RMAC

68000 Macro Assembler

Reference Manual

version 2.4.12

© and notes

NOTE: Every effort has been made to ensure the accuracy and robustness of this manual and the associated software. However, the authors are constantly improving and updating its computer software, we is unable to guarantee the accuracy of printed or duplicated material after the date of publication and disclaims liability for changes, errors or omissions.

Contents

Contents

Introduction

Introduction

This document describes RMAC, a fast macro assembler for the 68000. RMAC currently runs on the any POSIX compatible platform and the Atari ST. It was initially written at Atari Corporation by programmers who needed a high performance assembler for their work. Then, more than 20 years later, because there was still a need for such an assembler and what was available wasn't up to expectations, Subqmod and eventually the rmac authors continued work on the freely released source, adding Jaguar extensions and fixing bugs. Over time the assembler has been extended by adding support for Motorola's 68020/30/40/60, 68881/2, DSP56001 CPUs, Atari's Object Processor (OP) found on the Atari Jaguar, as well as MOS 6502.

RMAC is intended to be used by programmers who write mostly in assembly language. It was not originally a back-end to a C compiler, therefore it has creature comfort that are usually neglected in such back-end assemblers. It supports include files, macros, symbols with limited scope, some limited control structures,

and other features. RMAC is also blindingly fast, another feature often sadly and obviously missing in today's assemblers.[1]

RMAC is not entirely compatible with the AS68 assembler provided with the original Atari ST Developer's Kit, but most changes are minor and a few minutes with an editor should allow you to assemble your current source files. If you are an AS68 user, before you leap into the unknown please read the section on Notes for AS68 Users.

Getting Started

- The distribution disk contains a file called README that you should read. This file contains important nays about the contents of the distribution disk and summarizes the most recent changes to the tools.
- Hard disk users can simply copy the executable files to their work or binary directories. People with floppy disks can copy the executables to ramdisks, install the assembler with the -q option, or even work right off of the floppies.
- You will need an editor that can produce "normal" format text files. Micro Emacs will work well, as will most other commercial program editors, but not most word processors (such as First Word or Microsoft Write).
- You will probably want to examine or get a listing of the file "ATARI.S". It contains lots of definitions for the Atari ST, including BIOS variables, most BIOS, XBIOS and GEMDOS traps, and line-A equates. We (or you) could split the file up into pieces (a file for line-A equates, a file for hardware and BIOS variables and so on), but RMAC is so fast that it doesn't matter much.
- Read the rest of the manual, especially the first two chapters on The Command Line and Using RMAC. Also, Notes for migrating from other 68000 assemblers will save a lot of time and frustration in the long run. The distribution disk contains example programs that you can look at, assemble and modify.

The Command Line

The assembler is called "**rmac**" or "**rmac.prg**". The command line takes the form:

rmac [*switches*] [*files* ...]

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

Switch order is important. Command lines are processed from left to right in one pass, and switches usually take effect when they are encountered. In general it is best to specify all switches before the names of any input files.

If the command line is entirely empty then RMAC prints a copyright message along with usage info and exit.

Input files are assumed to have the extension "**.s**"; if a filename has no extension (i.e. no dot) then "**.s**" will be appended to it. More than one source filename may be specified: the files are assembled into one object file, as if they were concatenated.

RMAC normally produces object code in "**file.o**" if "**file.s**" is the first input filename. If the first input file is a special character device, the output name is noname.o. The **-o** switch (see below) can be used change the output file name.

Switch	Description
-dname*[=value]*	Define symbol, with optional value. Prefix value with **$** for hexadecimal input.
-e*[file[.err]]*	Direct error messages to the specified file.
-fa	ALCYON output object file format (implied when **-ps** is enabled).
-fb	BSD COFF output object file format.
-fc	Commodore 64 PRG format.

-fe	ELF output object file format.
-fr	Absolute address. Source code is required to have one .org statement.
-fx	Atari 800 com/exe/xex output object file format.
-g	Generate source level debug info. Requires BSD COFF object file format, or ST *.prg* format where it will generate HiSoft Line Number format.
-i*path*	Set include-file directory search path.
-l*[file[prn]]*	Construct and direct assembly listing to the specified file.
-l**[filename]*	Create an output listing file without pagination.
-m*cpu*	Switch CPU type *68000 - MC68000* *68020 - MC68020* *68030 - MC68030* *68040 - MC68040* *68060 - MC68060* *68881 - MC68881* *68882 - MC68882* *56001 - DSP56001* *6502 - MOS 6502* *tom - Jaguar GPU JRISC* *jerry - Jaguar DSP JRISC*
-o*file[.o]*	Direct object code output to the specified file.
+/~oall	Turn all optimisations on/off
+o*0-30*/*p*	Enable specific optimisation
~o*0-30*/*p*	Disable specific optimisation *0: Absolute long adddresses to word* *1: move.l #x,Dn/An to moveq* *2: Word branches to short* *3: Outer displacement 0(An) to (An)* *4: lea to addq* *5: 68020+ Absolute long base/outer displacement to word* *6: Convert null short branches to NOP* *7: Convert clr.l Dn to moveq #0,Dn* *8: Convert adda.w/l #x,Dy to addq.w/l #x,Dy* *9: Convert adda.w/l #x,Dy to lea x(Dy),Dy* *10: 56001 Use short format for immediate values if possible* *11: 56001 Auto convert short addressing mode to long (default: on)* *30: Enforce PC relative (alternative name: op)*
-p	Produce an executable (**.prg**) output file.
-ps	Produce an executable (**.prg**) output file with symbols.
-px	Produce an executable (**.prg**) output file with extended symbols.
-q	Make RMAC resident in memory (Atari ST only).

