# Madmac Macro Assembler

## Introduction

This document describes MADMAC, a fast macro assembler that generates code for the Motorola 68000, Atari Jaguar GPU/DSP, and 6502 processors. It was originally written at Atari Corporation by programmers who needed a high performance assembler for their work. Madmac was originally distributed as part of the Atari ST Computer Developer's Kit, and has been updated to support the requirements of the development system for the Atari Jaguar console.

Madmac is intended to be used by programmers who write mostly in assembly language. It was not originally intended to be a back-end to a C compiler. Therefore it has creature comforts that are usually neglected in such back-end assemblers. It supports include files, macros, local symbols, some limited control structures, and other features. Madmac is also blindingly fast[1], a feature often sadly and obviously missing in today's assemblers.

## The Command Line

The assembler is called MAC.EXE (for the PC/MSDOS version) or MAC.TTP (for the Atari/TOS version). The command line takes the form of:

```
mac [switches] [files ...]
```

A command line consists of any number of switches followed by the names of files to be assembled. A switch is specified by a dash ("-") followed immediately by a key character. Some switches accept or require arguments to immediately follow the key character, with no spaces in between. Key characters are not case-sensitive, so "**-d**" and "**-D**" 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 names of the input files.

If the command line is empty, the Madmac prints a copyright message and enters an interactive mode, prompting for successive command lines with an asterisk ("*") character. Hitting {Enter} on an empty command line will cause Madmac to exit. After each assembly in interactive mode, Madmac will print a summary of the memory usage, the number of lines processed, and the amount of time the assembly took.

Input files are assumed to have the extension ".S" and Madmac will look for a file with this extension if none is specified. Different extensions may be used if they are specified on the command line. More than once source file can be specified. The files are assembled into one object file as if they were concatenated.

[1] The PC/MSDOS version of Madmac has been benchmarked at over 240,000 lines per minute on a DX2/66-based PC. Of course, your mileage may vary.

Madmac normally produces object code files with the same filename as the input source file, except with a ".O" extension. If multiple files are specified, the name of the first file is used. If the first input filename is a device (like CON:), then the output filename will be NONAME.O. 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. Please note that some switches may not be applicable to Jaguar programming. They are listed for completeness.

| Switch | Description |
|---|---|
| **-?** | Print Madmac usage information. |
| **-6** | The **-6** switch causes Madmac to act as a back end assembler for the Alcyon C compiler. However, this mode is not 100% compatible with the AS68 assembler (which is the normal Alcyon C back-end assembler).<br><br>Symbols beginning with a capital "L" are not included in the object file. (These are special symbols used by the Alcyon C compiler.)<br><br>This is generally not applicable to Jaguar programming unless you're using the Alcyon C compiler on an Atari computer to generate 68000 code. |
| **-a[s]** *text, data, bss* | Output DRI-format absolute executable file (.ABS). Using **-as** instead of **-a** adds symbols to the output file.<br><br>*text* = Address for TEXT segment<br>*data* = Address for DATA segment<br>*bss* = Address for BSS segment<br><br>Values for *text*, *data*, and *bss* can be:<br>a hexadecimal value to be used as the address.<br>**r**: relocatable segment (not useful for Jaguar programs)<br>**x**: contiguous segment (contiguous with previous segment)<br><br>For example "-a 802000 x 4000" would put the TEXT segment at $802000, the DATA segment immediately after that, and the BSS section at $4000. |

 

| | |
|---|---|
| **-c** *cpu* | Start out in a DSP or GPU section instead of 68000, and output .BIN/.SYM files: *cpu* is either "dsp" or "gpu":<br><br>dsp: Jerry's DSP output code (i.e. **-cdsp**)<br>gpu: Tom's GPU output code (i.e. **-cgpu**)<br><br>External variables cannot be referenced in files assembled with these options, because BIN files contain only raw binary code with an 8-byte header:<br><br>typedef struct {<br>long exec_addr; /* values are in big-endian */<br>long code_size; /* (Motorola) format */<br>} BIN_Header;<br><br>You can use the **-fb** option to output BSD symbols and the **-g** option to output source-level debugging information in the .SYM file. Note that the use of .BIN and .SYM files is mostly for backwards compatibility with code originally written for the GASM assembler, and is not recommended for new code. |
| **-d** *symbol*[=*value*] | This switch permits symbols to be defined on the command line. The name of the symbol to be defined must immediately follow the switch (no spaces). The symbol name may optionally be followed by an 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. The symbol's attributes are "defined, not referenced, and absolute". This switch is most useful for enabling conditionally assembled debugging code or test code on the command line. For example:<br><br>`-dDEBUG -dLoopCount=999 -dDebugLevel=55`<br><br>This would define "DEBUG" and give it a value of zero, "LoopCount" with a value of 999, and "DebugLevel" with a value of 55. |
| **-e**[*errorfile*] | This switch causes Madmac 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 of ".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).<br><br>If no errors are encountered, then no error message file will be created. However, note that if an assembly produces no errors, then any error file from a previous assembly will not be deleted. |

