GNU Assembler Manual

Contents

Overview *3*
The GNU Assembler *5*
Object File Formats *5*
Command Line *5*
Input Files *6*
Output (Object) File *6*
Error and Warning Messages *7*
Command-Line Options *7*
Syntax *11*
Preprocessing *11*
Whitespace *11*
Comments *11*
Symbols *12*
Statements *12*
Constants *13*
Sections and Relocation *16*
Background *16*
ld Sections *17*
as Internal Sections *18*
Sub-Sections *18*
bss Section *19*
Symbols *20*
Labels *20*
Giving Symbols Other Values *20*
Symbol Names *20*
The Special Dot Symbol *21*
Symbol Attributes *21*
Value *22*
Type *22*
Symbol Attributes: a.out *22*
Symbol Attributes for COFF *22*
Symbol Attributes for SOM *22*
Expressions *23*
Empty Expressions *23*
Integer Expressions *23*
Arguments *23*
Operators *23*
Prefix Operator *24*
Infix Operators *24*
Assembler Directives *25*
Machine Dependent Features *36*
M680x0 Options *36*
Syntax *37*
Motorola Syntax *38*
Floating Point *39*
680x0 Machine Directives *39*

Following is a user guide to the GNU assembler AS-MCORE. The original GNU assembler and documentation this is based on was simply called AS. So throughout this manual, any remaining reference to AS is a reference to AS-MCORE.

Overview

Here is a brief summary of how to invoke AS-MCORE. For details, see the Command-Line Options section.

```
as-mcore [ -a[dhlns][=file] ] [ -D ]  [ --defsym sym=val ]
     [ -f ] [ --help ] [ -I dir ] [ -J ] [ -K ] [ -L ]
     [ -o objfile ] [ -R ] [ --statistics ] [ -v ] [ -version ]
     [ --version ] [ -W ] [ -w ] [ -x ] [ -Z ]
     [ -- | files ... ]
```

-a[dhlns]

Turn on listings, in any of a variety of ways:

- **-ad** omit debugging directives
- **-ah** include high-level source
- **-al** include assembly
- **-an** omit forms processing
- **-as** include symbols
- **=file** set the name of the listing file

You may combine these options; for example, use `-aln' for assembly listing without forms processing. The `=file' option, if used, must be the last one. By itself, `-a' defaults to `-ahls'---that is, all listings turned on.

-D Ignored. This option is accepted for script compatibility with calls to other assemblers.

--defsym sym=value

Define the symbol sym to be value before assembling the input file. value must be an integer constant. As in C, a leading `0x' indicates a hexadecimal value, and a leading `0' indicates an octal value.

-f "fast"---skip whitespace and comment preprocessing (assume source is compiler output).

--help Print a summary of the command line options and exit.

-I dir Add directory dir to the search list for .include directives.

-J Don't warn about signed overflow.

-K Issue warnings when difference tables altered for long displacements.

-L Keep (in the symbol table) local symbols, starting with `L'.

-o objfile Name the object-file output from as objfile.

-R Fold the data section into the text section.

--statistics Print the maximum space (in bytes) and total time (in seconds) used by assembly.

-v or -version Print the as version.

--version Print the as version and exit.

-W Suppress warning messages.

-w Ignored.

-x Ignored.

-Z Generate an object file even after errors.

-- | files ... Standard input, or source files to assemble.

The GNU Assembler

GNU as is really a family of assemblers. If you use (or have used) the GNU assembler on one architecture, you should find a fairly similar environment when you use it on another architecture. Each version has much in common with the others, including object file formats, most assembler directives (often called pseudo-ops) and assembler syntax.

as is primarily intended to assemble the output of the GNU C compiler gcc for use by the linker ld. Nevertheless, we've tried to make as assemble correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (see section Machine Dependent Features). This doesn't mean as always uses the same syntax as another assembler for the same architecture; for example, we know of several incompatible versions of 680x0 assembly language syntax.

Unlike older assemblers, as is designed to assemble a source program in one pass of the source file. This has a subtle impact on the .org directive (see section .org new-lc , fill).

Object File Formats

The GNU assembler can be configured to produce several alternative object file formats. For the most part, this does not affect how you write assembly language programs; but directives for debugging symbols are typically different in different file formats. See section Symbol Attributes.

On the machine specific, as can be configured to produce either a.out or COFF format object files. On the machine specific, as can be configured to produce either b.out or COFF format object files. On the machine specific, as can be configured to produce either SOM or ELF format object files.