-r *size*	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)*
-s	Warn about unoptimized long branches and applied optimisations.
-u	Force referenced and undefined symbols global.
-v	Verbose mode (print running dialogue).
-x	Turn on debugging mode.
-yn	Set listing page size to n lines.
-4	Use C style operator precedence.
file*[s]*	Assemble the specified file.

The switches are described below. A summary of all the switches is given in the table.

-d

The **-d** switch permits symbols to be defined on the command line. The name of the symbol to be defined immediately follows the switch (no spaces). The symbol name may optionally be followed by an equals sign (=) and a decimal number. If no value is specified the symbol's value is zero. The symbol attributes are "defined, not referenced, and absolute". This switch is most useful for enabling conditionally-assembled debugging code on the command line; for example:

```
-dDEBUG -dLoopCount=999 -dDebugLevel=55
```

-e

The -e switch causes RMAC to send error messages to a file, instead of the console. If a filename immediately follows the switch character, error messages are written to the specified file. If no filename is specified, a file is created with the default extension "**.err**" and with the root name taken from the first input file name (e.g. error messages are written to "**file.err**" if "**file**" or "**file.s**" is the first input file name). If no errors are encountered, then no error listing file is created. Beware! If an assembly produces no errors, any error file from a previous assembly is not removed.

-g

The **-g** switch causes RMAC to generate source-level debug symbols using the stabs format. When linked with a compatible linker such as RLN, these symbols can be used by source-level debuggers such as rdbjag, wdb, or gdb to step through assembly code line-by-line with all the additional context of labels, macros, constants, register equates, etc. available in the original assembly listings rather than relying on the simple disassembly or raw machine code available when stepping through instruction-by-instruction. This option only works with the BSD COFF object file format, as the others do not use the a.out-style symbol tables required by stabs, and RMAC does not currently support placing stabs debug symbols in their own dedicated section in ELF format object files.

-i

The **-i** switch allows automatic directory searching for include files. A list of semi-colon seperated directory search paths may be mentioned immediately following the switch (with no spaces anywhere). For example:

```
-im:;c:include;c:include\sys
```

will cause the assembler to search the current directory of device **M**, and the directories include and includesys on drive **C**. If *-i* is not specified, and the enviroment variable "**RMACPATH**" exists, its value is used in the same manner. For example, users of the Mark Williams shell could put the following line in their profile script to achieve the same result as the **-i** example above:

```
setenv RMACPATH="m:;c:include;c:include\sys"
```

-l

The -l switch causes RMAC to generate an assembly listing file. If a file- name immediately follows the switch character, the listing is written to the specified file. If no filename is specified, then a listing file is created with the default extension "**.prn**" and with the root name taken from the first input file name (e.g. the listing is written to "**file.prn**" if "**file**" or "**file.s**" is the first input file name).

-o

The -o switch causes RMAC to write object code on the specified file. No default extension is applied to the filename. For historical reasons the filename can also be seperated from the switch with a space (e.g. "**-o file**").

-p

-ps

The **-p** and **-ps** switches cause RMAC to produce an Atari ST executable file with the default extension of "**.prg**". If there are any external references at the end of the assembly, an error message is emitted and no executable file is generated. The **-p** switch does not write symbols to the executable file. The **-ps** switch includes symbols (Alcyon format) in the executable file.

-q

The **-q** switch is aimed primarily at users of floppy-disk-only systems. It causes RMAC to install itself in memory, like a RAMdisk. Then the program **m.prg** (which is very short - less than a sector) can be used instead of **mac.prg**, which can take ten or twelve seconds to load. (**NOTE** not available for now, might be re-implemented in the future).

-s

The **-s** switch causes RMAC to generate a list of unoptimized forward branches as warning messages. This is used to point out branches that could have been short (e.g. "bra" could be "bra.s").

-u

The **-u** switch takes effect at the end of the assembly. It forces all referenced and undefined symbols to be global, exactly as if they had been made global with a **.extern** or **.globl** directive. This can be used if you have a lot of external symbols, and you don't feel like declaring them all external.

-v

The **-v** switch turns on a "verbose" mode in which RMAC prints out (for example) the names of the files it is currently processing. Verbose mode is automatically entered when RMAC prompts for input with a star.

-y

The **-y** switch, followed immediately by a decimal number (with no intervening space), sets the number of lines in a page. RMAC will produce *N* lines before emitting a form-feed. If *N* is missing or less than 10 an error message is generated.

-4

Use C style order of precedence in expressions. See Order of Evaluation for more information.

Using RMAC

Let's assemble and link some example programs. These programs are included on the distribution disk in the "**EXAMPLES**" directory - you should copy them to your work area before continuing. In the following examples we adopt the conven- tions that the shell prompt is a percent sign (%) and that your input (the stuff you type) is presented in **bold face**.