| | |
|---|---|
| **-f**[*format*] | Select object file format to be output:<br><br>**-fa**: DRI (default output)<br>Symbols are limited to 8 characters length.<br>Source-level debugging information cannot be included.<br>No support for proper relocation of MOVEI GPU/DSP instruction.<br><br>**-fb**: BSD (Recommended format for Jaguar programming)<br>Symbol lengths are unlimited.<br>Source-level debugging information can be included.<br>Supports proper relocation for MOVEI GPU/DSP instruction.<br><br>**-fm**: Mark Williams (not applicable to Jaguar programming)<br>Symbols are limited to 8 characters.<br>Source-level debugging information cannot be included.<br>No support for proper relocation of MOVEI GPU/DSP instruction.<br><br>**-fmu**: Mark Williams, except moves leading underscore characters on symbols to be moved to the end of the symbol name (i.e. "***_main***" becomes "***main_***" and "***__main***" becomes "***_main_***"). |
| **-g** | Output source level debugging information (only when using **-fb** switch to select BSD format object file output). |
| **-i**[*path*] | The **-i** switch allows automatic directory searching for include files. A list of semi-colon separated directory search paths may be listed immediately following the switch (with no spaces anywhere). For example:<br><br>`-im:;c:\include;c:\include\sys`<br><br>will cause Madmac to search the current directory of drive M, and the directories INCLUDE and INCLUDE\SYS on drive C.<br><br>If the "-i" switch is not specified, Madmac searches for the MACPATH environment variable, which is used to specify include file directories in the same way. For example:<br><br>`set MACPATH=m:;c:\include;c:\include\sys`<br><br>will cause Madmac to search the same directories as the previous example. (Some command line interpreters may use "setenv" instead of "set" to set an environment variable instead of a shell variable.)<br><br>It is recommended that you set the MACPATH environment variable to point at your global include files, and use the **-i** option only to override or add to the paths specifed by MACPATH.<br><br>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 specifed by MACPATH, you can do something like this:<br><br>`-iproject\inc;$(MACPATH)`<br><br>And the $(MACPATH) macro will be expanded by your MAKE utility into the contents of the MACPATH environment variable. This is a standard feature of of nearly all MAKE utilities. |

 

| | |
|---|---|
| **-l**[*filename*] | The **-l** switch causes Madmac to generate an assembly listing file. If *filename* immediately follows the switch, the listing is written to the specified file.<br><br>If no filename is specified, 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 filename is FILE or FILE.S). |
| **-o** *file* | The **-o** switch causes Madmac to write its object code output to the specified file. No default extension is applied to the filename, so you need to specify whatever extension is appropriate. Unlike most other Madmac command line switches, a space between the switch and the filename is permitted (but not required). For example:<br><br>`-ojagmand.o`<br><br>will produce an object file named JAGMAND.O, regardless of what the source file was named. |
| **-p**<br>**-ps** | The **-p** and **-ps** switches cause Madmac to produce a GEMDOS format executable program file (with the default extension of ".PRG" unless otherwise specified by the **-o** switch).<br><br>If there are any unresolved external references at the end of the assembly, an error message is emitted and no executable file is created.<br><br>The **-ps** switch adds symbols (Alcyon format) to the output file.<br><br>This switch is not applicable to Jaguar programming. |
| **-q** | The **-q** switch was used originally on the Atari to install Madmac as a memory-resident program. This was intended to reduce load times for multiple calls to Madmac on floppy-disk based systems.<br><br>This switch is not available in the PC/MS-DOS version of Madmac. |
| **-r**[*size*] | The **-r** switch causes Madmac 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:<br><br>**-rw** word (2 bytes, default alignment)<br>**-rl** long (4 bytes)<br>**-rp** phrase (8 bytes)<br>**-rd** double phrase (16 bytes)<br>**-rq** quad phrase (32 bytes)<br><br>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** | The **-s** switch causes Madmac to generate warning messages about possible unoptimized forward short branches in 68000 code. This is used to point out branches that could have been short (e.g. "bra" could be "bra.s"). |
| **-u** | The **-u** switch causes Madmac to force all referenced and undefined symbols to be global, as though they had been explicitly specified with the **.extern** or **.globl** directives, or defined using a double-colon. (See **Symbols and Scope** for more information.)<br><br>This switch can be used as a short cut when you have a large number of external symbols, and don't want to use individual **.extern** or **.globl** directives to declare each one. |

| | |
|---|---|
| **-v** | Set verbose mode. This will cause Madmac to print out the names of each source file and include file as they are processed. Verbose mode is automatically entered when Madmac is called with no command line and prompts for your input. |
| **-y**[*pagelen*] | The **-y** switch, followed immediately by a decimal number (with no intervening spaces), sets the number of lines in a page for the assembly listing (if a listing is requested with the **-l** switch).<br><br>For example, **-y90** would set the number of lines per page to 90.<br><br>If the number of lines is missing, or less than 10, an error message is generated. |

## Using Madmac

Let's assemble some sample files. Load your favorite text editor and create a small text file 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 DOS command line, type the following command:

```
mac test.s
```

Assuming your system is setup correctly, this will call Madmac, which will assemble TEST.S and produce an object module file named TEST.O. If you see an error message telling you that Madmac cannot find the "JAGUAR.INC" file, then chances are you do not have your MACPATH environment variable set correctly. See the **Getting Started** section of your Jaguar Developer Documentation for information on how to set your environment variables.

So now we have an object module, which isn't of much use by itself until you run it through the linker, probably with other object modules, to create an executable program. But if you have been reading carefully, then you know that Madmac can generate an executable program file without requiring an external linker. This is useful for making small stand-alone programs that don't require external references or library routines. For example, the following two commands:

```
mac test.s
aln -e -a 802000 x 4000 -o test.cof test.o
```

could be replaced by the single command:

```
mac -a 802000 x 4000 -o test.cof test.s
```

 

To a certain degree, this can also be used to assemble multiple files at once, but it's probably easier in most cases to take advantage of the linker at that point. Now let's 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 DOS command, type the following command:

```
mac -ltest2.lst -y95 -o test2.cof -as 802000 x 4000 -Dcolor1=1 test2.s
```

This produces an assembly listing file named TEST2.LST with 95 lines per page, writes an executable program file (with symbols) to a file named TEST2.COF, and defines the symbol "***color1***" to have a value of 1 when the TEST2.S file is assembled.

Download and run the program we just created to the Jaguar using the command line:

```
rdbjag test2.cof -g -q
```

You'll see that all this program does is change the background color of the Jaguar screen by writing a value to the **BG** register. Depending on how *color1* is defined, you will different colors.

