reference.

4.1 DC: Define Data Constants

The DC directive allows you to define data areas in your program and initialize them to certain values. It may be used to define data in terms of bytes, words, or long words.

```
VALUEONE DC.L           1234
```

defines a long word (.L) containing the decimal number 1234 and assigns the symbol VALUEONE as the name of the long word.

Any number of constants can be specified, separated by commas.

```
NUMLIST     DC.W        1,3,4,6,23,16
```

defines a series of words containing the numbers 1, 3, 4, 6, 23, and 16.

The data items can be any absolute expression

```
BYTES       DC.B        A+16,32+'.',12*(NUML-BYTEB)
```

Note that the expression "32+'.'" above must be written with the number first. If it were written with the literal first, `SSASM` would assume this was the declaration of a character string. `SSASM` would expect a comma or end of statement after the string, not the "+32".

Long character strings may be defined by enclosing them in quotes,

```
MESSAGE1    DC.B        'Value of parameter not found.',0
```

defines 29 bytes containing the character string 'Value of parameter not found.' and adds a zero to the end (null-termination). Character strings passed to 1616/OS system calls require null-termination, as in the example above. Lower case letters are preserved as is. The symbol `MESSAGE1` is assigned the address of the first byte.

An unlimited number of bytes may be declared in a single DC.

4.2 DS: Define Storage

The DS directive defines a storage area and initializes it to zeroes. The length qualifier may be specified as B for bytes, W for words and L for long words. The operand field contains the number of data elements (bytes, words or long words) that are to be declared.

```
INBUFFER    DS.B        80          INPUT BUFFER STORAGE
WORD1       DS.W        1
STACKSP     DS.L        16          SPACE TO SAVE REGISTERS
```

`INBUFFER` is assigned the address of the first byte in an 80 byte area. `WORD1` is assigned the address of one word, and `STACKSP` is assigned the address of the first byte in a 16 long word area.

The operand field may contain any absolute expression whose value is known at the time the statement is assembled. (The expression may only use symbols defined prior to the statement.)

An unlimited number of bytes may be declared in a single DS.

4.3 END: Terminate the Source Program

The END directive must be the last line of your source program. The assembler will stop at this point and will not process any statements after this one.

```
            END
```

marks the end of your source program.

4.4 EQU: Equate a symbol to a value

The EQU directive will compute the value of its operand, an absolute expression, and assign the value of the result to the symbol in the label field. The label field must specify a unique name. It is not possible to recompute a symbol using EQU; see SET below, if recalculation is required. The label is required.

```
MEMSIZE     EQU         128
```

will assign the value 128 to `MEMSIZE`.

Any expression which yields an absolute result may be used in the operand field. All the symbols used in the expression MUST BE defined prior to the EQU directive in the assembly. Use of a symbol defined later in the assembly will cause an error. EQU is a handy way to replace a numeric constant with a more readable symbol, thus making it easier to follow the meaning of the code. Also, if the value requires a change, you can do so simply by altering the EQU, rather than by searching for each line that contains the value.

4.5 FAIL: Indicate an error during assembly

The FAIL directive may be used to indicate that an error has been made in specifying conditional assembly parameters or some other assembly time specification. The syntax of FAIL is

```
            FAIL        ERROR+13
```

the operand field may contain any absolute expression, this is the number of the error that will be reported when the FAIL directive is executed. There is an example of this in the `syscalls.mac` file.

4.6 INCLUDE: Include a Source File

The INCLUDE directive will include the named source file from a disk. The file name may specify a disk drive and a qualifier. If it doesn't, the current working disk is used, and the qualifier is "S".

```
INCLUDE         /INCLUDE/TRAPS.MAC
```

will include the file named `TRAPS.MAC` from the `/INCLUDE` directory of the current working disk. INCLUDEs may be nested to a level of 5.

4.7 LIST: Turn on Listing switch

The LIST directive will cause the source code following it to be listed on the output. LIST will enable listing of the source program after a NOLIST.

```
LIST
```

LIST has no operands and should not have a label.