Command Line

After the program name as, the command line may contain options and file names. Options may appear in any order, and may be before, after, or between file names. The order of file names is significant.

`--' (two hyphens) by itself names the standard input file explicitly, as one of the files for as to assemble.

Except for `--' any command line argument that begins with a hyphen (`-') is an option. Each option changes the behavior of as. No option changes the way another option works. An option is a `-' followed by one or more letters; the case of the letter is important. All options are optional.

Some options expect exactly one file name to follow them. The file name may either immediately follow the option's letter (compatible with older assemblers) or it may be the next command argument (GNU standard). These two command lines are equivalent:

```
as -o my-object-file.o mumble.s
as -omy-object-file.o mumble.s
```

Input Files

We use the phrase source program, abbreviated source, to describe the program input to one run of as. The program may be in one or more files; how the source is partitioned into files doesn't change the meaning of the source.

The source program is a concatenation of the text in all the files, in the order specified.

Each time you run as it assembles exactly one source program. The source program is made up of one or more files. (The standard input is also a file.)

You give as a command line that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name.

If you give as no file names it attempts to read one input file from the as standard input, which is normally your terminal. You may have to type ctl-D to tell as there is no more program to assemble.

Use `--' if you need to explicitly name the standard input file in your command line.

If the source is empty, as produces a small, empty object file.

Filenames and Line-numbers

There are two ways of locating a line in the input file (or files) and either may be used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a "logical" file. See section Error and Warning Messages.

Physical files are those files named in the command line given to as.

Logical files are simply names declared explicitly by assembler directives; they bear no relation to physical files. Logical file names help error messages reflect the original source file, when as source is itself synthesized from other files. See section .app-file string.

Output (Object) File

Every time you run as it produces an output file, which is your assembly language program translated into numbers. This file is the object file. Its default name is a.out, or b.out when as is configured for the Intel 80960. You can give it another name by using the -o option. Conventionally, object file names end with `.o'. The default name is used for historical reasons: older assemblers were capable of assembling self-contained programs directly into a runnable program. (For some formats, this isn't currently possible, but it can be done for the a.out format.)

The object file is meant for input to the linker ld. It contains assembled program code, information to help ld integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

Error and Warning Messages

as may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs as automatically. Warnings report an assumption made so that as could keep assembling a flawed program; errors report a grave problem that stops the assembly.

Warning messages have the format

```
file_name:NNN:Warning Message Text
```

(where NNN is a line number). If a logical file name has been given (see section .app-file string) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see section .line line-number) (see section .ln line-number) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (in the grand Unix tradition).

Error messages have the format

```
file_name:NNN:FATAL:Error Message Text
```

The file name and line number are derived as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

Command-Line Options

This chapter describes command-line options available in all versions of the GNU assembler; see section Machine Dependent Features, for options specific to particular machine architectures.

If you are invoking as via the GNU C compiler (version 2), you can use the `-Wa' option to pass arguments through to the assembler. The assembler arguments must be separated from each other (and the `-Wa') by commas. For example:

```
gcc -c -g -O -Wa,-alh,-L file.c
```

emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this `-Wa' mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. (You can call the GNU compiler driver with the `-v' option to see precisely what options it passes toeach compilation pass, including the assembler.)

Enable Listings: -a[dhlns]

These options enable listing output from the assembler. By itself, `-a' requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: `-ah' requests a high-level language listing, `-al' requests an output-program assembly listing, and `-as' requests a symbol table listing. High-level listings require that a compiler debugging option like `-g' be used, and that assembly listings (`-al') be requested also.

Use the `-ad' option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives `.list, .nolist, .psize, .eject, .title,` and `.sbttl`. The `-an' option turns off all forms processing. If you do not request listing output with one of the `-a' options, the listing-control directives have no effect.

The letters after `-a' may be combined into one option, e.g., `-aln'.

-D

This option has no effect whatsoever, but it is accepted to make it more likely that scripts written for other assemblers also work with as.

Work Faster: -f

`-f' should only be used when assembling programs written by a (trusted) compiler. `-f' stops the assembler from doing whitespace and comment preprocessing on the input file(s) before assembling them. See section Preprocessing.

> **Warning**: if you use `-f' when the files actually need to be preprocessed (if they contain comments, for example), as does not work correctly.