If you have been reading carefully, you know that RMAC can generate an executable file without linking. This is useful for making small, stand alone programs that don't require externals or library routines. For example, the following two commands:

```
% rmac examples
% aln -s example.s
```

could be replaced by the single command:

```
% rmac -ps example.s
```

since you don't need the linker for stand-alone object files.

Successive source files named in the command line are are concatenated, as in this example, which assembles three files into a single executable, as if they were one big file:

```
% rmac -p bugs shift images
```

Of course you can get the same effect by using the **.include** directive, but sometimes it is convenient to do the concatenation from the command line.

Here we have an unbelievably complex command line:

```
% rmac -lzorf -y95 -o tmp -ehack -Ddebug=123 -ps example
```

This produces a listing on the file called "**zorf.prn**" with 95 lines per page, writes the executable code (with symbols) to a file called "**tmp.prg**", writes an error listing to the file "**hack.err**", specifies an include-file path that includes the current directory on the drive "**M:**," defines the symbol "**debug**" to have the value 123, and assembles the file "**example.s**". (Take a deep breath - you got all that?)

One last thing. If there are any assembly errors, RMAC will terminate with an exit code of 1. If the assembly succeeds (no errors, although there may be warnings) the exit code will be 0. This is primarily for use with "make" utilities.

Things You Should Be Aware Of

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

- In listings, the object code for forward references is not shown. Instead, lower- case "xx"s are displayed for each undefined byte, as in the following example:

```
60xx      1: bra.s.2  ;forward branch
xxxxxxxx     dc.l .2  ;forward reference
60FE     .2: bra.s.2  ;backward reference
```

- Forward branches (including **BSR**s) are never optimized to their short forms. To get a short forward branch it is necessary to explicitly use the ".s" suffix in the source code.
- 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 files, re-sizing or eliminating memory-using programs such as ramdisks and desk accessories, or buying more RAM.
- rmac can be indentified during assembly by checking for the definition of a symbol called `_RMAC_`. Furthermore, the exact version of the assembler can be obtained by reading the value of `_RMAC_`. It is encoded in BCD format using the following convention:

1. Major version (1 BCD digit)
2. Minor version (1 BCD digit)
3. Patch version (2 BCD digits)

For exmaple, version 2.1.15 is encoded as `$02010105`.

Forward Branches

RMAC does not optimize forward branches for you, but it will tell you about them if you use the -s (short branch) option:

```
% mac -s example.s
"example.s", line 20: warning: unoptimized short branch
```

With the -e option you can redirect the error output to a file, and determine by hand (or editor macros) which forward branches are safe to explicitly declare short.

Notes for migrating from other 68000 assemblers

RMAC is not entirely compatible with the other popular assemblers like Devpac or vasm. This section outlines the major differences. In practice, we have found that very few changes are necessary to make other assemblers' source code assemble.

- A semicolon (;) must be used to introduce a comment, except that a star (*) may be used in the first column. AS68 treated anything following the operand field, preceeded by whitespace, as a comment. (RMAC treats a star that is not in column 1 as a multiplication operator).
- Labels require colons (even labels that begin in column 1).
- Conditional assembly directives are called **if**, **else** and **endif**. Devpac and vasm call these **ifne**, **ifeq** (etc.), and **endc**.
- **ifd** is **if ^^defined**.
- The tilde (~) character is an operator, and back-quote (`) is an illegal character. AS68 permitted the tilde and back-quote characters in symbols.
- There are no equivalents to org or section directives apart from .text, .data, .bss. The **.xdef** and **.xref** directives are not implemented, but **.globl** makes these unnecessary anyway.
- The location counter cannot be manipulated with a statement of the form:

  ```
  * = expression
  ```

  Exceptions to this rule are when outputting a binary using the **-fr** switch, 6502 mode, and Jaguar GPU/DSP.
- Back-slashes in strings are "electric" characters that are used to escape C-like character codes. Watch out for GEMDOS path names in ASCII constants - you will have to convert them to double-backslashes.
- Expression evaluation is done left-to-right without operator precedence. Use parentheses to force the expression evaluation as you wish. Alternatively, use the **-4** switch to switch to C style precedence. For more information refer to Order of Evaluation.
- Mark your segments across files. Branching to a code segment that could be identified as BSS will cause a "Error: cannot initialize non-storage (BSS) section"
- In 68020+ mode **Zan** and **Zri** (register suppression) is not supported.
- rs.b/rs.w/rs.l/rscount/rsreset can be simulated in rmac using **.abs**. For example the following source:

```
rsreset
label1: rs.w 1
label2: rs.w 10
label3: rs.l 5
label4: rs.b 2

size_so_far equ rscount
```

can be converted to:

```
abs
label1: ds.w 1
label2: ds.w 10
label3: ds.l 5
label4: ds.b 2

size_so_far equ ^^abscount
```

- A rare case: if your macro contains something like:

```
macro test
move.l #$\1,d0
endm

test 10
```

then by the assembler's design this will fail as the parameters are automatically converted to hex. Changing the code like this works:

```
macro test
move.l #\1,d0
endm

test $10
```

Text File Format

For those using editors other than the "Emacs" style ones (Micro-Emacs, Mince, etc.) this section documents the source file format that RMAC expects.