4.8 NOLIST: Turn off Source Listing

The NOLIST directive will terminate the listing of source statements until a LIST directive is found. Typical use would be to avoid printing out an INCLUDE file. For example:

```
NOLIST
INCLUDE         /INCLUDE/TRAPS.MAC
LIST
```

No label or operands should be placed on the NOLIST directive.

4.9 OPT: Change Assembler Options

The OPT directive allows you to set and change various options involving the assembly of your program. Each option is denoted by a unique keyword, and any number of keywords may be specified in the operand field of the instruction. Each keyword is separated from the next by a comma.

```
OPT             BRL,FRL         Forward References Long
```

forces all forward references in branch and other instructions to be assembled using long forms.

The keywords and their meanings are as follows

BRL	All forward branches will use the long form (default).
BRS	All forward branches will use the short form.
FRL	All forward data references will use the long form (default).
FRS	All forward data references will use the short form.
PCO	PC relative addressing. Backward references will use PC relative addressing when possible (default).
NOPCO	Disable PCO.

4.10 ORG: Define Absolute Origin

The ORG directive defines the start address of a section of code and must have as an operand an absolute expression.

```
            ORG         $4000
```

defines an absolute section starting at hexadecimal location 4000.

The label field, if present, is ignored.

You may not have more than 16 ORG statements in a single assembly module.

4.11 PAGE: Skip to the next page

The listing of your program is continued at the top of the next page. `SSASM` will write the required number of carriage return-line feeds to the listing file.

The PAGE directive should not have any label or operands. The PAGE directive is not printed on the listing.

4.12 REG: Define a register list for MOVEM

REG will define a symbol which can be used in the register list portion of the MOVEM instruction. This allows you to determine what registers are used in a subroutine and insure that the same registers are saved and restored at entry and exit.

```
REG1LIST    REG         D0-D3/D6/A0-A5
            MOVEM       REG1LIST,-(SP) SAVE REGISTERS
            MOVEM       (SP)+,REG1LIST RESTORE REGISTERS
```

the register list above denotes registers D0 thru D3 (D0-D3), D6 and A0 thru A5 (A0- A5).

Sequences of registers are separated by a dash, while different lists are separated by a slash. Any number of registers may be specified in a list. Registers separated by a dash must be of the same type: data or address. There is an example of this in the `syscalls.mac` file.

4.13 SPC: Insert Spaces in Listing

The SPC directive requires an operand field containing an absolute expression which will cause a number of blank lines (carriage return-line feeds) to be inserted in the listing file. The same effect may be had by inserting the null lines yourself in the input source file.

```
            SPC         2
```

will space 2 lines.

The SPC directive will never space beyond the end of the current listing page.

4.14 SET: Compute a new value for a label

SET is similar to EQU, except that the label field may be redefined any number of times through the assembly. The syntax is identical to the EQU directive above.

```
PARM1       SET         16
PARM1       SET         32
```

will set the value of PARM1 to 16 and 32 respectively.

The same restrictions apply to the SET operand expression as apply to the EQU operand expression (see EQU above).

4.15 TTL: Define the Listing Title

The TTL directive may be used to define a title which will appear at the top of every page of your listing. The title line is defined as every character in the operand field from the first non-blank character after the directive to the end of the line, including blanks and tabs.

```
        TTL             INPUT/OUTPUT SUBSYSTEM ROUTINES.
```

There is no limit to the number of TTL's in a program. Each TTL will cause a new page to be started. The TTL directive itself is not printed.

5
Structured Constructs

Unfortunately, the vast majority of the structured constructs mentioned below have failed to work in most versions of this assembler. Don't count on them (but please do tell us which ones you got working).

5.1 Structured Programming Constructs

This chapter explains the structured programming constructs that SSASM provides to ease your programming, provide some higher level language benefits, and help you organize your programs in a more logical and precise manner.

The constructs supplied for structured programming include

- FOR/ENDF
- IF/THEN/ELSE/ENDI
- REPEAT/UNTIL
- WHILE/DO/ENDW.

FOR allows counting loops which may either increment or decrement.

IF allows a choice between one of two alternative sequences of code, one of which may be to do nothing.

REPEAT allows you to code a general loop with the test of the loop performed at the end.

WHILE allows a general loop with the loop test performed at the beginning.