## Interactive Mode

If you invoke Madmac with an empty command line, it will print a copyright message and prompt you for more commands with an asterisk character (*). This is useful if you want to assemble several files in succession without reloading the assembler for each assembly.

In interactive mode, the assembler is also in verbose mode, as if you had specified "-v" on each command line:

```
E:\JAGUAR\SRC\JAGMAND>mac
--------------------------------
 MADMAC  Atari Macro Assembler
 Copyright 1987-94 Atari Corp.
V3.03 Aug 20 1994
--------------------------------
* -fb -g jagmand.s
 [Including: jagmand.s]
[Including: jaguar.inc]
[Leaving: jaguar.inc]
[Including: cry.pal]
[Leaving: cry.pal]
[Leaving: jagmand.s]
[Writing BSD object file: jagmand.o]
33K used, 367 lines
*
```

You can see that Madmac gave a "blow-by-blow" account of the files it processed, as well as a summary of the assembler's memory usage, and the number of lines processed (including macro and repeat-block expansion as appropriate).

After the assembly is finished, Madmac prompts for another command line with the asterisk. At this point, you can either type in a new command line to be processed, or you can exit Madmac by hitting {Enter} on an empty line.

## Things You Should Be Aware Of

Madmac is a one pass assembler. This means that it gets all of the work done by reading each source file exactly one time, and then "back-patching" to fix up forward references. This one-pass nature is usually transparent to the programmer, with the following important exceptions:

- Error messages may appear at the end of the assembly, referring to earlier source lines that contained undefined symbols.

- All object code generated must fit in memory. Running out of memory is a fatal error that you must deal with by splitting up your source code files, resizing them, or by increasing your available memory.[2]

- Forward branches (including BSR instructions) are never optimized to their short forms (because this would change the length of the code which has already been generated). To get a short forward branch, it is necessary to explicitly use the ".s" suffix in the source code.

[2] The PC/MSDOS version of Madmac is a DOS Protected Mode Interface program and is not subject to the 640K memory limitations of MS-DOS versions 6.22 and earlier.

 

## Forward Branches

Madmac does not automatically optimize forward branches for you, but it will tell you about them if you use the "-s" switch on the command line:

```
E:\JAGUAR\SRC\JAGMAND>mac -s example.s
"example.s", line 20: warning: unoptimized short branch
```

With the "-e" switch, you can redirect the error & warning output to a file, and determine by hand (or using editor macros) which forward branches are save to explicitly declare as short.

## Text File Format

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

- Files must contain characters with ASCII values less than 128. Characters with ASCII values above 127 must be contained in strings (i.e. between single or double quotes) or in comments.
- Lines of text are terminated by carriage return/linefeed, linefeed-only, or carriage return only. (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 semi-colon (;) then the entire line is a comment. A semi-colon anywhere on the line (except in a string) begins a comment field which extens to the end of the line.

The label, if it appears, must be terminated with one or two colons. If it is terminated with a double colon, it is automatically declared as a global. It is illegal to declare a confined symbol as global (see **Symbols and Scope**).

## 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 the value of an expression, which must be defined (no forward or external references). The third form, with two equals signs, 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 at different positions within the same file, like a variable. The last form equates the symbol to a 16-bit register mask specifed by a register list.

It is possible to equate confined symbols. For example:

```
cr          equ         13              ; carriage return
lf          =           10              ; linefeed
DEBUG       ==          1               ; global debug flag
count       set         0               ; variable
count       set         count+1         ; increment the variable
.regs       reg         d3-d7/a3-a6     ; register list
.cr         =           13              ; confined (local) equate
```

## 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 valid 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 instruction mnemonics, assembler directives, or register names.

Symbols are limited to 100 characters in length, but may be truncated to 8 characters if the DRI object module format is selected, or 16 characters if the Mark Williams object module format is selected. No warning or error message is given in the event of a conflict created by symbol names being truncated. If BSD object module output is selected, the entire symbol, up to 100 characters, is used.

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

```
reallyLongSymbolName                .dc move
.reallyLongConfinedSymbolName       .move
a10                                 frog
.a10                                .frog
ret                                 aa6
.ret                                .a9
dc                                  a9
```

 

```
.9                              .
????                            ._
.????                           _fog
.0                              ?zippo?
.00                             sys$system
.000                            atari
.1                              Atari
.11                             ATARI
.111                            aTaRi
```

While all of the following symbols are illegal:

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

Symbols beginning with a period (.) are confined; their scope is limited to the space between two normal (unconfined) labels. Confined symbols may be either 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 symbols 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 as they allow the programmer to be much less inventive about finding small, unique names that also have meaning.

It is legal to define symbols that have the same name as processor mnemonics (such as "move"or "rts") or assembler directives. However, one should be careful when doing so 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 wasn't 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 macro names):

```
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 ar 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) or 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 from one to four 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. MADMAC will generate a warning if a backslash is followed by a character not appearing below:

| | | |
|---|---|---|
| \\ | $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 MS-DOS and GEMDOS path names, and that you may have to escape backslash characters in your source code. For example, to get the filename "C:\AUTO\AHDI.S" you would specify the string "C:\\AUTO\\AHDI .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 separated by slashes. A register range consists of two register names, R*m* and R*n*, *m* < *n*, separated by a dash. For example:

*Note: older versions of Madmac supported the use of register names* R0, R1, ... R15 *as register names. This is no longer supported because these are now reserved as Jaguar GPU & DSP register names.*

```
Resister 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 resister 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           d3-47/d3-a0  ; 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" registers
```

# 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 when the program is relocated (for instance, the number 0, all labels declared in an ABS 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 1355), 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) attempt 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-I    *, *
```

