SUBQMOD'S
MACRO ASSEMBLER
FOR THE
ATARI JAGUAR

Programmers Reference Manual
For Release 1.0.12 (BETA)

SubQMod's Macro Assembler for the Atari Jaguar

Introduction

This document describes SMAC, **S**ubQMod's **MAC**ro Assembler (pronounced 'ess-mack'), which generates code for the processors of the classic and powerful Atari Jaguar console. SMAC is designed to be familiar with existing Jaguar developers by operating in a near identical manner to MADMAC, Atari's original Jaguar assembler. In fact, the foundation to SMAC is the MADMAC source code that existed prior to being modified for the Atari Jaguar and is used with the kind permission of its author, Landon Dyer. SMAC is partially an homage to MADMAC and the fantastic Atari Jaguar console as well as a platform to provide Windows, MAC and Linux users with the capability of once more creating great code for this great system. It will generate code for the Jaguar's key and most powerful chips, the GPU and DSP RISC processors, as well as the Motorola 68000 processor. SMAC is a fast, single pass, assembler that supports includes files, macros, local and external symbols, limited control structures and other features.

It is always recommended to code as much as possible of your applications utilizing the RISC processors, especially GPU code running in main memory. Atari stated that this was not possible due to hardware constraints of the Jaguar but it is now known that this is possible following a few basic rules. SMAC is designed to assist developers in compiling code accurately to execute in main memory by checking the 'execution in main' rules and either notifying the user or modifying code to meet those rules.

The Command Line

The assembler is called SMAC.EXE and is natively compiled for Win32 based systems such as Windows XP, 2000, 2003 etc. SMAC can, and is, being compiled on various other systems such as Mac OS/X and Linux and is processor endian aware.

The command line takes the form of:

```
smac [switches] [sourcefile]
```

A command line consists of any number of switches followed by the name of the file to be assembled. A switch is specified by a dash ("-") followed immediately by a key character. Some switches accept or require arguments to follow the key character. Key Characters are not case-sensitive, so "-c" and "-C" produce the same effect.

Switch order can be important. Command lines are processed from left to right in one pass and switches usually take effect when they are encountered. It is best to specify all switches before listing the name of the input file.

If the command line is empty, SMAC prints a copyright message and displays the list of available switches.

Input files are assumed to have the extension ".s" and SMAC will look for a file with this extension if none is specified.

SMAC normally produces object code files with the same filename as the input source file, except with a ".o" extension. The "-o" switch can be used to change the output filename.

Command Line Switches

A summary of the available command line switches is shown below.

Switch	Description
-? or -h	Display SMAC usage information. Use -? for Windows based systems and –h for all other systems.
-d *symbol[=value]*	This switch permits symbols to be defined on the command line. The symbol name may optionally be followed by and equals sign ("=") and a decimal number for the value to be assigned to the symbol. If no value is specified the symbol's value will be set to zero. This switch is most useful for enabling conditional assembly of the source code. For example: **-dDEBUG –dLoopCount=999 –dDebugLevel=55** This would define 'DEBUG' with a zero value, 'LoopCount' with a value of 999 and 'DebugLevel' with a value of 55.
-e *[errorfile]*	This switch causes SMAC to send error messages to a file instead of the console. If a filename immediately follows, error messages are written to the specified filename. If no filename is specified, a filename is created with the default extension or ".ERR" and the root name taken from the first input file. (i.e. error messages are written to FILE.ERR if the first input filename is FILE or FILE.S) If no errors are encountered then no error message file will be created.
-f*[format]*	Select object file format to be output: **-fb**: BSD COFF (Default Output) Additional formats are planned for future releases.
-i*[path]*	The "–i" switch allows for automatic directory searching for include files. A list of semi-colon separated directory search paths may be listed following the switch (however no spaces can appear in the paths). For example; **-im:;c:\include;c:\include\sys** This will cause SMAC to search the current directory of drive M and the directories INCLUDE and INCLUDE\SYS on the C drive. If the "-i" switch is not specified, SMAC searches for the SMACPATH environment variable (in that order) which is used specifically to specify include file directories in the same way. For example; **set SMACPATH=m:;c:\include;c:\include\sys** This will cause SMAC to search the same directories as the previous example.

	If you are using a MAKE utility, and in your MAKEFILE you need to use the "-i" option to specify a certain include path for specific files, but you also need access to the paths specified by SMACPATH, you can do something like this: **-iproject\inc;$(SMACPATH)** The $(SMACPATH) macro will be expanded by your MAKE utility into the contents of the SMACPATH environment variable. This is a standard feature of nearly all MAKE utilities.
-l	The **-l** switch causes SMAC to generate an assembly listing file. A filename is created with the default extension of ".PRN" and the root name taken from the first input file. (i.e. the listing is written to FILE.PRN if the first input file is FILE or FILE.s).
-o *file*	The **–o** swich causes SMAC to write its object code to the specified file. No default extension is applied to the filename so you need to specify whatever extension is appropriate.