Unlike other statements in SSASM, the operand field of structured statements may contain blanks to improve readability.

5.2 FOR/ENDF Loops

The FOR loop permits you to code a counting loop which increments or decrements some counter by some value. The statements are coded as follows

```
        FOR.q       o1 = o2 TO o3 [BY o4] DO.e
```

any assembler statements including structured.

```
        ENDF
```

or

```
        FOR.q       o1 = o2 DOWNTO o3 [BY o4] DO.e
```

any assembler statements including structured.

```
        ENDF
```

In the above, 'q' may be any of the usual qualifiers B, W, or L for the length of the index of the loop. 'e' is the extent of the branch, and may be either S or L for short or long.

'o1' contains the index of the loop and must be a data alterable location, either a memory location (not using the PC) or a register. If o1 is a register, then the 'B' qualifier may not be used.

'o2' is the starting value of the loop; it may be any form of operand, including immediate.

'o3' is the ending value of the loop; it may also be any form of operand, including immediate.

'o4' is the increment (or decrement) value of the loop; it may also use any form of operand. If 'BY o4' is not specified (which is what the square brackets mean) it will default to #1.

The first form, using the TO, is an incrementing loop; while the second form, using the DOWNTO, is a decrementing loop. In the first case o4 is added to o1, while in the second case, o4 is subtracted from o1.

For incrementing loops, o2 must be less than o3 for the loop to execute, while for decrementing loops, o2 must be larger than o3.

To execute an incrementing loop, o1 is initially assigned the value of o2. If o1 is less than o3 the statements between the FOR and the ENDF will be executed. (If the value of o2, assigned to o1, is greater than the value of o3 the loop is never executed.)

At the ENDF statement, the value of o4 is added to o1 and the loop is repeated from the comparison of o1 and o3.

Only certain combinations of o1, o3, and o4n are allowed. o1 and o3 must be combinable in a CMP, CMPA or CMPI instruction. o1 and o4 must be combinable in an ADD, ADDI or ADDQ instruction.

The following are examples of the use of FOR/ENDF, the statements in vertical bars are the 68000 equivalent of the FOR/ENDF logic.

```
*           A3 POINTS TO 16 BYTES TO ADD UP.

            CLR.L           D0              CLEAR THE RESULT
   FOR.L                    D1 = #0 TO #15 DO.S     LOOP OVER 16 NUMBERS
            |               MOVEQ.L          #0,D1      |
            |L0:                                        |
            |               CMP.L            #15,D1     |
            |               BHI.S            L1         |

            ADD.B           0(A3,D1),D0     ADD THE NEXT NUMBER

   ENDF                                     END OF LOOP
            |               ADDQ.L          #1,D1      |
            |               BRA.S            L0         |
            |               L1:                         |
```

In the above 'L0' and 'L1' are symbolic internal labels, the real labels will never conflict with any label acceptable to SSASM from your program.

5.3 IF/THEN/ELSE/ENDI

IF statements allow you to select one of two possible sequences of statements and machine instructions; one of the sequences may be empty. The forms of the IF/THEN/ELSE/ENDI statements are

```
            IF.q  expression  THEN.e
any assembler statements including structured.
            ENDI
```

or

```
            IF.q  expression   THEN.e
any assembler statements including structured.
            ELSE.e
any assembler statements including structured.
            ENDI
```

'q' and 'e' have the same meanings as in the FOR statement above.

The 'expression' above can be composed of a single condition, two operands compared with a condition, or two comparisons connected by an AND or OR logical operator.

The simplest expression is a condition, which is any branch condition permitted on the Bcc instruction enclosed in "<" and ">". The following are permitted

```
<CC>    <CS>    <EQ>    <GE>    <GT>
<HI>    <LE>    <LS>    <LT>    <MI>
<NE>    <PL>    <VC>    <VS>
```

The following IF statements would generate the code in vertical bars.

```
IF      <EQ>            THEN.S
        |               BNE.S L0       |
ELSE.S
        |               BRA.S L1       |
        |L0:                           |
ENDI
        |L1:                           |
```

Note that the condition in the branch instruction has been reversed from the original condition. This is because the code after the `IF` is to be executed whenever the condition is TRUE; therefore a branch is done when the condition is FALSE. The logical negation of EQ is NE.