- Files must contain characters with ASCII values less than 128; it is not per- missable 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/line-feed, linefeed alone, or carriage-return alone.
- The file is assumed to end with the last terminated line. If there is text beyond the last line terminator (e.g. control-Z) it is ignored.

Source Format

Statements

A statement may contain up to four fields which are identified by order of ap- pearance 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 seperated 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 (see: Symbols and Scope).

As an addition, the exclamation mark character (**!**) can be placed at the very first character of a line to disbale all optimisations for that specific line, i.e.

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

Equates

A statement may also take one of these special forms:

symbol **equ** *expression*

symbol **=** *expression*

symbol **==** *expression*

symbol **set** *expression*

symbol **reg** *register list*

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 just like an equate except that it also makes the symbol global. (As with labels, it is illegal to make a confined equate 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 (see: Symbols and Scope). For example:

```
cr    equ    13          carriage-return
if    =      10          line-feed
DEBUG ==     1           global debug flag
count set    0           variable
count set    count + 1   increment variable
.rags reg    d3-d7/a3-a6 register list
.cr          13          confined 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 an upper or lowercase letter, a digit (**0-9**), an underscore, a dollar sign (**$**), or a question mark. (Periods can only begin a symbol, they cannot appear as a symbol continuation character). 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 mnemonics, assembler direc- tives and register names. Symbols are limited to 100 characters. When symbols are written to the object file they are silently truncated to eight (or sixteen) char- acters (depending on the object file format) with no check for (or warnings about) collisions.

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

```
reallyLongSymbolName .reallyLongConfinedSymbolName
a10 ret  move  dc  frog  aa6 a9 ????
.a1 .ret .move .dc .frog .a9 .9 ????
.0  .00  .000 .1  .11. .111 . ._
_frog ?zippo? sys$syetem 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 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:  rta
FF::   subq.w #1,d1
       bmi.s .99
.loop: move.w -1,(a0)+
       dbra  d0,.loop
.99:   its
```

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 classic (in 6502 mode):

```
       .6502
.org   =     $8000
```

which equates a confined symbol to a hexadecimal value, rather than setting the location counter, which the .org directive does (without the equals sign).

Keywords

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

```
Common:
equ set reg
MC68000:
sr ccr pc sp ssp usp
d0 d1 d2 d3 d4 d5 d6 d7
a0 a1 a2 a3 a4 a5 a6 a7
Tom/Jerry:
r0 r1 r2 r3 r4 r5 r6 r7
r8 r9 r10 r11 r12 r13 r14 r15
6502:
x y a
```

```
DSP56001:
x x0 x1 x2 y y0 y1 y2
a a0 a1 a2 b b0 b1 b2 ab ba
mr omr la lc ssh ssl ss
n0 n1 n2 n3 n4 n5 n6 n7
m0 m1 m2 m3 m4 m5 m6 m7
r0 r1 r2 r3 r4 r5 r6 r7
```

Note that only the keywords that correspond to the architecture are illegal, for example `x0` can be used safely in 68000 mode.

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 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 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. RMAC 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    $0c1   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 GEMDOS path names, and that you may have to escape backslash characters in your existing source code. For example, to get the file `c:\auto\ahdi.s` you would specify the string `c:\auto\ahdi.s`.

Register Lists

Register lists are special forms used with the **movem** 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, Rm and Rn,m<n, seperated by a dash. For example:

```
register list           value
-------------           -----
d0-d7/a0-a7             $FFFF
d2-d7/a0/a3-a6          $39FC
d0/d1/a0-a3/d7/a6-a7    $CF83
d0                      $0001
r0-r16                  $FFFF
```

Register lists and register equates may be used in conjunction with the movem 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" registers
```

Expressions

Order of Evaluation

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 (**[]**).

All the above behavior is the default. However if the command line switch **-4** is used, then C style of operator precedence is enforced. The following list shows the order of precedence in this mode, from lowest to highest:

- bitwise XOR ^
- bitwise OR |
- bitwise AND &
- relational = < <= >= > !=
- shifts << >>
- sum + -
- product * /

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 sym- bol, 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 Atari ST 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 yeilds 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:

```
linel:  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 longwords that point to line 2. Line 2 deposits two longwords that point to line 1. Line 3 deposits two longwords that have the absolute value eight. The fourth line will result in an assembly error, since the expressions (re- spectively) attempt to add two relocatable values, shift a relocatable value right by one, and divide a relocatable value by four.

The pseudo-symbol "*****" (star) 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**":

```
too:    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 (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.
^^abscount	Returns the size of current .abs section
^^filesize *string_filename*	Returns the file size of supplied filename

- 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 unref- erenced (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. Do *not* attempt to modulo by 0 or 1.
& | ^	Bit-wise **AND**, **OR** and **Exclusive-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, with the exception of the **-4** switch. For more information refer to Order of Evaluation.
- 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. 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. 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   (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   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            ; AO -> message
       jsr    debug                ; print a message
       rts                         ; and return
string: dc.b  "Help me, Spock!",0  ; (the message)
            .
            .
            .
       .iif ^^referenced 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 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**, **.1**, **.d**, **.x**, **.p**, or **.q**); the default is word (**.w**) if no size is specified. The **.s** suffix is identical to **.b**, with the exception of being used in a **dc** statement. In that case the **.s** refers to single precision floating point numbers. Directives relating to the 6502 are described in the chapter on 6502 Support.