.include search path: -I path

Use this option to add a path to the list of directories as searches for files specified in .include directives (see section .include "file"). You may use -I as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, as searches any `-I' directories in the same order as they were specified (left to right) on the command line.

Difference Tables: -K

as sometimes alters the code emitted for directives of the form `.word sym1-sym2'; see section .word expressions. You can use the `-K' option if you want a warning issued when this is done.

Include Local Labels: -L

Labels beginning with `L' (upper case only) are called local labels. See section Symbol Names. Normally you do not see such labels when debugging, because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both as and ld discard such labels, so you do not normally debug with them.

This option tells as to retain those `L...' symbols in the object file. Usually if you do this you also tell the linker ld to preserve symbols whose names begin with `L'.

By default, a local label is any label beginning with `L', but each target is allowed to redefine the local label prefix. On the HPPA local labels begin with `L$'.

Assemble in MRI Compatibility Mode: -M

The -M or --mri option selects MRI compatibility mode. This changes the syntax and pseudo-op handling of as to make it compatible with the ASM68K or the ASM960 (depending upon the configured target) assembler from Microtec Research. The exact nature of the MRI syntax will not be documented here; see the MRI manuals for more information. The purpose of this option is to permit assembling existing MRI assembler code using as.

The MRI compatibility is not complete. Certain operations of the MRI assembler depend upon its object file format, and can not be supported using other object file formats. Supporting these would require enhancing each object file format individually. These are:

- global symbols in common section The m68k MRI assembler supports common sections which are merged by the linker. Other object file formats do not support this. as handles common sections by treating them as a single common symbol. It permits local symbols to be defined within a common section, but it can not support global symbols, since it has no way to describe them.

- complex relocations The MRI assemblers support relocations against a negated section address, and relocations which combine the start addresses of two or more sections. These are not support by other object file formats.

- END pseudo-op specifying start address The MRI END pseudo-op permits the specification of a start address. This is not supported by other object file formats. The start address may instead be specified using the -e option to the linker, or in a linker script.

- IDNT, .ident and NAME pseudo-ops The MRI IDNT, .ident and NAME pseudo-ops assign a module name to the output file. This is not supported by other object file formats.

- ORG pseudo-op The m68k MRI ORG pseudo-op begins an absolute section at a given address. This differs from the usual as .org pseudo-op, which changes the location within the current section. Absolute sections are not supported by other object file formats. The address of a section may be assigned within a linker script.

There are some other features of the MRI assembler which are not supported by as, typically either because they are difficult or because they seem of little consequence. Some of these may be supported in future releases.

- EBCDIC strings EBCDIC strings are not supported.

- packed binary coded decimal Packed binary coded decimal is not supported. This means that the DC.P and DCB.P pseudo-ops are not supported.

- FEQU pseudo-op The m68k FEQU pseudo-op is not supported.

- NOOBJ pseudo-op The m68k NOOBJ pseudo-op is not supported.

- OPT branch control options The m68k OPT branch control options---B, BRS, BRB, BRL, and BRW--- are ignored. as automatically relaxes all branches, whether forward or backward, to an appropriate size, so these options serve no purpose.

- OPT list control options The following m68k OPT list control options are ignored: C, CEX, CL, CRE, E, G, I, M, MEX, MC, MD, X.

- other OPT options The following m68k OPT options are ignored: NEST, O, OLD, OP, P, PCO, PCR, PCS, R.

- OPT D option is default The m68k OPT D option is the default, unlike the MRI assembler. OPT NOD may be used to turn it off.

- XREF pseudo-op. The m68k XREF pseudo-op is ignored.

- .debug pseudo-op The i960 .debug pseudo-op is not supported.

- .extended pseudo-op The i960 .extended pseudo-op is not supported.

- .list pseudo-op. The various options of the i960 .list pseudo-op are not supported.
- .optimize pseudo-op The i960 .optimize pseudo-op is not supported.
- .output pseudo-op The i960 .output pseudo-op is not supported.
- .setreal pseudo-op The i960 .setreal pseudo-op is not supported.

Name the Object File: -o

There is always one object file output when you run as. By default it has the name `a.out' (or `b.out', for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, as overwrites any existing file of the same name.

Join Data and Text Sections: -R

-R tells as to write the object file as if all data-section data lives in the text section. This is only done at the very last moment: your binary data are the same, but data section parts are relocated differently. The data section part of your object file is zero bytes long because all its bytes are appended to the text section. (See section Sections and Relocation.)