If two operands are separated by a condition, then the operands must be combinable in a CMP, CMPA, CMPI or CMPM instruction. The following might be used

```
IF.L  D3 <NE> #5        THEN.S
        |               CMP.L          #5,D3          |
                        BEQ.S L0                      |

ELSE.L
        |               BRA.L L1                      |
        |L0:                                          |
ENDI
        |L1:                                          |
```

Note that `SSASM` reversed the order of the operands for the CMP instruction, and that the NE condition has been properly reversed for the BEQ instruction.

A condition and a comparison can be combined with the AND logical operator to produce the following

```
IF.L    <GT> AND D3 <LT> FIELD1(A3) THEN.L
        |               BLE.L L0                     |
        |               CMP.L FIELD1(A3),D3          |
        |               BLT.L L0                     |
ELSE.L
        |               BRA.L L1                     |
        |L0:                                         |
ENDI
        |L1:                                         |
```

Note that since the operands have been reversed in the CMP instruction, the second Bcc still has a condition of LT. The reversal of the operands changes the original LT to GE (a < b implies b >= a) and the negation for the branch returns the original LT condition.

Finally, it is possible to combine two comparisons

```
IF.L    D2 <GE> D4 OR D3 <EQ> D5 THEN.L
        |               CMP.L D2,D4                 |
        |               BGE.S L0                    |
        |               CMP.L D3,D5                 |
        |               BNE L1                      |
        |L0:                                        |
ELSE.L
        |               BRA.L L2                    |
        |L1:                                        |
ENDI
        |L2:                                        |
```

Note the first branch in the IF. If the first condition is true, (D2 is greater than or equal to D4) we must execute the statements following the IF. The second condition is not tested and a short branch is done to the label at the end of the IF. Note also that the second condition has been reversed as usual.

5.4 REPEAT/UNTIL

The REPEAT/UNTIL construct allows you to code a general loop with the repeatition test at the end of the loop. The loop will always be executed once, before the conditions of the loop are tested. The format of the REPEAT/UNTIL is

```
            REPEAT
any assembler statements including structured
            UNTIL.q       expression
```

where 'q' is the length code to be used in evaluating the expression.

The expression must conform to the same rules as given above for the expression in the IF statement.

The following is an example of a REPEAT/UNTIL loop

```
            * Search the vector of words starting at VECTOR for
            * an entry matching the contents of D0.
            LEA           ENDVEC,A2     A2 - LAST ENTRY IN VECTOR
            LEA           VECTOR-4,A3   A3 - FIRST ENTRY-1
   REPEAT
            |L0:                                          |

            ADDQ.L #4,A3 INCREMENT POINTER
            CMP.L (A3),D0 COMPARE FOR RESULT
   UNTIL.L  <EQ> OR A2 <EQ> A3
            |             BEQ.S L1                       |
            |             CMPA.L A2,A3                   |
            |             BNE.S L0                       |
            |L1:                                         |
```

As usual the code generated by SSASM is in vertical bars.

Note that any of the expression formats permitted in the IF statement above are also permitted in the UNTIL statement. The same considerations also apply for their generation. (Eg. the operands must be of compatible forms.)

5.5 WHILE/DO/ENDW

The WHILE/DO/ENDW construct allows you to code general loops where the condition of the loop is tested at the beginning. The loop may not be executed at all if the conditions are not meet.

The format of the WHILE/DO/ENDW loop is

```
            WHILE.q  expression  DO.e
any assembler statements including structured
            ENDW
```

As above, 'q' is the qualifier that determines the length codes used in evaluating the expression, and 'e' is used to determine the size of the forward branches, if the expression is not true. The expression may be any expression valid in a IF statement above. The same restrictions apply.

An example of a WHILE loop is

```
            * Count the characters in an input line, result is in D0
LF          EQU           10            LINE FEED
            LEA.L         INLINE,A3
            CLR.L         D0
   WHILE.B #LF <NE> (A3)+ DO.S
            |L0:                                          |
            |             CMP.B #LF,(A3)+                  |
            |             BNE.S L1                         |
            ADDI.L #1,D0
   ENDW
            |             BRA.S L0                         |
            |L1:                                           |
```

As usual, the code generated by SSASM is in vertical bars.