.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.

.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.

.print

This 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
```

.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.

.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.

.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.

.align *expression* [, *value*]

A generalised version of the above directives, this will align the program counter to the boundary of the specified value. Note that there is not much error checking happening (only values 0 and 1 are rejected). Also note that in DSP56001 mode the align value is assumed to be in DSP words, i.e. 24 bits. Additionally, it is possible to supply a 16-bit *value* which is used to fill the gap.

.assert *expression* [,*expression*...]

Assert that the conditions are true (non-zero). If any of the comma-seperated 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. 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.

.68000 .68020 .68030 .68040 .68060

Enable different flavours of the MC68000 family of CPUs. Bear in mind that not all instructions and addressing modes are available in all CPUs so the correct CPU should be selected at all times. Notice that it is possible to switch CPUs during assembly.

.68881 .68882

Enable FPU support. Note that *.68882* is on by default when selecting *.68030*.

.56001

Switch to Motorola DSP56001 mode.

.org *location* [*X:*/*Y:*/*P:*/*L:*]

This directive sets the value of the location counter (or **pc**) to location, an expression that must be defined and absolute. It is legal to use the directive in the following modes: 6502, Tom, Jerry, OP, 56001 and 680x0 (only with -fr switch). Especially for the 56001 mode the *location* field **must** be prefixed with the intended section (*X:*, *Y:*, *P:* or *L:*).

.opt *"+On"* **.opt** *"~On"* **.opt** *"+Oall"* **.opt** *"~Oall"*

These directives control the optimisations that rmac applies to the source automatically. Each directive is applied immediately from the line encountered onwards. So it is possible to turn specific optimisations on and off globally (when placed at the start of the first file) or locally (by turning desired optimisations on and off at certain parts of the source). For a list of the optimisations (*n*) available please consult the table in section The Command Line.

all, as expected, turns all available optimisations on or off. An exception to this is *o10*/*op* as this is not an optimisation that should be turned on unless the user absolutely needs it.

Lastly, as a "creature comfort" feature, if the first column of any line is prefixed with an exclamation mark (*!*) then for that line all optimisations are turned off.

.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 .ds are based absolute. This directive is useful for declaring structures or hardware locations. For example, the following equates:

```
VPLANES = 0
VWRAP   = 2
CONTRL  = 4
INTIN   = 8
PTSIN   = 12
```

could be as easily defined as:

```
        .abs
VPLANES: ds.w    1
VWRAP:  ds.w     1
CONTRL: ds.l     1
INTIE:  ds.l     1
PTSIN:  ds.l     1
```

Another interesting example worth mentioning is the emulation of "C"'s "union" keyword using *.abs*. For example, the following "C" code:

```
struct spritesheet
{
     short spf_w;
     short spf_h;
     short spf_xo;
     short spf_yo;
     union { int spf_em_colour;     int spf_emx_colour;    };
     union { int spf_em_psmask[16]; int spf_emx_colouropt; };
}
```

can be expressed as:

```
.abs
*-------------------------------------------------------*
spf_w:          ds.w    1   ;<- common
spf_h:          ds.w    1
spf_xo:         ds.w    1
spf_yo:         ds.w    1
spf_data:       ds.l    0
*-------------------------------------------------------*
;           .union  set
spf_em_colour:      ds.l    1   ;<- union #1
spf_em_psmask:      ds.l    16
*-------------------------------------------------------*
.68000
            .abs spf_em_colour
;           .union  reset
spf_emx_colour:     ds.l    1   ;<- union #2
spf_emx_colouropt:  ds.l    1
spf_emx_psmask:     ds.l    16
spf_emx_psmaskopt:  ds.l    16

.68000
;*-------------------------------------------------------*

    move #spf_em_colour,d0
    move #spf_emx_colour,d0
```

In this example, *spf_em_colour* and *spf_emx_colour* will have the same value.

Note: currently this does not work as advertised for 56001 structs.

.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.

.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 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.

.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.

.dsp

Switch to Jaguar DSP assembly mode. This directive must be used within the TEXT or DATA segments.

.dsm *expression*

Only working for 56001 mode.

The DSM directive reserves a block of memory the length of which in words is equal to the value of *expression*.If the runtime location counter is not zero, this directive first advances the runtime location counter to a base address that is a multiple of 2^k, where 2^k >= *expression*.

label, if present, will be assigned the value of the runtime location counter after a valid base address has been established.

This directive should be directly equivalent to Motorola's reference 56001 assembler.

.init[.*size*] [#*expression*,]*expression*[.*size*] [,...]

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-seperated 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:

```
.init.l -1, 0.w, #16,'z'.b, #3,0, 11.b
```

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 (before the first value is deposited) if the default size is word or long.

.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, .fileliams.1, .openMode, .butPointer.l
```

could be used to declare offsets from 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),a0  ; a0 -> string
      moveq  #-1,d0          ; initial size = -1
.1:   addq.1 #1,d0           ; bump size
      tst.b  (a0)+           ; at end of string?
      bne   .1               ; (no -- try again)
      rts                    ; return string length
```

.error ["*string*"]

Aborts the build, optionally printing a user defined string. Can be useful inside conditional assembly blocks in order to catch errors. For example:

```
.if ^^defined JAGUAR
  .error "TOS cannot be built on Jaguar, don't be silly"
.endif
```

.end

End the assembly. In an include file, end the include file and resume 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. For example:

```
.gpu
Clipw .equr r19
. . .
     add ClipW,r0 ; ClipW actually is r19
```

.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 state- ment, which may be an instruction, a directive or a macro, is executed. If the expression is false, the statement is ignored. No **.endif** is required. For example:

```
.iif age < 21, canDrink = 0
.iif weight > 500, dangerFlag = 1
.iif !(^^defined DEBUG), .include dbsrc
```

.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.

.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.

.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.

.globl *symbol* [, *symbol*...]

.extern *symbol* [, *symbol*...]

Each symbol 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**.

.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.

Note: If the filename is 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 symbols.

If the include file cannot be found in the current directory, then the directory search path, as specified by -i on the commandline, or' by the 'RMACPATH' enviroment string, is traversed.

.incbin "*file*" [, [*size*], [*offset*]]

Include a file as a binary. This can be thought of a series of **dc.b** statements that match the binary bytes of the included file, inserted at the location of the directive. The directive is not allowed in a BSS section. Optional parameters control the amount of bytes to be included and offset from the start of the file. All the following lines are valid:

::

.incbin "test.bin" ; Include the whole file .incbin "test.bin",,$30 ; Skip the first 48 bytes .incbin "test.bin",$70,$30 ; Include $70 bytes starting at offset $30 .incbin "test.bin",$48 ; Include the file starting at offset 48 till the end .incbin "test.bin",, ; Include the whole file

.eject

Issue a page eject in the listing file.

.title "*string*"

.subttl [-] "*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 .subttl will cause page ejects unless the subtitle string is preceeded by a dash (-).

.list

.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.

.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:

: *label*

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.

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.

For example, here is a silly way to count from 1 to 10 without using **.rept**:

```
              .macro Count
count         set     1
:loop         dc.w    count
count         set     count + 1
              iif count <= 10, goto loop
              .endm
```

.gpu

Switch to Jaguar GPU assembly mode. This directive must be used within the TEXT or DATA segments.

.gpumain

No. Just... no. Don't ask about it. Ever.

.prgflags *value*

Sets ST executable .PRG field *PRGFLAGS* to *value*. *PRGFLAGS* is a bit field defined as follows:

Definition	Bit(s)	Meaning
PF_FASTLOAD	0	If set, clear only the BSS area on program load, otherwise clear the entire heap.
PF_TTRAMLOAD	1	If set, the program may be loaded into alternative RAM, otherwise it must be loaded into standard RAM.
PF_TTRAMMEM	2	If set, the program's Malloc() requests may be satisfied from alternative RAM, otherwise they must be satisfied from standard RAM.
--	3	Currently unused.
See left.	4 & 5	If these bits are set to 0 (PF_PRIVATE), the processes' entire memory space will be considered private (when memory protection is enabled). If these bits are set to 1 (PF_GLOBAL), the processes' entire memory space will be readable and writable by any process (i.e. global). If these bits are set to 2 (PF_SUPERVISOR), the processes' entire memory space will only be readable and writable by itself and any other process in supervisor mode. If these bits are set to 3 (PF_READABLE), the processes' entire memory space will be readable by any application but only writable by itself.
--	6-15	Currently unused.

.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.

68000 Mnemonics

Mnemonics

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` through `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

68020+ Addressing Modes

The following addressing modes are only valid for 68020 and newer CPUs. In these modes most of the parameters like Base Displacement (**bd**), Outer Displacement (**od**), Base Register (**An**) and Index Register (**Xn**) can be omitted. RMAC will detect this and *suppress* the registers in the produced code.

Other assemblers use a special syntax to denote register suppression like **Zan** to suppress the Base Register and **Rin** to suppress the Index Register. RMAC has no support for this behaviour nor needs it to suppress registers.

In addition, other assemblers will allow reordering of the parameters (for example ([*An*,*bd*])). This is not allowed in RMAC.

Also noteworthy is that the Index Register can be an address or data register.

To avoid internal confusion the 68040/68060 registers *DC*, *IC* and *BC* are named *DC40*, *IC40* and *BC40* respectively.

Assembler Syntax	Description
bd(*An*, *Xi*[.*size*][**scale*])	Address register indirect indexed
([*bd*,*An*],*Xn*[.*siz*][**scale*],*od*)	Register indirect preindexed with outer displacement
([*bd*,*An*,*Xn*[.*siz*][**scale*],*od*)	Register indirect postindexed with outer displacement
([*bd*,*PC*],*Xn*[.*siz*][**scale*],*od*)	Program counter indirect preindexed with outer displacement
([*bd*,*PC*,*Xn*[.*siz*][**scale*],*od*)	Program counter indirect postindexed with outer displacement

Branches

Since RMAC 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.

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.

Branch Synonyms

Alternate name	Becomes:
bhs	bcc
blo	bcs
bse, bs	beq
bns	bne
dblo	dbcs
dbse	dbeq
dbra	dbf
dbhs	dbhi
dbns	dbne

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...127. However, **ADD** and **SUB** are never translated to their quick forms; **ADDQ** and **SUBQ** must be explicit.
- All optimisations are controllable using the **.opt** directive. Refer to its description in section Directives.
- All optimisations are turned off for any source line that has an exclamation mark (*!*) on their first column.

- Optimisation switch 11 is turned on by default for compatibility with the Motorola reference 56001 DSP assembler. All other levels are off by default. (refer to section The Command Line for a description of all the switches).
- Optimisation warnings are off by default. Invoke RMAC with the *-s* switch to turn on warnings in console and listing output.
- 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).
- RMAC 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.

Macros

Macro declaration

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 - 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. It is not advised to begin an argument name with a numeric value.

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 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          ; longvord 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, 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.

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 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.1   \file,-(sp)  ;push address of tile 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 the string's address there
    .else                                       ; otherwise
      pea .\~                                   ; push the string'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                           ;
```