When you specify -R it would be possible to generate shorter address displacements (because we do not have to cross between text and data section). We refrain from doing this simply for compatibility with older versions of as. In future, -R may work this way.

When as is configured for COFF output, this option is only useful if you use sections named `.text' and `.data'.

-R is not supported for any of the HPPA targets. Using -R generates a warning from as.

Display Assembly Statistics: --statistics

Use `--statistics' to display two statistics about the resources used by as: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds).

Announce Version: -v

You can find out what version of as is running by including the option `-v' (which you can also spell as `-version') on the command line.

Suppress Warnings: -W

as should never give a warning or error message when assembling compiler output. But programs written by people often cause as to give a warning that a particular assumption was made. All such warnings are directed to the standard error file. If you use this option, no warnings are issued. This option only affects the warning messages: it does not change any particular of how as assembles your file. Errors, which stop the assembly, are still reported.

Generate Object File in Spite of Errors: -Z

After an error message, as normally produces no output. If for some reason you are interested in object file output even after as gives an error message on your program, use the `-Z' option. If there are any errors, as continues anyways, and writes an object file after a final warning message of the form `n errors, m warnings, generating bad object file.'

Syntax

This chapter describes the machine-independent syntax allowed in a source file. as syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler, except that as does not assemble Vax bit-fields.

Preprocessing

The as internal preprocessor:

- adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any otherwhitespace on the line into a single space.
- removes all comments, replacing them with a single space, or an appropriate number of newlines.
- converts character constants into the appropriate numeric values.

It does not do macro processing, include file handling, or anything else you may get from your C compiler's preprocessor. You can do include file processing with the .include directive (see section .include "file"). You can use the GNU C compiler driver to get other "CPP" style preprocessing, by giving the input file a `.S' suffix. See section `Options Controlling the Kind of Output' in Using GNU CC.

Excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not preprocessed.

If the first line of an input file is #NO_APP or if you use the `-f' option, whitespace and comments are not removed from the input file. Within an input file, you can ask for whitespace and comment removal in specific portions of the by putting a line that says #APP before the text that may contain whitespace or comments, and putting a line that says #NO_APP after this text. This feature is mainly intend to support asm statements in compilers whose output is otherwise free of comments and whitespace.

Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Unless within character constants (see section Character Constants), any whitespace means the same as exactly one space.

Comments

There are two ways of rendering comments to as. In both cases the comment is equivalent to one space.

Anything from `/*' through the next `*/' is a comment. This means you may not nest these comments.

```
/*
  The only way to include a newline ('\n') in a comment
```

```
  is to use this sort of comment.
*/

/* This sort of comment does not nest. */
```

Anything from the line comment character to the next newline is considered a comment and is ignored. The line comment character sequence on the MCORE family is two forware slashes // same as the C++ end of line comment.

The line comment character is `#' on the Vax; `#' on the i960; `!' on the SPARC; `|' on the 680x0; `;' for the AMD 29K family; `;' for the H8/300 family; `!' for the H8/500 family; `;' for the HPPA; `!' for the Hitachi SH; `!' for the Z8000; see section Machine Dependent Features.

On some machines there are two different line comment characters. One character only begins a comment if it is the first non-whitespace character on a line, while the other always begins a comment.

To be compatible with past assemblers, lines that begin with `#' have a special interpretation. Following the `#' should be an absolute expression (see section Expressions): the logical line number of the next line. Then a string (see section Strings) is allowed: if present it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
                            # This is an ordinary comment.
# 42-6 "new_file_name"      # New logical file name
                            # This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of as.

Symbols

A symbol is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters `_.$'. On most machines, you can also use $ in symbol names; exceptions are noted in section Machine Dependent Features. No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by the beginning of a file (since the source program must end with a newline, the end of a file is not a possible symbol delimiter). See section Symbols.

Statements

A statement ends at a newline character (`\n') or an "at" sign (`@'). The newline or at sign is considered part of the preceding statement. Newlines and at signs within character constants are an exception: they do not end statements. A statement ends at a newline character (`\n') or an exclamation point (`!'). The newline or exclamation point is considered part of the preceding statement. Newlines and exclamation points within character constants are an exception: they do not end statements. A statement ends at a newline character (`\n'); or (for the H8/300) a dollar sign (`$'); or (for the Hitachi-SH or the H8/500) a semicolon (`;'). The newline or separator character is considered part of the preceding statement. Newlines and separators within character constants are an exception: they do not end statements. A statement ends at a newline character (`\n') or line separator character. (The line separator is usually `;', unless this conflicts with the comment character; see section Machine Dependent Features.) The newline or separator character is considered part of the

preceding statement. Newlines and separators within character constants are an exception: they do not end statements.