-r *[size]*	The **–r** switch causes SMAC to automatically pad the size of each segment in the output file until the size is an integral multiple of the specified boundary. *Size* is a letter that specifies the desired boundary.
	
	`-rw  Word (2 bytes, default alignment)`
	`-rl  Long (4 bytes)`
	`-rp  Phrase (8 bytes)`
	`-rd  Double Phrase (16 bytes)`
	`-rq  Quad Phrase (32 bytes)`
	
	For example, if the TEXT segment of the output file would normally be 434 bytes long then using the "-rp" switch would cause it to be padded in length to 440 bytes long, which would make the end of the segment fall on a phrase boundary.

-s	Warn about possible short branches.
-v	Set verbose mode. This will cause SMAC to print out the names of each source file and include file as they are processed.
-y[pagelen]	Set the output list file page length, the default being 61.

Using SMAC

Lets assemble some sample files. Load your favorite text editor and create a small text files that looks like this:

```
      .include "jaguar.inc"
start:
      move.w      #$FF80, BG
      illegal
      .end
```

Save the file as plain ASCII text to the filename TEST.S. Exit your editor and at the command line type the following command:

```
smac test.s
```

Assuming your system is setup correctly, this will call SMAC, which will assemble TEST.S and produce an object module file named TEST.O. If you see an error message telling you that SMAC cannot find the "JAGUAR.INC" include file then chances are that you do not have your SMACPATH environment variable set correctly.

So now we have an object module, which isn't much use by itself until you run it through the linker, probably with other object modules, to create an executable progam. Atari's ALN linker can be utilized as can SLN, SubQMod's Linker for Atari Jaguar. For example;

```
aln –e –a 802000 x 4000 –o test.cof test.o
```

or

```
sln –e –a 802000 x 4000 –o test.cof test.o
```

Now lets try a few other command line options. Reload your text editor and load TEST.S into it again. Change the text to look like this:

```
      .include "jaguar.inc"
start:
      .if color1
      move.w      #$FF80, BG
      .else
      move.w      #$FF40, BG
      .endif

      illegal
      .end
```

Again, save the file as plain ASCII text. This time use the filename TEST2.S. Exit your editor and at the command line type the following command:

```
smac –l –o test2.obj –Dcolor1=1 test2.s
```

This produces an assembly listing file named TEST2.LST, writing an object file with a name of TEST2.OBJ and defines the symbol "color1" to have a value of 1 when the file TEST2.S is assembled.

Text File Format

SMAC expects source code files to conform to the following rules:

- Files must contain characters with ASCII values less than 128; it is not permissible to have characters with their high bits set unless those characters are contained in strings (i.e. between single or double quotes) or in comments.
- Lines of text are terminated with carriage-return/linefeed, linefeed alone, or carriage-return alone. (Carriage Return is ASCII value 13. Linefeed is ASCII value 10.)
- The file is assumed to end with the last terminated line or with a Control-Z (ASCII 26). If there is text beyond the last line terminator, it is ignored.

Statements

A statement may contain up to four fields which are identified by order of appearance and terminating characters. The general form of an assembler statement is:

```
label:     operator operand(s) ; comment
```

The label and comment fields are optional. An operand field may not appear without an operator field. Operands are separated with commas. Blank lines are legal. If the first character on a line is an asterisk (*****) or semicolon (**;**) then the entire line is a comment. A semicolon anywhere on the line (except in a string) begins a comment field which extends to the end of the line.

The label, if it appears, must be terminated with a single or double colon. If it is terminated with a double colon it is automatically declared global. It is illegal to declare a confined symbol global.

Equates

A statement may also take one of these special forms:

```
Symbol     equ        expression
symbol     =          expression
symbol     ==         expression
symbol     set        expression
symbol     reg        expression
```

The first two forms are identical; they equate the symbol to the value of an expression, which must be defined (no forward references). The third form, double-equals (==), is similar except that it also makes the symbol global. The fourth form allows a symbol to be set to a value any number of times, like a variable. The last form equates the symbol to a 16-bit register mask specified by a register list.

It is possible to equate confined symbols. For example:

```
cr          equ         13              ; carriage-return
lf          =           10              ; line-feed
DEBUG       ==          1               ; global debug flag
count       set         0               ; variable
count       set         count + 1       ; increment variable
.cr         =           13              ; confined equate
```

Register equates can also be defined and used as per the GASM assembler. They take the form;

```
Symbol      .equr           register[,bank]
Symbol      .regequ         register[,bank]
```

These can be used in the code to make it more readable and structured as well as being able to ensure correct register bank usage in tandem with the **`.regbank0`** and **`.regbank1`** directives.

Register equates can be undefined at any point.

```
.equrundef      Symbol
.regundef       Symbol
```

Symbols and Scope

Symbols may start with an uppercase or lowercase letter (A-Z, a-z), an underscore (_), a question mark (?) or a period (.). Each remaining character may be any of these characters, except a period, a numerical digit (0-9) or a dollar sign ($). Symbols are terminated with a character that is not a symbol continuation character (e.g. a period or comma, whitespace, etc.).