Jaguar GPU/DSP Mode

RMAC 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
```

Jaguar Object Processor Mode

What is it?

An assembler to generate object lists for the Atari Jaguar's Object processor.

Why is it here?

To really utilize the OP properly, it needs an assembler. Otherwise, what happens is you end up writing an assembler in your code to assemble the OP list, and that's a real drag--something that *should* be handled by a proper assembler.

How do I use it?

The OP assembler works similarly to the RISC assembler; to enter the OP assembler, you put the .objproc directive in your code (N.B.: like the RISC assembler, it only works in a TEXT or DATA section). From there, you build the OP list how you want it and go from there. A few caveats: you will want to put a .org directive at the top of your list, and labels that you want to be able to address in 68xxx code (for moving from a data section to an address where it will be executed by the OP, for example) should be created in .68xxx mode.

What are the opcodes?

They are **bitmap**, **scbitmap**, **gpuobj**, **branch**, **stop**, **nop**, and **jump**. **nop** and **jump** are psuedo-ops, they are there as a convenience to the coder.

What are the proper forms for these opcodes?

They are as follows:

bitmap *data addr*, *xloc*, *yloc*, *dwidth*, *iwidth*, *iheight*, *bpp*, *pallete idx*, *flags*, *firstpix*, *pitch*

scbitmap *data addr*, *xloc*, *yloc*, *dwidth*, *iwidth*, *iheight*, *xscale*, *yscale*, *remainder*, *bpp*, *pallete idx*, *flags*, *firstpix*, *pitch*

gpuobj *line #*, *userdata* (bits 14-63 of this object)

branch VC *condition (<, =, >) line #*, *link addr*

branch OPFLAG, *link addr*

branch SECHALF, *link addr*

stop

nop

jump *link addr*

Note that the *flags* field in bitmap and scbitmap objects consist of the following: **REFLECT**, **RMW**, **TRANS**, **RELEASE**. They can be in any order (and should be separated by whitespace **only**), and you can only put a maximum of four of them in. Further note that with bitmap and scbitmap objects, all the parameters after *data addr* are optional--if they are omitted, they will use defaults (mostly 0, but 1 is the default for pitch). Also, in the scbitmap object, the *xscale*, *yscale*, and *remainder* fields can be floating point constants/expressions. *data addr* can refer to any address defined (even external!) and the linker (rln v1.6.0 or greater) will properly fix up the address.

What do they do?

Pretty much what you expect. It's beyond the scope of this little note to explain the Jaguar's Object Processor and how it operates, so you'll have to seek explanations for how they work elsewhere.

Why do I want to put a *.org* directive at the top of my list?

You want to put a *.org* directive at the top of your list because otherwise the assembler will not know where in memory the object list is supposed go--then when you move it to its destination, the object link addresses will all be wrong and it won't work.

Why would I copy my object list to another memory location?

Simple: because the OP destroys the list as it uses it to render the screen. If you don't keep a fresh copy stashed away somewhere to refresh it before the next frame is rendered, what you see on the screen will not be what you expect, as the OP has scribbled all over it!

Does the assembler do anything behind my back?

Yes, it will emit **NOP** s to ensure that bitmaps and scbitmaps are on proper memory boundaries, and fixup link addresses as necessary. This is needed because of a quirk in how the OP works (it ORs constants on the address lines to get the phrases it needs and if they are not zeroes, it will fail in bizarre ways). It will also set all *ypos* constants on the correct half-line (as that's how the OP views them).

Why can't I define the link addresses for all the objects?

You really, *really* don't want to do this. Trust me on this one.

How about an example of an object list?