It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

You may write a statement on more than one line if you put a backslash (\) immediately in front of any newlines within the statement. When as reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is allowed, and may include whitespace. It is ignored.

A statement begins with zero or more labels, optionally followed by a key symbol which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot `.' then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language instruction: it assembles into a machine language instruction. Different versions of as for different computers recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is a symbol immediately followed by a colon (:). Whitespace before a label or after a colon is permitted, but you may not have whitespace between a label's symbol and its colon. See section Labels.

For HPPA targets, labels need not be immediately followed by a colon, but the definition of a label must begin in column zero. This also implies that only one label may be defined on each line.

```
label:     .directive    followed by something
another_label:           # This is an empty statement.
           instruction   operand_1, operand_2, ...
```

Constants

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.
.ascii "Ring the bell\7"                  # A string constant.
.octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
.float 0f-314159265358979323846264338327\
95028841971.693993751E-40                 # - pi, a flonum.
```

Character Constants

There are two kinds of character constants. A character stands for one character in one byte and its value may be used in numeric expressions. String constants (properly called string literals) are potentially many bytes and their values may not be used in arithmetic expressions.

Strings

A string is written between double-quotes. It may contain double-quotes or null characters. The way to get special characters into a string is to escape these characters: precede them with a backslash `\' character. For example `\\' represents one backslash: the first \ is an escape which tells as to interpret the second character literally as a backslash (which prevents as from recognizing the second \ as an escape character). The complete list of escapes follows.

\b Mnemonic for backspace; for ASCII this is octal code 010.

\f Mnemonic for FormFeed; for ASCII this is octal code 014.

\n Mnemonic for newline; for ASCII this is octal code 012.

\r Mnemonic for carriage-Return; for ASCII this is octal code 015.

\t Mnemonic for horizontal Tab; for ASCII this is octal code 011.

\ digit digit digit

An octal character code. The numeric code is 3 octal digits. For compatibility with other Unix systems, 8 and 9 are accepted as digits: for example, \008 has the value 010, and \009 the value 011.

\x hex-digit hex-digit

A hex character code. The numeric code is 2 hexadecimal digits. Either upper or lower case x works.

**** Represents one `\' character.

\"

Represents one `"' character. Needed in strings to represent this character, because an unescaped `"' would end the string.

\ anything-else

Any other character when escaped by \ gives a warning, but assembles as if the `\' was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However as has no other interpretation, so as knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think the BSD 4.2 assembler recognizes, and is a subset of what most C compilers recognize. If you are in doubt, do not use an escape sequence.

Characters

A single character may be written as a single quote immediately followed by that character. The same escapes apply to characters as to strings. So if you want to write the character backslash, you must write '\\ where the first \ escapes the second \. As you can see, the quote is an acute accent, not a grave accent. A newline (or at sign `@') (or dollar sign `$', for the H8/300; or semicolon `;' for the Hitachi SH or H8/500) immediately following an acute accent is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. as assumes your character code is ASCII: 'A means 65, 'B means 66, and so on.

Number Constants

as distinguishes three kinds of numbers according to how they are stored in the target machine. Integers are numbers that would fit into an int in the C language. Bignums are integers, but they are stored in more than 32 bits. Flonums are floating point numbers, described below.

Integers

- A binary integer is `0b' or `0B' followed by zero or more of the binary digits `01'.
- An octal integer is `0' followed by zero or more of the octal digits (`01234567').
- A decimal integer starts with a non-zero digit followed by zero or more digits (`0123456789').
- A hexadecimal integer is `0x' or `0X' followed by one or more hexadecimal digits chosen from `0123456789abcdefABCDEF'.

Integers have the usual values. To denote a negative integer, use the prefix operator `-' discussed under expressions (see section Prefix Operator).

Bignums

A bignum has the same syntax and semantics as an integer except that the number (or its negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

Flonums

A flonum represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by as to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to a particular computer's floating point format (or formats) by a portion of as specialized to that computer.

A flonum is written by writing (in order)

- The digit `0'. (`0' is optional on the HPPA.)
- A letter, to tell as the rest of the number is a flonum. e is recommended. Case is not important. On the H8/300, H8/500, Hitachi SH, and AMD 29K architectures, the letter must be one of the letters `DFPRSX' (in upper or lower case). On the Intel 960 architecture, the letter must be one of the letters `DFT' (in upper or lower case). On the HPPA architecture, the letter must be `E' (upper case only).
- An optional sign: either `+' or `-'.
- An optional integer part: zero or more decimal digits.
- An optional fractional part: `.' followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - An `E' or `e'.

- Optional sign: either `+' or `-'.
- One or more decimal digits.

At least one of the integer part or the fractional part must be present. The floating point number has the usual base-10 value.

as does all processing using integers. Flonums are computed independently of any floating point hardware in the computer running as.

Sections and Relocation

Background

Roughly, a section is a range of addresses, with no gaps; all data "in" those addresses is treated the same for some particular purpose. For example there may be a "read only" section.

The linker ld reads many object files (partial programs) and combines their contents to form a runnable program. When as emits an object file, the partial program is assumed to start at address 0. ld assigns the final addresses for the partial program, so that different partial programs do not overlap. This is actually an oversimplification, but it suffices to explain how as uses sections.

ld moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a section. Assigning run-time addresses to sections is called relocation. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses. For the H8/300 and H8/500, and for the Hitachi SH, as pads sections if needed to ensure they end on a word (sixteen bit) boundary.

An object file written by as has at least three sections, any of which may be empty. These are named text, data and bss sections.

When it generates COFF output, as can also generate whatever other named sections you specify using the `.section' directive (see section .section name, subsection). If you do not use any directives that place output in the `.text' or `.data' sections, these sections still exist, but are empty.

When as generates SOM or ELF output for the HPPA, as can also generate whatever other named sections you specify using the `.space' and `.subspace' directives. See HP9000 Series 800 Assembly Language Reference Manual (HP 92432-90001) for details on the `.space' and `.subspace' assembler directives.

Additionally, as uses different names for the standard text, data, and bss sections when generating SOM output. Program text is placed into the `$CODE$' section, data into `$DATA$', and BSS into `BSS'.

Within the object file, the text section starts at address 0, the data section follows, and the bss section follows the data section.

When generating either SOM or ELF output files on the HPPA, the text section starts at address 0, the data section at address 0x4000000, and the bss section follows the data section.

To let ld know which data changes when the sections are relocated, and how to change that data, as also writes to the object file details of the relocation needed. To perform relocation ld must know, each time an address in the object file is mentioned:

- Where in the object file is the beginning of this reference to an address?
- How long (in bytes) is this reference?
- Which section does the address refer to? What is the numeric value of `(address) - (start-address of section)`?
- Is the reference to an address "Program-Counter relative"?

In fact, every address as ever uses is expressed as