Similarly, the pseudo-symbol "$" has the value of the current section's location counter, 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 (l'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 I if the expression is true, and 6 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 (i.e. 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. |
| & I ^ | Bit-wise AND, OR and Exdusive Or. |
| « » | Bit-wise 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 '1=" 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. The format is a 16-bit word with bits ... .4 indicating the day of the month (1... 31), bits 5.. .8 indicating the month (I... 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. 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          labl:     neg.l     do
2        00000002    427900000000  lab2:     clr.w     labl
3                    =00000064     equl      =         100
4                    =00000096     equ2      =         equl + 50
5        00000008    00000064                dc.l      labl + equl
6        0000000c    7FFFFFE6                dc.l      (equl + -equ2) » 1
7        00000010    0001                    dc.w      ^^defined equl
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      labl = (lab2 - 6)
```

Lines 1 through 4 are used to set up the rest of the example. Line 5 deposits a relocatable pointer to the location 100 bytes beyond the label ***labl***. Line 6 is a nonsensical expression that uses the ~ and right-shift operators. Line 7 deposits a word of 1 because the symbol ***equl*** 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, you can 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      ; (the message)
            .
            .
            .
            .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.S debugging source code 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 any mix of upper- or lowercase. The leading periods are optional, though they are shown here and their use is encouraged. Directives may be preceeded by a label; the label is defined before the directive is executed. Some directives accept size suffixes (.b, .s, .w or .1); the default is word (.w) if no size is specified. The .s suffix is identical to .b.

| Directive | Description |
|---|---|
| **.6502** | Switch to 6502 assembly mode. The location counter is undefined, and must be set with the **.org** directive before any code can be generated.<br><br>Inside a 6502 segment, the **dc.w** directive will produce 6502-format words (little-endian, with low byte first).<br><br>The reserved keywords for other sections (d0-d7/a0-a7/ssp, usp, and so on) remain reserved (and thus unusable) while in the 6502 section.<br><br>The directives **globl**, **dc.l**, **dcb.l**, **text**, **data**, **bss**, **abs**, **even** and **comm** are illegal in the 6502 section.<br><br>It is permitted, though probably not useful, to generate both 6502 and 68000 code in the same object file.<br><br>*Please note that the 6502 assembly capabilities of MADMAC have not been tested since the addition of the Jaguar GPU and DSP assembly modes. It is quite possible that the 6502 capabilities are broken in current versions of MADMAC.* |
| **.68000** | Switch to 680x0 assembly mode. This directive **must** be used within the TEXT or DATA segments. Instructions for the 6502, Jaguar GPU, and Jaguar DSP may not be assembled while in 680x0 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:<br><br>`.assert *-start = $76`<br>`.assert stacksize >= $400` |

**.AUTOEVEN**

Enables automatic word alignment between directives and instructions. For example, if you do:

```
.DC.B     $12
.DC.L     $3456789A
```

and the address at the DC.L directive following the .DC.B directive is not word-aligned, then Madmac will pad with a zero byte before the DC.L directive. This results in $12 $00 $34 $56 $78 $9A being output. This is the default mode of operation.

**.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 uninitialized 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.

**.cargs** [*#expression*,] *symbol*[.*size*] [, *symbol*[.*size*]...]

Compute stack offsets to C (and other language) arguments. Each symbol is assigned an absolute value (like **equ**) which starts at *expression* and increases by the size of each symbol, for each symbol. If the expression is not supplied, the default starting value is 4. For example:

```
.cargs #8, .fileName.l,  openMode, .bufPointer.l
```

could be used to declare offsets from register A6 to a pointer to a filename, a word containing an open mode, and a pointer to a buffer. (Note that the symbols used here are confined). Another example, a C-style "string-length" function, could be written as:

```
 strlen::
     .cargs   .string              ; declare arg
     move.l   .string(sp),ao       ; a0 -> string
     moveq    #-1,dO               ; initial size  = -1
.l:
     addq.l   #1,d0                ; bump size
     tst.b    (aO)+                ; at end of string?
     bne      .1                   ; (no -- try again)
     rts                           ; return string length
```

**.CCDEF** *expression*

Allows you to define names for the condition codes used by the **JUMP** and **JR** instructions for GPU/DSP code. For example:

```
Always          .CCDEF     0
...
     jump       Always,(r3)            ; 'Always' is actually 0
```

**.CCUNDEF** *registername*

Undefines a register name previously assigned using the .CCDEF directive. This is only implemented for GPU/DSP code sections.

**.CLEAR**

After this directive, Madmac allows the use of the **CLR.L** instruction for the 680x0. The **CLR.L** instruction does not work properly on the Jaguar when accessing hardware register locations. The default state is .CLEAR.

**.comm** *symbol*, *expression*

Specifies a label and the size of a common region. The label is made global, thus confined symbols cannot be made common. The linker groups all common regions of the same name; the largest size determines the real size of the common region when the file is linked.

 

| | |
|---|---|
| **.DC.l** *expression* | This directive generates long data values and is similar to the **DC.L** directive, except the high word and low word 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 (**.w**) or long (**.b**), the assembler will execute an **.even** directive before depositing data. If the size is byte (**.b**), then strings that are not part of arithmetic expressions are deposited byte-by-byte.<br><br>If no size is specified, the default is **.w**.<br><br>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 an **.even** directive before generating data.<br><br>If no size is specified, the default is **.w**.<br><br>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 within 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 the size is word or long, the assembler will execute an **.even** directive before reserving space.<br><br>If no size is specified, the default size is **.w**.<br><br>This directive can only be used in the BSS or ABS sections (in TEXT or DATA, use **.dc.b** 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. All DSP instructions, as defined in the **Jaguar Software Reference Manual - Tom And Jerry**, may be assembled while in DSP assembly mode. |
| **.eject** | Issue a page eject in the listing file. |
| **.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, macros or **.rept** blocks. |
| **.EQUR** *expression* | Allows you to name a register. This is only implemented for GPU/DSP code sections. For example:<br><br>`ClipW .EQUR r19`<br>`...`<br>`add ClipW,r0 ; ClipW actually is r19` |
| **.EQURUNDEF** *registername* | Undefines a register name previously assigned using the .EQUR directive. This is only implemented for 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*...]<br><br>**.extern** *symbol* [,*symbol*...] | Each symbol specified is made global. 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.<br><br>None of the symbols may be confined symbols (those starting with a period).<br><br>The **.extern** directive is merely a synonym for **.globl**. |
| **.goto** *label* | This directive provides unstructured flow of control within a macro definition. It will transfer control to the line of the macro containing the specified goto label. A goto label is a symbol preceeded by a colon that appears in the first column of a source line within a macro definition;<br><br>`:label`<br><br>where the label itself can be any valid symbol name, followed immediately by whitespace and a valid source line (or end of line). The colon must appear in the first column.<br><br>The goto-label is removed from the source line prior to macro expansion - to all intents and purposes the label is invisible except to the **.goto** directive. Macro expansion does not take place within the label.<br><br>For example, here is a silly way to count from 1 to 10 without using **.rept**:<br><br>`.macro Count`<br>`count set 1`<br>`:loop dc.w count`<br>`count set count + 1`<br>`iif count <= 10, goto loop`<br>`.endm` |
| **.GPU** | Switch to Jaguar GPU assembly mode. This directive **must** be used within the TEXT or DATA segments. All GPU instructions, as defined in the **Jaguar Software Reference Manual - Tom And Jerry**, may be assembled while in GPU assembly mode. |
| **.if** *expression*<br>**.else**<br>**.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.<br><br>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, is executed. If the expression is false, the statement is ignored. No **.endiif** is required. For example:<br><br>`.iif age < 21, canDrink = 0`<br>`.iif weight > 500, dangerFlag = 1`<br>`.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. If no filename extension is specified, then .BIN is added automatically. The data in the binary file is included verbatim in the output file. For example:<br><br>`picture_dat::`<br>`  .INCBIN  "picture.dat"`<br><br>will include the data within the file PICTURE.DAT at the position following the *picture_dat* label.<br><br>Note that for large files, it's much more efficient to use the "-i" or "-ii" switch of the ALN linker rather than the .INCBIN directive; your compile times and object file sizes will be significantly shorter. |
| **.include** *"file"* - | Include a file. If the filename is not enclosed in quotes, then a default extension of ".s" is applied to it. if the filename is quoted, then the name is not changed in any way.<br><br>Note: If the filename is not quoted and not a valid symbol, then the assembler will generate an error message. You should enclose filenames such as "ATARI.S" in quotes, because such names are not valid symbols.<br><br>If the include file cannot be found in the current directory, then the directory search path, as specified by -i on the conunandline, or by the MACPATH enviroment string, is traversed. |
| **.init**[*.size*]<br>[*#expression*]<br>*expression*[*.size*]<br>[, ...] | Generalized initialization directive. The size specified on the directive becomes the default size for the rest of the line. (The "default" default size is **.w**.) A comma-separated list of expressions follows the directive; an expression may be followed by a size to override the default size. An expression may be preceeded by a sharp sign, an expression and a comma, which specifies a repeat count to be applied to the next expression. For example;<br><br>`.init.l -1, 0.w, #16,'z'.b, #3,0, 11.b`<br><br>will deposit a longword of -1, a word of zero, sixteen bytes of lower-case 'z', three longwords of zero, and a byte of 11. No auto-alignment is performed within the line, but a even is done once at the beginning (before the first value is deposited) if the default size is word or long. |