Case is significant for user-defined symbols, but not for 68000, GPU or DSP mnemonics, assembler directives or register names. Symbols are limited to 100 characters. When symbols are written to the BSD object file the entire symbol, up to a 100 characters, is used.

For example, all of the following symbols are legal and unique:

```
reallyLongSymbolName
.reallyLongConfinedSymbolName
a10
.a10
ret
.ret
dc
.dc_move
.9
.????
.0
.00
.000
.1
_frog
```

```
?zippo?
sys$system
atari
Atari
ATARI
aTaRi
```

While all of the following symbols are illegal:

```
12days          dc.10      dc.z            'quote
.right.here     @work      hi.there        $money$
~tilde
```

Symbols beginning with a period (.) are confined; their scope is limited to the space between two normal (unconfined) labels. Confined symbols may be labels or equates. It is illegal to make a confined symbol global (with the `.globl' directive, a double colon, or a double equals). Only unconfined labels delimit a confined symbol's scope; equates (of any kind) do not count. For example, all symbols are unique and have unique values in the following:

```
zero::      subq.w     #1,d1
            bmi.s      .ret
.loop:      clr.w      (a0)+
            dbra       d0,.loop
.ret:       rts
FF::        subq.w     #1,d1
            bmi.s      .99
.loop:      move.w     #-1,(a0)+
            dbra       d0,.loop
.99:        rts
```

Confined symbols are useful since the programmer has to be much less inventive about finding small, unique names that also have meaning.

It is legal to define symbols that have the same names as processor mnemonics (such as `move' or `rts') or assembler directives (such as `.even'). Indeed, one should be careful to avoid typographical errors, such as this:

```
            .gpu
.org        =          G_RAM
```

which equates a confined symbol to the value of the G_RAM equate, rather than setting the code generation address which the .ORG directive does (if the equal sign was not there).

Keywords

The following names, in all combinations of uppercase and lowercase, are reserved keywords and may not be used as symbols (e.g. labels, equates, or the names of macros):

```
equ   set   reg   sr    ccr   pc    sp    ssp   usp
d0    d1    d2    d3    d4    d5    d6    d7
a0    a1    a2    a3    a4    a5    a6    a7
r0    r1    r2    r3    r4    r5    r6    r7
r8    r9    r10   r11   r12   r13   r14   r15
r16   r17   r18   r19   r20   r21   r22   r23
r24   r25   r26   r27   r28   r29   r30   r31
```

Constants

Numbers may be decimal, hexadecimal, octal, binary or concatenated ASCII. The default radix is decimal, and it may not be changed. Decimal numbers are specified with a string of digits (0-9). Hexadecimal numbers are specified with a leading dollar sign ($) followed by a string of digits (0-9) and uppercase or lowercase letters (A-F a-f). Octal numbers are specified with a leading at-sign (@) followed by a string of octal digits (0-7). Binary numbers are specified with a leading percent sign (%) followed by a string of binary digits (0-1). Concatenated ASCII constants are specified by enclosing one or more characters in single or double quotes. For example:

```
1234        decimal
$1234       hexadecimal
@777        octal
%10111      binary
"z"         ASCII
'frog'      ASCII
```

Negative numbers are specified with a unary minus (-). For example:

```
-5678       -@334       -$4e71
-%11011     -'z'        -"WIND"
```

Strings

Strings are contained between double (") or single (') quote marks. Strings may contain non-printable characters by specifying `backslash' escapes, similar to the ones used in the C programming language. SMAC will generate a warning if a backslash is followed by a character not appearing below:

```
Code        Value       Meaning
----        -----       --------------------
\\          $5C         backslash
\n          $0A         line-feed (newline)
\b          $08         backspace
\t          $09         tab
\r          $0D         carriage return
\f          $0C         form-feed
\e          $1B         escape
\'          $27         single-quote
\"          $22         double-quote
```

It is possible for strings (but NOT symbols) to contain characters with their high bits set (i.e. character codes 128…255).

You should be aware that backslash characters are popular in Windows path names, and that you may have to escape backslash characters in your existing source code. For example, to get the file "C:\JAGUAR\SMAC.S" you would specify the string "C:\\JAGUAR\\SMAC.s".

Register Lists

Register lists are special forms used with the **movem** 68000 mnemonic and the **reg** directive. They are 16-bit values, with bits 0 through 15 corresponding to registers D0 through A7. A register list consists of a series of register names or register ranges seperated by slashes. A register range consists of two register names, Rn and Rm, n < m, separated by a dash. For example:

Note: Older versions of Atari's MADMAC assembler supported the use of R0, R1, … R15 as register names. This is not supported in SMAC as these are reserved as Jaguar GPU and DSP register names.

```
Register List               Value
-------------               -----
d0-d7/a0-a7                 $FFFF
d2-d7/a0/a3-a5              $39FC
d0/d1/a0-a3/d7/a6-a7        $CF83
d0                          $0001
```

Register lists and register equates may be used in conjunction with the **movem** 68000 mnemonic, as in this example:

```
temps       reg         d0-d2/a0-a2       ; temp registers
keeps       reg         d3-d7/d3-a6       ; registers to preserve
allregs     reg         d0-d7/a0-a7       ; all registers
            movem.l #temps,-(sp)          ; these two lines ...
            movem.l d0-d2/a0-a2,-(sp)     ; ... are identical
            movem.l #keeps,-(sp)          ; save "keep" registers
            movem.l (sp)+,#keeps          ; restore "keep" regs
```

Expressions

All values are computed with 32-bit 2's complement arithmetic. For boolean operations (such as **if** or **assert**) zero is considered false, and non-zero is considered true.