```
        objList = $10000
        bRam = $20000
;
        .68000
objects:            ; This is the label you will use to address this in 68K code
        .objproc    ; Engage the OP assembler
        .org    objList ; Tell the OP assembler where the list will execute
;
        branch      VC < 69, .stahp     ; Branch to the STOP object if VC < 69
        branch      VC > 241, .stahp    ; Branch to the STOP object if VC > 241
        bitmap      bRAM, 22, 70, 24, 24, 22, 4
        bitmap      bRAM, 20+96+96, 70, 24, 24, 22, 4, 0, REFLECT
        scbitmap    tms, 20, 70, 1, 1, 8, 3.0, 3.0, 2.9999, 0, 0, TRANS
        scbitmap    tmsShadow, 23, 73, 1, 1, 8, 3.0, 3.0, 2.9999, 0, 3, TRANS
        bitmap      sbRelBM, 30, 108, 3, 3, 8, 0, 1, TRANS
        bitmap      txt1BM, 46, 132, 3, 3, 8, 0, 2, TRANS
        bitmap      txt2BM, 46, 148, 3, 3, 8, 0, 2, TRANS
        bitmap      txt3BM, 22, 164, 3, 3, 8, 0, 2, TRANS
        jump        .haha
.stahp:
        stop
.haha:
        jump        .stahp
```

DSP 56001 Mode

RMAC fully supports Motorola's DSP56001 as used on the Atari Falcon and can output binary code in the two most popular formats: *.lod* (ASCII dump, supported by the Atari Falcon XBIOS) and *.p56* (binary equivalent of *.lod*)

Differences from Motorola's assembler

- Motorola's assembler aliases **and #xxx,reg** with **andi #xxx,reg** and can distinguish between the two. rmac needs the user to be explicit and will generate an error if the programmer tries to use syntax from one instruction to the other.
- Similarly Motorola's assembler can alias **move** with **movec**, **movep** and **movem**. rmac also not accept such aliasing and generate an error.

- Motorola's assembler uses the underscore character (_) to define local labels. In order for rmac to maintain a uniform syntax across all platforms, such labels will not be treated as local.
- Macros syntax is different from Motorola's assembler. This includes local labels inside macros. The user is encouraged to study the Macros section and compare syntactical differences.
- Motorola's assembler allows reordering of addressing modes **x:**, **x:r**, **r:y**, **x:y**. rmac will only accept syntax as is defined on the reference manual.
- In **L:** section a dc value cannot be 12 hex digits like Motorola's assmebler. Instead, the value needs to be split into two parts separated by **:**.

6502 Support

RMAC will generate code for the MOS 6502 microprocessor. This chapter describes extra addressing modes and directives 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, RMAC may reasonably be used for large assemblies because of its blinding speed.)

Currently there is no support for undocumented opcodes. This will be addressed in a future release.

6502 Addressing Modes

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	immediate low byte of a word
#>expr	immediate high byte of a word
(*expr*,x)	indirect X
(*expr*),y	indirect Y
(*expr*)	indirect
expr,x	indexed X
expr,y	indexed Y
@*expr*(x)	indirect X
@*expr*(y)	indirect Y
@expr	indirect
x,*expr*	indexed X
y,*expr*	indexed Y

6502 Directives

.6502

This directive enters the 6502 section. The location counter is undefined, and must be set with ".org" before any code can be generated.

The "`dc.w`" directive will produce 6502-format words (low byte first). The 68000's reserved keywords (`d0-d7/a0-a7/ssp/usp` and so on) remain reserved (and thus unusable) while in the 6502 section. The directives **globl**, **dc.l**, **dcb.l**, **text**, **data**, **bss**, **abs**, **even** and **comm** are illegal in the 6502 section. It is permitted, though probably not useful, to generate both 6502 and 68000 code in the same object file.

.68000

This directive leaves the 6502 segment and returns to the 68000's text segment. 68000 instructions may be assembled as normal.

.org *location*

This directive sets the value of the location counter (or **pc**) to location, an expression that must be defined, absolute, and less than $10000.

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, RMAC will give no warning before flaking out.

6502 Object Code Format

Traditionally Madmac had a very kludgy way of storing object files. This has been replaced with a more standard *.exe* (or *.com* or *.xex* if you prefer). Briefly, the *.exe* format consists of chunks of this format (one after the other):

```
Offset     Description
00-01      $FFFF - Indicates a binary load file. Mandatory for first segment, optional for any other segment
02-03      Start Address. The segment will load at this address
04-05      End Address. The last byte to load for this segment
06-..      The actual segment data to load (End Address-Start Address + 1 bytes)
```

In addition there is the standard output format for Commodore 64 binaries (.PRG).

Error Messages

When Things Go Wrong

Most of RMAC'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.[3]

You can write editor macros (or sed or awk scripts) to parse the error messages RMAC 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 RMAC uses a C-style escape system in strings.

label ignored

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

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 resultant line was too long for RMAC (longer than 200 characters or so).

memory exhausted

The assembler ran out of memory. 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.

.init not permitted in BSS or ABS

You tried to use `.init` in the BSS or ABS 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 .b (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 dc) in the BSS or ABS section.

cannot use '.b' 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

RMAC 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 RMAC is one-pass.

hit EOF without finding matching .endif

The assembler fell off the end of last input file without finding a **.endif** to match an . it. You probably forgot a **.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 RMAC 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 gener- ating 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.

undefined expression

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

unimplemented directive

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

unimplemented mnemonic

You've found a bug.

unknown symbol following ^^

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

unterminated string

You specified a string starting with a single or double quote, but 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.

1 It processes 30,000 lines a minute on a lightly loaded VAX 11/780; maybe 40,000 on a 520-ST with an SH-204 hard disk. Yet it could be sped up even more with some effort and without resorting to assembly language; C doesn't have to be slow!

3 If you come across an internal error, we would appreciate it if you would contact the rmac development team and let us know about the problem.