5.6 Nesting

The above control structures may be nested, contained within one another, to a maximum level of 16.

The start and end of each control structure which is nested in another must be completely contained in the enclosing control structure. It is illegal to have the end of an IF, for example, after the end of a WHILE the IF is enclosed in.

The following is valid

```
IF ...
WHILE ...
ENDW
ELSE
REPEAT
IF ...
ENDI
UNTIL ...
ENDI
```

Note that the WHILE is entirely contained in the first clause of the IF, and the REPEAT is entirely contained in the second clause. The second IF is also entirely contained in the REPEAT.

6
Macros and Conditional Assembly

This chapter explains the macro and conditional assembly features of SSASM. Macros allow you to code repetitive sequences of code in one place, and use modified versions of them in many other places.

Conditional assembly allows you to customise your program for different parameters such as storage size, I/O devices etc.

6.1 MACRO Definitions

Macro definitions provide the prototype statements for your macros to the assembler. They must always begin with the MACRO directive and end with the ENDM directive. The MACRO statement must have a label field which specifies the unique name of your macro.

```
label       MACRO
            ...  macro definition statements
            ENDM
```

The statements between the MACRO and ENDM statements form the prototype for the macro. Any statements may be used, including assembler directives and conditional assembly directives.

None of the statements in the macro definition are assembled at the time the definition is processed, they are stored in memory until required for expansion. (Macros and the assembler symbol table share the same memory space.)

Macro definitions must precede their usage; therefore it is a good idea to place all of your macro definitions at the beginning of your program. If you are developing a program in multiple modules; put your macros in a macro library and use the -m switch on the SSASM command.

All comment and null statements are removed from the macro definition when it is stored in memory, to save space. They will not be present when the macro is expanded.

An example of a simple macro that executes a trap instruction is

```
TRAP7       MACRO
            TRAP #7
            ENDM
```

6.2 MACRO Expansion

You invoke a macro in your code by placing in the operation field the name of the macro (assigned on the MACRO directive), followed by an optional qualifier, followed by any parameters. A label may also be specified if desired.

```
[label]     macro-name[.qualifier] [parameters]
```

If the TRAP7 macro above were invoked in the assembler, you might see

```
            TRAP7 2 000000 4E47 + TRAP #7
```

expanded in the code. The `2` is the section number of this code. The `000000` is the address of the instruction, `4E42` is the assembled instruction (hexadecimal) and the '`+`' indicates the instruction was assembled inside a macro.

Macro definitions may contain statements that invoke other macros; ie. macro invocations may be nested. The maximum level of nesting is five. Macro invocations may be recursive, but the maximum nesting level must not be exceeded.

6.3 MACRO Parameters

Macro parameters are strings of characters, separated by commas, placed on the macro invocation statement. These parameters may be used inside the macro expansion anywhere you wish.

You control the placement of these strings by putting a backslash followed by a digit or letter in your prototype statements where you wish to place the appropriate parameter.

Parameter 0 is the qualifier that was used on the macro invocation, and parameters 1 thru 9 are the first 9 parameters given in the operand field of the macro invocation.

The macro defintion

```
TESTMAC     MACRO
            TST.\0  \1
            BNE.L  \2
            ENDM
```

will cause the following expansion

```
            TESTMAC.B     DATA,LOCATION
+           TST.B         DATA
+           BNE.L         LOCATION
```

Of course the symbols `DATA` and `LOCATION` must be appropriately defined.

You may have up to 35 parameters on a macro invocation. The first 9 are address by 1 thru 9, the next 26 are addressed using the letters A thru Z. Macro invocation statements may be continued on more than one line, by simply coding the comma after the last parameter on the line and continuing with the next parameter on the next line. No comments may be placed between continuation lines. No null lines are permitted. At least one blank must precede the parameters on the continuation lines.

The TESTMAC example above could have been coded

```
            TESTMAC.B     DATA,
            LOCATION
+           TST.B         DATA
+           BNE.L         LOCATION
```

WARNING: No other statements in `SSASM` may be continued.