Expressions are evaluated strictly left-to-right, with no regard for operator precedence.

Thus the expression "1 + 2 * 3" evaluates to 9, not 7. However, precedence may be forced with parenthesis (()) or square-brackets ([]).

Types

Expressions belong to one of three classes: undefined, absolute or relocatable. An expression is undefined if it involves an undefined symbol (e.g. an undeclared symbol, or a forward reference). An expression is absolute if its value will not change if the program were to be relocated (for instance, the number 0, all labels declared in an absolute section, and all Jaguar hardware register locations are absolute values).

An expression is relocatable if it involves exactly one symbol that is contained in a TEXT, DATA or BSS section.

Only absolute values may be used with operators other than addition (+) or subtraction (-). It is illegal, for instance, to multiply or divide by a relocatable or undefined value. Subtracting a relocatable value from another relocatable value in the same section results in an absolute value (the distance between them, positive or negative). Adding (or subtracting) an absolute value to or from a relocatable value yields a relocatable value (an offset from the relocatable address).

It is important to realize that relocatable values belong to the sections they are defined in (e.g. TEXT, DATA or BSS), and it is not permissible to mix and match sections. For example, in this code:

```
line1:      dc.l  line2, line1+8
line2:      dc.l  line1, line2-8
line3:      dc.l  line2-line1, 8
error:      dc.l  line1+line2, line2 >> 1, line3/4
```

Line 1 deposits two long words that point to line 2. Line 2 deposits two long words that point to line 1. Line 3 deposits two long words that have the absolute value eight. The fourth line will result in an assembly error, since the expressions (respectively) attempted to add two relocatable values, shift a relocatable value right by one, and divide a relocatable value by four.

The pseudo-symbol `*' (asterisk) has the value that the current section's location counter had at the beginning of the current source line. For example, these two statements deposit three pointers to the label `bar':

```
foo:        dc.l        *+4
bar:        dc.l        *, *
```

Similarly, the pseudo-symbol `$' has the value that the current section's location counter has, and it is kept up to date as the assembler deposits information "across" a line of source code. For example, these two statements deposit four pointers to the label "zip":

```
zip:        dc.l        $+8, $+4
zop:        dc.l        $, $-4
```

Unary Operators

Operator	Description
-	Unary minus (2's complement)
!	Logical (Boolean) NOT
~	Tilde: bitwise NOT (1's complement)
^^defined symbol	True if symbol has a value
^^referenced symbol	True if symbol has been referenced
^^streq string1 string2	True if the strings are equal
^^macdef macroname	True if the macro is defined

- The Boolean operators generate the value 1 if the expression is true, and 0 if it is not.
- A symbol is referenced if it is involved in an expression. A symbol may have any combination of attributes: undefined and unreferenced, defined and unreferenced (declared but never used), undefined and referenced (in the case of a forward or external reference), or defined and referenced.

Binary Operators

Operator	Description
+ - * /	The usual arithmetic operators
%	Modulo
& | ^	Bitwise AND, OR and XOR
<< >>	Bitwise shift left and shift right
< <= >= >	Boolean magnitude comparisons
=	Boolean equality
<> !=	Boolean inequality

- All binary operators have the same precedence: expressions are evaluated strictly left to right.
- Division or modulo by zero yields an assembly error.
- The "<>" and "!=" operators are synonyms.
- Note that the modulo operator (%) is also used to introduce binary constants (see "**Constants**"). A percent sign should be followed by at least one space if it is meant to be a modulo operator, and is followed by a `0' or ``1".

Special Forms

Special Form	Description
^^date	The current system date (GEMDOS format)
^^time	The current system time (GEMDOS format)

- The "^^date" special form expands to the current system date, in GEMDOS format (for legacy compatibility purposes). The format is a 16-bit word with bits (0..4) indicating the day of the month (1 - 31), bits (5..8) indicating the month (1 - 12) and bits (9..15) indicating the year since 1980, in the range (0 – 119).
- The "^^time" special form expands to the current system time, in GEMDOS format (for legacy compatibility purposes). The format is a 16-bit word with bits (0..4) indicating the current second divided by 2, bits (5..10) indicating the current minute (0 - 59) and bits (11..15) indicating the current hour (0 - 23).