| **.JPAD** | After this directive, a **NOP** instruction will automatically be added after each **JUMP** or **JR** instruction in GPU or DSP assembly mode. The default is for padding to be turned off. Each time you switch sections using the .GPU or .DSP directives, padding is turned off. |
| **.list**<br><br>**.nlist** | Enable or disable source code listing. These directives increment and decrement an internal counter, so they may be appropriately nested. They have no effect if the **-l** switch is not specified on the commandline. |
| **.LONG** | Align the program counter to the next integral long boundary (4 bytes). Note that GPU/DSP code sections are not contained within 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*, ...]<br>**.endm**<br>**.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...*]<br><br>formerly known as:<br>**.undefmac** *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.<br><br>Older versions of Madmac recognized the **.undefmac** directive. In current versions of MADMAC, the **.undefmac** directive has been replaced by the **.macundef** directive. |
| **.NOAUTOEVEN** | Disables automatic word alignment between directives and instructions. For example, if you do:<br><br>`.DC.B $12`<br>`.DC.L $3456789A`<br><br>then Madmac will output $12 $34 $56 $78 $9A regardless of the alignment of the data. This directive does not affect the directives .EVEN, .LONG, .PHRASE, .DPHRASE, .QPHRASE or "-r" commandline switch. The default mode of operation is .AUTOEVEN. |
| **.NOCLEAR** | After this directive, Madmac no longer allows the use of the **CLR.L** instruction for the 680x0. The **CLR.L** instruction does not work properly on the Jaguar when accessing hardware register locations. The default state is .CLEAR. |
| **.NOJPAD** | After this directive, **NOP** instructions will no longer be added automatically after each **JUMP** or **JR** instruction in GPU or DSP assembly mode. |
| **.NOLIST** | Turns off the assembly listing output. This is basically the same as the .NLIST directive, and has been added for better compatibility with other assemblers. |
| **.offset** [*location*]<br><br>formerly known as:<br>**.abs** [*location*] | Start an absolute section, beginning with the specified *location* (or zero, if no location is specified). An absolute section is much like BSS, except that locations declared with the **.ds** directive are absolute and not relocatable by the linker. This directive is useful for declaring structures or hardware locations. For example, the following equates:<br><br>`VPLANES = 0`<br>`VWRAP = 2`<br>`CONTRL = 4`<br>`INTIN = 8`<br>`PISIN = 12`<br><br>could be as easily defined as:<br><br>`.abs`<br>`VPLANES: ds.w 1`<br>`VWRAP: ds.w 1`<br>`CONTRL: ds.l 1`<br>`INTIN: ds.l 1`<br>`PTSIN: ds.l 1`<br><br>Older versions of MADMAC recognized the **.abs** directive. In current versions of MADMAC, the **.abs** directive has been replaced by the **.offset** directive. |

 

| | |
|---|---|
| **.ORG** *expession* | 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.<br><br>The **.ORG** directive is intended for Jaguar GPU, Jaguar DSP, or 6502 code. It is not legal in 68000 sections. For 6502 sections, the address specified must be less than $10000 (the upper limit of the 6502 address range.)<br><br>All symbols generated following this directive will be non-relocatable. |
| **.PHRASE** | Align the program counter to the next integral phrase boundary (8 byte). Note that GPU/DSP code sections are not contained within 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. If an expression is undefined, Madmac will output "<???>" instead of the value. Several format flags that can be used to format your output are also supported. If the value is a label with a value relative to the start of the TEXT, DATA, or BSS segments, it will be displayed in a format like "TEXT + x".<br><br>/x hexadecimal<br>/d signed decimal<br>/u unsigned decimal<br>/w word<br>/l long<br><br>For example:<br><br>`MASK .EQU $FFF8`<br>`VALUE .EQU -100000`<br>`.print "Mask: $",/x/w MASK`<br>`.print "Value: ",/d/l VALUE`<br><br>This will print "Mask: $FFF8" and "Value: -100000" |
| **.QPHRASE** | Align the program counter to the next integral quad phrase boundary (32 bytes). Note that GPU/DSP code sections are not contained within 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*<br>**.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. |
| **.title** "string"<br><br>**.subttl** [-1 "string" | Set the title or subtitle on the listing page. The title should be specified on the the first line of the source program in order to take effect on the first page. The second and subsequent uses of title will cause page ejects. The second and subsequent uses of .suhttl will cause page ejects unless the subtitle string is preceeded by a dash (-). |

## Notes On Assembly Directives:

- The directives **.INIT**, **.CARGS**, **.TEXT**, **.DATA**, and **.BSS** are forbidden while in GPU or DSP sections.

## 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, GPU, or DSP instruction mnemonic or an assembler directive. (The name may begin with a period - macros cannot be made locally confined like labels or equated symbols.) The formal argument list is optional; it is specified with a comma-separated 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** directive. 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
\{godzi11a}vs\{reagan}

The formal parameter name is terminated with a character that is not valid in a symbol (e.g. whitespace or puncuation); 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
deposit     4           ; word of 4 (no explicit size)
```