No blanks, tabs or commas are permitted in parameters coded in the above fashion. (Lower case alphabetics are, however.) To specify a parameter containing these characters, enclose the parameter in angle brackets

```
MESSAGE     MACRO
            DC.B          '\1'
            DC.B          0
            ENDM
```

will generate

```
            MESSAGE       Please, don''t walk on the grass.
+           DC.B          'Please, don''t walk on the grass.'
+           DC.B          0
```

Parameters may use either form, and the forms may be intermixed.

6.4 Unique Label Generation

It is frequently the case that macros need internal labels for branching and other uses. Quite often these labels must be unique since the macro may be used more than once in an assembly.

The \ @ symbol generates a unique 4 digit number which is the same inside a single macro invocation, but different for every invocation. It is simply the sequential number of the invocation found by SSASM.

The macro definition

```
SUM         MACRO
            CLR.\0          \3
            LOOP\@:
            ADD.\0          (\1)+,\3
            DBF             \2,LOOP\@
            ENDM
```

could generate

```
            SUM.L           A2,D1,D0
+           CLR.L           D0
+LOOP0001:
+           ADD.L           (A2)+,D0
+           DBF             D1,LOOP0001

            SUM.B           A3,D2,D3
+           CLR.B           D3
+LOOP0002:
+           ADD.B           (A3),D3
+           DBF             D2,LOOP0002
```

The first invocation generated the label LOOP0001, while the second generated LOOP0002.

Nesting invocations does not affect the uniqueness of the number. A nested invocation will be assigned a different number from the enclosing invocation, and the enclosing number will be restored after the nested expansion is complete.

6.5 Number of Arguments to a MACRO Invocation

The special name NARG is defined inside macro expansions as the number of arguments (parameters) coded on the macro invocation statement. The qualifier is not included in this count. If no parameters are provided the count is zero.

```
TABLE       MACRO
            IFNE            NARG
            DC.W            NARG
            DC.W            \1
            IFNE            NARG-1
            DC.W            \2
            ENDC
            ENDC
```

might generate

```
            TABLE           2,3
+           IFNE            NARG
2 000000 0002 +             DC.W NARG
2 000002 0002 +             DC.W 2
+           IFNE            NARG-1
2 000004 0003 +             DC.W 3
+           ENDC
+           ENDC
```

There are other examples in the syscalls.mac file.

6.6 MEXIT Directive

The MEXIT directive may be used only inside macros. If it is executed during the macro expansion, the remainder of the macro expansion is skipped, and the next assembler statement processed is the one after the orginal invocation statement.

Usually, the MEXIT directive will be in a conditionally assembled portion of the macro expansion.

The MEXIT directive has no label and no operands, although it may have comments.

```
MEXIT          [comments]
```

6.7 Conditional Assembly

Conditional assembly allows you to code a single general purpose program module, and by changing parameters assemble it in a number of differing configurations. It is particularly useful inside macro expansions.

Conditional assembly provides two basic directives, an IFxx directive, and the ENDC directive. The IFxx directive is used to test some condition (see below). If that condition is TRUE, the code between the IFxx and it's corresponding ENDC will be assembled. There are examples in the `syscalls.mac` file.

If the condition is FALSE, the code between the IFxx and the ENDC is NOT assembled.

Each IFxx must be matched with an ENDC. IFxx/ENDC combinations may be nested to any level (less than 32768). If an enclosing IFxx has a FALSE condition, NONE of the enclosed code is assembled, even if some enclosed IFxx has a TRUE condition.

The IFxx directive may be used to compare strings, or to compare an absolute numeric expression against zero. IFC compares two strings for equality, IFNC compares two strings for inequality.

```
    IFC          'string1','string2'
... assembled if string1 = string2
    ENDC
```

and

```
    IFNC 'string1','string2'
... assembled if string1 not = string2
    ENDC
```

The strings must be enclosed in apostrophes, a single apostrophe must be represented by two apostrophes in a row. Symbolic substitution of macro parameters may be done inside the strings.

Numeric expressions may be compared to zero using any of the following IFxx directives.