Example Expressions

```
line  address     contents          source code
---- -------      --------          --------------------
1     00000000    4480              lab1: neg.l d0
2     00000002    427900000000      lab2: clr.w lab1
3                 =00000064         equ1  = 100
4                 =00000096         equ2  = equ1 + 50
5     00000008    00000064                dc.l lab1 + equ1
6     0000000C    7FFFFFE6                dc.l (equ1 + ~equ2) >> 1
7     00000010    0001                    dc.w ^^defined equ1
8     00000012    0000                    dc.w ^^referenced lab2
9     00000014    00000002                dc.l lab2
10    00000018    0001                    dc.w ^^referenced lab2
11    0000001A    0001                    dc.w lab1 = (lab2 - 6)
```

Lines 1 through four here are used to set up the rest of the example. Line 5 deposits a relocatable pointer to the location 100 bytes beyond the label `lab1'. Line 6 is a nonsensical expression that uses the `~' and right-shift operators. Line 7 deposits a word of 1 because the symbol `equ1' is defined (in line 3). Line 8 deposits a word of 0 because the symbol `lab2', defined in line 2, has not been referenced. But the expression in line 9 references the symbol `lab2', so line 10 (which is a copy of line 8) deposits a word of 1. Finally, line 11 deposits a word of 1 because the boolean equality operator evaluates to true.

The operators **^^defined** and **^^referenced** are particularly useful in conditional assembly. For instance, it is possible to automatically include debugging code if the debugging code is referenced, as in:

```
            lea         string,a0  ; A0 -> message
            jsr         debug      ; print a message
            rts                    ; and return
string:     dc.b        "Help me, Spock!",0
            .
            .
            .
            .iif ^^defined debug, .include "debug.s"
```

The **jsr** statement references the symbol ***debug***. Near the end of the source file, the **.iif** statement includes the file "debug.s" if the symbol ***debug*** was referenced. In production code, presumably all references to the ***debug*** symbol will be removed, and the debug source file will not be included. (We could have as easily made the symbol ***debug*** external, instead of including another source file).

Directives

Assembler directives may be a mix of upper or lower case. Directives may be preceded by a label; the label is defined before the directive is executed. Some directives accept size suffixes (.b, .w or .l); the default is word (.w) if no size is specified.

Switch	Description
.68000	Switch to 68000 assembly mode. This directive must be used within the TEXT or DATA segments. Instructions for the Jaguar RISC processors may not be assembled while in 68000 assembly mode.

.assert *expression* [*expression*]	Assert that the conditions are true (non-zero). If any of the comma-separated expressions evaluates to zero an assembler warning is issued. For example:
	`.assert *-start = $76`
	`.assert stacksize >= $400`

.bss **.data** **.text**	Switch to the BSS, Data or TEXT segments. The TEXT segment typically contains your executable program code. The DATA segment typically contains pre-initialized data (strings, tables, etc.). The BSS segment is used for un-initialized data storage. Instructions and data may not be assembled into the BSS segment, but symbols may be defined and storage may be reserved with the **.ds** directive. Each assembly starts out in the TEXT segment.

.ccdef *expression*	Allows you to define names for the condition codes used by the JUMP and JR instructions for GPU and DSP code. For example:
	`Always .ccdef 0`
	`. . .`
	`jump Always,(r3) ; 'Always' is actually 0`

.ccundef *regname*	Undefines a register name (regname) previously assigned using the .CCDEF directive. This is only implemented in GPU and DSP code sections.
.dc.i *expression*	This directive generates long data values and is similar to the DC.L directive, except the high and low words are swapped. This is provided for use with the GPU/DSP MOVEI instruction.
.dc[.size] *expression* *[,expression]*	Deposit initialized storage in the current section. If the specified size is word or long, the assembler will execute a `.even' before depositing data. If the size is `.b', then strings that are not part of arithmetic expressions (e.g. strings along) are deposited byte-by-byte. If no size is specified, the default is `.w'. This directive cannot be used in the BSS section.

.dcb[.size] *expression1 [,expression2, …]*	Generate an initialized block of `expression1' bytes, words or longwords of the value `expression2'. If the specified size is word or long, the assembler will execute `.even' before generating data.  If no size is specified, the default is `.w'. This directive cannot be used in the BSS section.
.dphrase	Align the program counter to the next integral double phrase boundary (16 bytes). Note that GPU/DSP code sections are not contained in their own segments and are actually part of the TEXT or DATA segments. Therefore, to align GPU/DSP code, align the current section before and after the GPU/DSP code.
.ds[.size] *expression*	Reserve space in the current segment for the appropriate number of bytes, words or longwords. If no size is specified, the default size is `.w'.  If the size is word or long, the assembler will execute `.even' before reserving space. This directive can only be used in the BSS section (in TEXT or DATA, use `.dcb' to reserve large chunks of initialized storage.)
.dsp	Switch to Jaguar DSP assembly mode. This directive must be used within the TEXT or DATA segments.
.end	End the assembly of the current file. In an include file, ends the include file and resumes assembling the superior file. This statement is not required, nor are warning messages generated if it is missing at the end of a file. This directive may be used inside conditional assembly, macro's or rept blocks.