## 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.
- 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 hexadecimal |
| \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 pop-out of their include levels appropriately. That is, if a macro performs a include to include a source file, and 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 availabilityof 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.

## Example Macros

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 as 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. Additionally, the ***Fopen*** macro calls another macro.

 

```
macro Fopen file, mode
            move.w      \mode,-(sp)   ; push open mode
            move.l      \file,-(sp)   ; push address of file nane
            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 the string's address there
            .else                     ; otherwise
            pea         .\            ; push the Btring's address
            .endif
            .data                     ; put the string data
.\~:        dc.b        \str,0        ; in the data segment
            text                      ; and switch back to the text
segment
            .endm
```

The construction ".\~" will expand to a label of the form ".M*n*" (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 AO
        .endr
```

## 68000 Mode

All of the standard Motorola 68000 mnemonics and addressing modes are supported; you should refer to **The Motorola M68000 Programmer's Reference Manual** for a description of the instruction set and the allowable addressing modes for each instruction. With one major exception (forward branches) 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. The alternate forms R0 through R15 may be used to specify D0-D7 and A0-A7. 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

Since MADMAC is a one pass assembler, forward branches cannot 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. 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 |

## Linker Constraints

It is not possible to make an external reference that will fix up a byte. For example:

```
.extern     frog
move.l      frog(pc,d0),d1
```

is illegal (and generates an assembly error) when ***frog*** is external, because the displacement occupies a byte field in the 68000 offset word, which the object file cannot represent[3].

## Optimizations and Translations

The assembler provides "creature comforts" when it processes 68000 mnemonics:

- **CLR.x An** will really generate **SUB.x An,An**. -- -
- **ADD**, **SUB** and **CMP** with an address register will really generate **ADDA**, **SUBA** and **CMPA**.
- The **ADD**, **AND**, **CMP**, **EOR**, **OR**, and **SUB** mnemonics with immediate first operands will generate the "I" forms of their instructions (**ADDI**, etc.) if the second operand is not register direct.
- All shift instructions with no count value assume a count of one.
- **MOVE.L** is optimized to **MOVEQ** if the immediate operand is defined and in the range -128 to 127. However, **ADD** and **SUB** are never translated to their quick forms; **ADDQ** and **SUBQ** must be explicit.

3 I don't think this applies to output of BSD object modules.

# Jaguar GPU/DSP Mode

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

## Condition Codes

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

### Motorola-Style

| | |
|---|---|
| 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 |

### Intel-Style[4]

| | |
|---|---|
| A | = %00101 |
| NBE | = %00101 |
| AE | = %00100 |
| NB | = %00100 |
| B | = %01000 |
| NAE | = %01000 |
| E (Equal) | = %00010 |
| NE (Not Equal) | = %00001 |
| NZ (Not Zero) | = %00001 |
| NS | = %01110 |
| S | = %10010 |

## 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)**

[4] Unfortunately, we have been unable to track down the definitions of all of the Intel-style condition code mnemonics, although their meanings can be derived by comparison with the Motorola-style mnemonics. They are included primarily for purposes of backwards compatibility with the GASM assembler.

- Madmac tests all GPU/DSP restrictions, and corrects them whenever 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. A warning is displayed.

- Older versions of Madmac supported the use of the register names R0, R1, R2, ... R15 in 68000 code sections. This is no longer supported because of the conflict with Jaguar GPU/DSP register names. Use D0 to D7, A0 to A7, and SP instead.

## 6502 Support

*Please note that the 6502 assembly capabilities of MADMAC have not been tested since the addition of the Jaguar GPU and DSP assembly modes. It is quite possible that the 6502 capabilities are broken in current versions of MADMAC.*

MADMAC will generate code for the Motorola 6502 microprocessor. This chapter describes extra addressing modes and other features used to support the 6502.

As the 6502 object code is not linkable (currently there is no linker) external references may not be made. (Nevertheless, MADMAC may reasonably be used for large, all-inclusive assemblies because of its blinding speed.)

All standard 6502 addressing modes are supported, with the exception of the accumulator addressing form, which must be omitted (e.g. "ror a" becomes "ror"). Five extra modes, synonyms for existing ones, are included for compatibility with the Atari Coinop assembler.

| | |
|---|---|
| empty | implied or accumulator (e.g. tsx or ror) |
| expr | absolute or zeropage |
| #expr | immediate |
| (expr,x) | indirect X |
| (expr),y | indirect Y |
| (expr) | indirect |
| expr,x | indexed X |
| expr,y | indexed Y |
| @expr(x) | indirect X |
| @xpr(y) | indirect Y |
| @expr | indirect |
| x,expr | indexed X |
| y,expr | indexed Y |

While MADMAC lacks 'high" and 'low" operators, high bytes of words may be extracted with the shift (») or divide (/) operators, and low bytes may be extracted with the bitwise AND (a) operator.

See the descriptions of the **.6502**, **.org**, and **.68000** directives in the **Directives** section for information on how these directives affect 6502 assembly mode.

**.org** location
This directive is only legal in non-68000 sections. It sets the value of the location counter (or pc) to location, an expression that must be defined, absolute, and less than $10000 (the upper limit of the 6502 address range.)

## WARNING

It is possible to assemble "beyond" the microprocessor's 64K address space, but attempting to do so will probably screw up the assembler. DO NOT attempt to generate code like this:

```
org     $FFFE
nop
nop
nop
```

the third **nop** in this example, at location $10000, may cause the assembler to crash or exhibit spectacular schizophrenia. In any case, MADMAC will give no warning before flaking out.

## Object Code Format

This is a little bit of a kludge. An object file consists of a page map, followed by one or more page images, followed by a normal Alcyon 68000 object file. If the page map is all zero, it is not written.

The page map contains a byte for each of the 256 256-byte pages in the 6502's 64K address space. The byte is zero ($0) if the page contained only zero bytes, or one ($01) if the page contained any non-zero bytes. If a page is flagged with a one, then it is written (in order) following the page map.

The following code:

```
.6502
org     $8000
.dc.b   1
org     $8100
.dc.b   1
org     $8300
.dc.b   1
end
```

will generate a page map that looks (to a programmer) something like:

```
<$80 bytes of zero>
01 01 00 01
<$7C more bytes of zero, for $100 total>
<image of page $80>
<image of page $81>
<image of page $83>
```

Following the last page image is an Alcyon-format object file, starting with the magic number $601A. It may contain 68000 code (although that is probably useless), but the symbol table is valid and available for debugging purposes. 6502 symbols will be absolute (not in **text**, **data** or **bss** sections).

# Error Messages

## When Things Go Wrong

Most of MADMAC'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.[5]

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

"filename", line line-number: message

The first element, a filename enclosed in double quotes, indicates the file that generated the error. The filename is followed by a comma, the word 'line", and a line number, and finally a colon and the text of the message. The filename "(*top*)" indicates that the assembler could not determine which file had the problem.

The following sections list warnings, errors and fatal errors in alphabetical order, along with a short description of what may have caused the problem.

## Warnings

**bad backslash code in string**
You tried to follow a backslash in a string with a character that the assembler didn't recognize. Remember that MADMAC uses a C-language style escape system in strings.

**label ignored**
You specified a label before a macro, rept or endas directive. The assembler is warning you that the label will not be defined in the assembly.

5 Of course, if you come across an internal error, Atari would appreciate it if you would contact Developer Support and let us know about the problem.

**unoptimized short branch**
This warning is only generated if the **-s** switch is specified on the command line. The message refers to a forward, unsized long branch that you could have made short (.s).

## Fatal Errors

**cannot continue**
As a result of previous errors, the assembler cannot continue processing. The assembly is aborted.

**line too long as a result of macro expansion**
When a source line within a macro was expanded, the resulting line was too long for MADMAC (longer than 200 characters or so).

**memory exhausted**
The assembler ran out of mernory. You should (1) split up your source files and assemble them seperately, or (2) if you have any ramdisks or RAM-resident programs (like desk accessories) decrease their size so that the assembler has more RAM to work with. As a rule of thumb, pure 68000 code will use up to twice the number of bytes contained in the source files, whereas 6502 code will use 64K of RAM right away, plus the size of the source files. The assembler itself uses about 80K bytes. Get out your calculator...

**too many ENDMs**
The assembler ran across an **endm** directive when it wasn't expecting to see one. The assembly is aborted. Check the nesting of your macro definitions - you probably have an extra **endm**.

## Errors

**.cargs syntax**
Syntax error in **.cargs** directive.

**comm symbol already defined**
You tried to **.comm** a symbol that was already defined.

**.ds permitted only In BSS**
You tried to use the **.ds** directive in the **text** or **data** section.

**.init not permitted in BSS or ABS**
You tried to use **.init** in a BSS or ABS section.

**.org permitted only in .6502 section**
You tried to use **.org** in a 68000 section.

**Cannot create: filename**
The assembler could not create the indicated filename.

**External quick reference**
You tried to make the immediate operand of a **moveq**, **subq** or **addq** instruction external.

**PC-relative expr across sections**
You tried to make a PC-relative reference to a location contained in another section.

**[[bwsl] must follow '.' in symbol**
You tried to follow a dot in a symbol name with something other than one of the four characters 'B', 'W', 'S', or 'L'.

**addressing mode syntax**
You made a syntax error in an addressing mode.

**assert failure**
One of your assert directives failed!

**bad (section) expression**
You tried to mix and match sections in an expression.

**bad 6502 addressing mode**
The 6502 mnemonic will not work with the addressing mode you specified.

**bad expression**
There's a syntax error in the expression you typed.

**bad size specified**
You tried to use an inappropriate size suffix for the instruction. Check your 68000 manual for allowable sizes.

**bad size suffix**
You can't use .h (byte) mode with the **movem** instruction.

**cannot** .globl **local symbol**
You tried to make a confined symbol global or common.

**cannot initialize non-storage (BSS) section**
You tried to generate instructions (or data, with the **dc** directive) in the BSS or ABS section.

**cannot use '.h' with an address register**
You tried to use a byte-size suffix with an address register. The 68000 does not perform byte-sized address register operations.

**directive illegal in .6502 section**
You tried to use a 68000-oriented directive in the 6502 section.

**divide by zero**
The expression you typed involves a division by zero.

**expression out of range**
The expression you typed is out of range for its application.

**external byte reference**
You tried to make a byte-sized reference to an external symbol, which the object file format will not allow.

**external short branch**
You tried to make a short branch to an external symbol, which the linker cannot handle.

**extra (unexpected) text found after addressing mode**
MADMAC thought it was done processing a line, but it ran up against "extra" stuff. Be sure that any comment on the line begins with a semicolon, and check for dangling commas, etc.

**forward or undefined assert**
The expression you typed after a **assert** directive had an undefined value. Remember that MADMAC is a one pass assembler.

**hit EOF without finding matching endif**
The assembler fell off the end of last input file without finding an **.endif** to match an **.if**. You probably forgot an **.endif** somewhere.

**illegal 6502 addressing mode**
The 6502 instruction you typed doesn't work with the addressing mode you specified.

**illegal absolute expression**
You can't use an absolute-valued expression here.

**illegal bra.s with zero offset**
You can't do a short branch to the very next instruction (read your 68000 manual).

**illegal byte-sized relative reference**
The object file format does not permit bytes contain relocatable values; you tried to use a byte-sized relocatable expression in an immediate addressing mode.

**illegal character**
Your source file contains a character that MADMAC doesn't allow. (Most control characters fall into this category.)

**illegal initialization of section**
You tried to use **.dc** or **.dcb** in the BSS or ABS sections.

**illegal relative address**
The relative address you specified is illegal because it belongs to a different section.

**illegal word relocatable (in PRG mode)**
You can't have anything other than long relocatable values when you're generating a .PRG file.

**inappropriate addressing mode**
The mnemonic you typed doesn't work with the addressing modes you specified. Check your 68000 manual for allowable combinations.

**invalid addressing mode**
The combination of addressing modes you picked for the **movem** instruction are not implemented by the 68000. Check your 68000 reference manual for details.

**invalid symbol following ^^**
What followed the ^^ wasn't a valid symbol at all.

**mis-nested** .endr
The assembler found a **.endr** directive when it wasn't prepared to find one. Check your repeat-block nesting.

**mismatched** .else
The assembler found a **.else** directive when it wasn't prepared to find one. Check your conditional assembly nesting.

**mismatched** .endif
The assembler found a **.endif** directive when it wasn't prepared to find one. Check your conditional assembly nesting.

**missing '='**
**missing '}'**
**missing argument name**
**missing close parenthesis ')'**
**missing close parenthesis ']'**
**missing comma**
**missing filename**
**missing string**
**missing symbol**
**missing symbol or string**
The assembler expected to see a symbol/filename/string (etc...), but found something else instead. In most cases the problem should be obvious.

**misuse of '.', not allowed in symbols**
You tried to use a dot (.) in the middle of a symbol name.

**mod (%) by zero**
The expression you typed involves a modulo by zero.

**multiple formal argument definition**
The list of formal parameter names you supplied for a macro definition includes two identical names.

**multiple macro definition**
You tried.to define a macro which already had a definition.

**non-absolute byte reference**
You tried to make a byte reference to a relocatable value, which the object file format does not allow.

**non-absolute byte value**
You tried to **dc.b** or **dcb.b** a relocatable value. Byte relocatable values are not permitted by the object file format.

**register list order**
You tried to specify a register list like D7-D0, which is illegal. Remember that the first register number must be less than or equal to the second register number.

**register list syntax**
You made an error in specifying a register list for a **.reg** directive or a **movem** instruction.

**symbol list syntax**
You probably forgot a comma between the names of two symbols in a symbol list, or you left a comma dangling on the end of the line.

**syntax error**
This is a "catch-all" error message for errors which are not covered by other messages.

**undefined expression**
The expression has an undefined value because of a forward reference, or an undefined or external symbol.

**unimplemented addressing mode**
You tried to use 68020 "square-bracket" notation for a 68020 addressing mode. MADMAC does not support 68020 addressing modes.

**unimplemented directive**
You have found a directive that didn't appear in the documentation. It doesn't work.

**unimplemented mnemonic**
You've found an assembler (or documentation) bug.

**unknown symbol following ^^**
You followed a ^^ with something other than one of the names **defined**, **referenced** or **streq**.

 

**unsupported 68020 addressing mode**
The assembler saw a 68020-type addressing mode. MADMAC does not assemble code for the 68020 or 68010.

**unterminated string**
You specified a string starting with a single or double quote, hut forgot to type the closing quote.

**write error**
The assembler had a problem writing an object file. This is usually caused by a full disk, or a bad sector on the media.