IFEQ	expression [= 0]
IFNE	expression [not = 0]
IFLT	expression [less than 0]
IFLE	expression [less than or = 0]
IFGT	expression [greater than 0]
IFGE	expression [greater than or = 0]

If the expression on the IFEQ is equal to zero, the code following the IFEQ will be assembled. Similarly, if the expression on the IFGT is greater than zero, the code following the IFGT will be assembled. The other directives operate in a similar manner.

The following example demonstrates the use of these facilities

```
SWITCH      SET         0
            IFEQ        SWITCH
            NOP         ASSEMBLED
            ENDC
SWITCH      SET         1
            IFEQ        SWITCH
            NOP         NOT ASSEMBLED
            ENDC
```

7
Error Messages and Changes

7.1 Error Messages

The following is a list of all the messages produced by SSASM, together with some explanations of their possible cause. The number of error messages has been increased considerably, and they are now very much clearer than in earlier versions. Beginners should remember that the assembler is **not** intelligent; if it encounters an error, it may give misleading messages where the circumstances of the error are not clearcut.

Error Number	Explanation
1	Bad file name Source file name is not correct, or the source file is not on the disk. Fatal error (naturally).
2	Bad argument Parameter list is not valid for SSASM. An invalid parameter may have been entered, or the format may not be correct.
3	File name not correct. Fatal error. Check the file names given on the SSASM command, they may be too long or have an invalid character.
4	Unexpected EOF The end of file was encountered unexpectedly. The source program is missing the END directive, or mismatched conditional assembly directives, or mismatched MACRO/ENDM directives.
5	Out of memory Insufficient memory in 1616 to assemble the program.
8	Can’t open the input file. Probably the file name is incorrect.
9	Can’t open the input file on pass 2. If this happens report it to Applix immediately.
11	Can’t open output Can’t open the object file. Insufficient space on the disk may cause this error.
24	Error closing input Error closing the input file. Report to Applix immediately.
	Relocatable instruction used without -r flag
100	Bad effective address syntax The syntax of an effective address operand is incorrect.
101	Bad effective address or displacement Effective address mode is not proper for this instruction and operand combination. Could also be that the value of a displacement or other data is out of the valid range.
102	Bad label, opcode, or symbol Invalid label, operation code, or symbol.
103	Bad operand or unexpected EOL Invalid operand or unexpected end of line.
104	Too many absolute sections Too many ORG absolute (15 is the maximum).

105 Bad opcode
Operation code not recognized.

Code generated in OFFSET section

107 Redefined name
Duplicately defined name.

108 Undefined name

IDNT misplaced

110 Bad expression
Error in expression processing. Invalid operand.

111 Bad assembler directive
Misplaced or invalid assembler directive.

120 Error in macro, conditional or structured thingy
Error in macro, conditional assembly, or structured programming constructs.

130 Include level too deep
Include level is too deep. Limit is 5.

131 Macro level too deep
Macro level is too deep. Limit is 5.

199 Phase error
Phase error, label has different value in pass 2 than the one defined in pass1. This is usually caused by some previous error. Report to Applix if it persists.

998 Symbol table file error
Error processing the symbol table overflow file.

Out of memory

Address misalignment (check 'dc.b' statements)

Cannot allocate memory at $4000

Internal error - report to Applix

Stack frame

7.2 Version changes

Version 1.2

- Output files close correctly when assembly is aborted by Ctrl C or by an assembly error.
- Default output file now ends with `.exec` extension, although use of `-o` switch can override this.
- DS pseudo-opcode now fills defined space with zeros, and allocates space for this in the output file.
- The MOVE CCR, <EA> instructions now work.
- Error reporting cleaned up. The line where the error occured is printed out.

Version 1.3

- Line numbers within include files and macros are now tracked correctly.
- Memory allocator is used for internal storage, if running under 1616/OS Version 3.0 or higher. Uses old system if eproms are old. Thus, `SSASM` 1.3 and up should also run on old version 1616/OS.

- Double quotes (") are accepted as well as single quotes in strings.
- DS pseudo-opcode now works for any amount of space (the 127 byte limit no longer applies).

Later Versions

- Both * and ; are accepted as comment separators.

Table of Contents