.equr *expression*	Allows you to name a register. This is only implemented for GPU/DSP code sections. For example:
	
	`Clipw .equr r19`
	`. . .`
	`add ClipW,r0 ; ClipW actually is r19`

.equrundef *regname*	Undefines a register name previously assigned using the .EQUR directive. This is only implemented in GPU/DSP code sections.
.even	If the location counter for the current section is odd, make it even by adding one to it. In TEXT and DATA sections a zero byte is deposited if necessary. See also the directives .LONG, .PHRASE, .DPHRASE and .QPHRASE.

.globl *symbol [,symbol, …]* **.extern** *symbol [,symbol, …]*	Each symbol specified is made global. None of the symbols may be confined symbols (those starting with a period). If the symbol is defined in the assembly, the symbol is exported in the object file. If the symbol is undefined at the end of the assembly, and it was referenced (i.e. used in an expression), then the symbol value is imported as an external reference that must be resolved by the linker. The `.extern' directive is merely a synonym for `.globl'.
.gpu **.gpumain**	Switch to Jaguar GPU assembly mode. This directive must be used within the TEXT or DATA segments. The ".gpumain" directive processes the GPU source code into a state that can be run in main RAM instead of local GPU memory.
.if *expression* **.else** **.endif**	Start a block of conditional assembly. If the expression is true (non-zero) then assemble the statements between the `.if' and the matching `.endif' or `.else'. If the expression is false, ignore the statements unless a matching `.else' is encountered. Conditional assembly may be nested to any depth. It is possible to exit a conditional assembly block early from within an include file (with `.end') or a macro (with `.endm').
.iif *expression, statement*	Immediate version of `.if'. If the expression is true (non-zero) then the statement, which may be an instruction, a directive or a macro (or even another `.iif'), is executed. If the expression is false, the statement is ignored. No `.endiif' is required. For example:  `.iif age < 18, canDrive = 0` `.iif weight > 500, dangerFlag = 1` `.iif !(^^defined DEBUG), .include dbsrc`

.incbin *filename*	Include a binary file in your source at the present position. The syntax is the same as the .INCLUDE directive. The data in the binary file is included verbatim in the output file. For example:
	
	`picture_dat::`
	`.incbin "picture.dat"`
	
	Will include the data within the file PICTURE.DAT at the position following the "picture_dat" label.

.list **.nlist**	Enable or disable source code listing.
.long	Align the program counter to the next integral long boundary (4 bytes). Note that GPU/DSP code sections are not contained in their own segments and are actually part of the TEXT or DATA segments. Therefore, to align GPU/DSP code, align the current section before and after the GPU/DSP code.

.macro *name [formal, formal, …]* **.endm** **.exitm**	Define a macro called `name' with the specified formal arguments. The macro definition is terminated with a `.endm' statement. A macro may be exited early with the `.exitm' directive.  See the chapter on `Macros' for more information.
.macundef *macroName [,macroName, …]*	Remove the macro-definition for the specified macro names. If reference is made to a macro that is not defined, no error message is printed and the name is ignored.
.nolist	Turns off assembly listing output. This is basically the same as the .NLIST directive and is added for compatibility.
.org *expression*	Define the origin address used for code generation. It sets the value of the location counter (or PC) to the value specified by expression, which must be defined and absolute. The .ORG directive is intended for Jaguar GPU and DSP code. It is not legal in 68000 sections. All symbols generated following this directive will be non-relocatable.
.phrase	Align the program counter to the next integral phrase boundary (8 bytes). Note that GPU/DSP code sections are not contained in their own segments and are actually part of the TEXT or DATA segments. Therefore, to align GPU/DSP code, align the current section before and after the GPU/DSP code.

.print *expression*	The .PRINT directive is similar to the standard 'C' library *printf()* function and is used to print user messages from the assembly process. You can print any string or valid expression. Several format flags that can be used to format your output are also supported.
	
	`/x hexadecimal`
	`/d signed decimal`
	`/u unsigned decimal`
	`/w word`
	`/l long`
	
	For example:
	`MASK .equ $FFF8`
	`VALUE .equ -100000`
	`.print "Mask: $",/x/w MASK`
	`.print "Value: ",/d/l VALUE`

.qphrase	Align the program counter to the next integral quad phrase boundary (32 bytes). Note that GPU/DSP code sections are not contained in their own segments and are actually part of the TEXT or DATA segments. Therefore, to align GPU/DSP code, align the current section before and after the GPU/DSP code.
.regequ *expression*	Essentially the same as .EQUR. Included for compatibility with the GASM assembler.
.regundef	Essentially the same as .EQURUNDEF. Included for compatibility with the GASM assembler.

.rept *expression* **.endr**	The statements between the .REPT and .ENDR directives will be repeated *expression* times. If the expression is zero or negative, no statements will be assembled. No label may appear on a line containing either of these directives.

Macros

A macro definition is a series of statements of the form:

```
.macro name [ formal-arg, ...]
:
: statements making up the macro body
:
.endm
```

The name of the macro may be any valid symbol that is not also a 68000 instruction or an assembler directive. (The name may begin with a period, but macros cannot be made confined the way labels or equated symbols can be). The formal argument list is optional; it is specified with a comma-seperated list of valid symbol names. Note that there is no comma between the name of
the macro and the name of the first formal argument.

A macro body begins on the line after the `.macro' directive. All instructions and directives, except other macro definitions, are legal inside the body.

The macro ends with the `.endm' statement. If a label appears on the line with this directive, the label is ignored and a warning is generated.

Parameter Substitution

Within the body, formal parameters may be expanded with the special forms:

\name
\{name}

The second form (enclosed in braces) can be used in situations where the characters following the formal parameter name are valid symbol continuation characters. This is usually used to force concatentation, as in:

\{frog}star
\{godzilla}vs\{reagan}

The formal parameter name is terminated with a character that is not valid in a symbol (e.g. whitespace or punctuation); optionally, the name may be enclosed in curly-braces. The names must be symbols appearing on the formal argument list, or a single decimal digit (\1 corresponds to the first argument, \2 to the second, \9 to the ninth, and \0 to the tenth). It is possible for a macro to have more than ten formal arguments, but arguments 11 and on must be referenced by name, not by number.

Other special forms are:

Special Form	Description
****	A single '\'
\~	A unique label of the form "Mn"
\#	The number of arguments actually specified
\!	The 'dot-size' specified on the macro invocation
\?name	Conditional expansion
\?{name}	Conditional expansion

The last two forms are identical: if the argument is specified and is non-empty, the form expands to a `1', otherwise (if the argument is missing or empty) the form expands to a `0'.

The form `\!' expands to the `dot-size' that was specified when the macro was invoked. This can be used to write macros that behave differently depending on the size suffix they are given, as in this macro which provides a synonym for the `dc' directive:

```
.macro     deposit    value
           dc\!       \value
.endm
deposit.b  1          ; byte of 1
deposit.w  2          ; word of 2
deposit.l  3          ; longword of 3
```

Macro Invocation

A previously-defined macro is called when its name appears in the operation field of a statement. Arguments may be specified following the macro name; each argument is seperated by a comma. Arguments may be empty. Arguments are stored for substitution in the macro body in the following manner:

- Numbers are converted to hexadecimal (they also acquire a leading `$').
- All spaces outside strings are removed.
- Keywords (such as register names, dot sizes and `^^' operators) are converted to lowercase.
- Strings are enclosed in double-quote marks (").

For example, a hypothetical call to the macro `mymacro', of the form:

```
mymacro A0, , 'Zorch' / 32, ^^DEFINED foo, , , tick tock
```

will result in the translations:

Argument	Expansion	Comment
\1	a0	"a0" converted to lower case
\2		Empty
\3	"Zorch"/$20	'Zorch' in double-quotes, 32 in hex
\4	^^defined foo	"^^DEFINED" converted to lower case
\5		Empty
\6		Empty
\7	ticktock	Spaces removed (note concatenation)

The `.exitm' directive will cause an immediate exit from a macro body. Thus the macro definition:

```
.macro foo source
          .iif !\?source, .exitm     ;  exit  if  source  is
     empty
          move source,d0             ;   otherwise,   deposit
     source
.endm
```

will not generate the move instruction if the argument `source' is missing from the macro invocation.

The `.end', `.endif' and `.exitm' directives all popout of their include levels appropriately. That is, if a macro performs a `.include' to include a source file, an executed `.exitm' directive within the include-file will pop out of both the include-file and the macro.

Macros may be recursive or mutually recursive to any level, subject only to the availability of memory. When writing recursive macros, take care in the coding of the termination condition(s). A macro that repeatedly calls itself will cause the assembler to exhaust its memory and abort the assembly.

Macro Invocation

The `Gemdos' macro is used to make file system calls. It has two parameters, a function number and the number of bytes to clean off the stack after the call. The macro pushes the function number onto the stack and does the trap to the file system. After the trap returns, conditional assembly is used to choose an ADDQ or an ADD.W to remove the arguments that were pushed.

```
.macro Gemdos trpno, clean
          move.w #\trpno,-(sp)  ; push trap number
          trap !1               ; do GEMDOS trap
          .if \clean <= 8       ;
          addq #\clean,sp       ; clean-up up to 8 bytes
          .else                 ;
          add.w #\clean,sp      ; clean-up more than 8 bytes
          .endif                ;
.endm
```

The `Fopen' macro is supplied two arguments; the address of a filename, and the open mode. Note that plain MOVE instructions are used, and that the caller of the macro must supply an appropriate addressing mode (e.g. immediate) for each argument.

```
.macro Fopen file, mode
     move.w #\mode,-(sp)         ; push open mode
     move.l #\file,-(sp)         ; push address of file name
     Gemdos $3d,8                ; do the GEMDOS call
.endm
```

The `String' macro is used to allocate storage for a string, and to place the string's address somewhere. The first argument should be a string or other expression acceptable in a `dc.b' directive. The second argument is optional; it specifies where the address of the string should be placed. If the second argument is omitted, the string's address is pushed onto the stack. The string data itself is kept in the data segment.

```
.macro     String str,loc
           .if \?loc             ; if loc is defined
           move.l #.\~,\loc      ; put string's address there
           .else                 ; otherwise
           pea #.\~              ; push the string's address
           .endif                ;
           .data                 ; put the string data
.\~:       dc.b r,0              ; in the data segment
           .text                 ; switch back to TEXT
.endm
```

The construction `.\~' will expand to a label of the form `.Mn' (where `n' is a unique number for every macro invocation), which is used to tag the location of the string. The label should be confined because the macro may be used along with other confined symbols.

Unique symbol generation plays an important part in the art of writing fine macros. For instance, if we needed three unique symbols, we might write `.\~a' `.\~b' and `.\~c'.

Repeat Blocks

Repeat-blocks provide a simple iteration capability. A repeat block allows a range of statements to be repeated a specified number of times. For instance, to generate a table consisting of the numbers 255 through 0 (counting backwards) you could write:

```
.count      set         255         ; initialize counter
            .rept       256         ; repeat 256 times:
            dc.b        .count      ; deposit counter
.count      set         .count - 1  ; and decrement it
            .endr                   ; (end of repeat block)
```

Repeat blocks can also be used to duplicate identical pieces of code (which are common in bitmap-graphics routines). For example:

```
            .rept       16          ; clear 16 words
            clr.w       (a0)+       ; starting at A0
            .endr                   ;
```

68000 Mode

All of the standard Motorola 68000 mnemonics and addressing modes are supported; you should refer to Motorola's `68000 PROGRAMMER'S REFERENCE MANUAL' for a description of the instruction set and the allowable addressing modes for each instruction. The assembler performs all the reasonable optimizations of instructions to their short or address register forms.

Register names may be in upper or lower case. All register names are keywords, and may not be used as labels or symbols. None of the 68010 or 68020 register names are keywords (but they may become keywords in the future).

Addressing Modes

Assembler Syntax	Description
Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect postincrement
-(An)	Address register indirect predecrement
disp(An)	Address register indirect with displacement
bdisp(An,Xi[.size])	Address register indirect indexed
abs.w	Absolute short
abs	Absolute (long or short)
abs.l	Forced absolute long
disp(PC)	Program counter with displacement
bdisp(PC,Xi)	Program counter indexed
#imm	Immediate

Branches

To adhere to the historic aspects of MADMAC being a one pass assembler, forward branches will not be automatically optimized to their short form. Instead, unsized forward branches are assumed to be long. Backward branches are always optimized to the short form if possible. SMAC is a two pass assembler and future optimizations of forward branch calls are planned. A table that lists `extra' branch mnemonics (common synonyms for the Motorola defined mnemonics) appears below.

Alternate Name	Becomes
bhs	bcc
blo	bcs
bze,bz	beq
bnz	bne
dblo	dbcs
dbze	dbeq
dbra	dbf
dbhs	dbhi
dbnz	dbne

Jaguar GPU/DSP Mode

SMAC will generate code for the Atari jaguar GPU and DSP custom RISC (Reduced Instruction Set Computer) processors. See the Atari Jaguar Software reference Manual – Tom & Jerry for a complete listing of Jaguar GPU and DSP assembler mnemonics and addressing modes.

Condition Codes

The following condition codes for the GPU/DSP JUMP and JR instructions are built-in:

CC (Carry Clear)	= %00100
CS (Carry Set)	= %01000
EQ (Equal)	= %00010
MI (Minus)	= %11000
NE (Not Equal)	= %00001
PL (Plus)	= %10100
HI (Higher)	= %00101
T (True)	= %00000

Optimizations and Translations

The assembler provides "creature comforts" when it processes GPU/DSP mnemonics:

- In GPU/DSP code sections, you can use JUMP (Rx) in place of JUMP T, (Rx) and JR (Rx) in place of JR T,(Rx).
- SMAC tests all GPU/DSP restrictions and corrects them wherever possible (such as inserting a NOP instruction when needed).
- The "(Rx+N)" addressing mode for GPU/DSP instructions is optimized to "(Rx)" when "N" is zero.

Error Messages

Most of SMAC's error messages are self-explanatory. They fall into four classes: warnings about situations that you (or the assembler) may not be happy about, errors that cause the assembler to not generate object files, fatal errors that cause the assembler to abort immediately, and internal errors that should never happen.

You can write editor macros (or sed or awk scripts) to parse the error messages SMAC generates. When a message is printed, it is of the form:

```
filename[line number]: type: message
```

The first element, a filename, indicates the file that generated the error. The filename is followed by the line number in the file (enclosed in square brackets), then a colon followed by a space. The type of error message (Error, Warning, Fatal etc) is displayed followed by a colon and space and then finally the error message itself.