```
(section) + (offset into section)
```

Further, most expressions as computes have this section-relative nature. (For some object formats, such as SOM for the HPPA, some expressions are symbol-relative instead.)

In this manual we use the notation {secname N} to mean "offset N into section secname."

Apart from text, data and bss sections you need to know about the absolute section. When ld mixes partial programs, addresses in the absolute section remain unchanged. For example, address {absolute 0} is "relocated" to run-time address 0 by ld. Although the linker never arranges two partial programs' data sections with overlapping addresses after linking, by definition their absolute sections must overlap. Address {absolute 239} in one part of a program is always the same address when the program is running as address {absolute 239} in any other part of the program.

The idea of sections is extended to the undefined section. Any address whose section is unknown at assembly time is by definition rendered {undefined U}---where U is filled in later. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has section undefined.

By analogy the word section is used to describe groups of sections in the linked program. ld puts all partial programs' text sections in contiguous addresses in the linked program. It is customary to refer to the text section of a program, meaning all the addresses of all partial programs' text sections. Likewise for data and bss sections.

Some sections are manipulated by ld; others are invented for use of as and have no meaning except during assembly.

ld Sections

ld deals with just four kinds of sections, summarized below.

named sections
text section
data section

These sections hold your program. as and ld treat them as separate but equal sections. Anything you can say of one section is true another. When the program is running, however, it is customary for the text section to be unalterable. The text section is often shared among processes: it contains instructions, constants and the like. The data section of a running program is usually alterable: for example, C variables would be stored in the data section.

bss section

This section contains zeroed bytes when your program begins running. It is used to hold unitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes in the object file. The bss section was invented to eliminate those explicit zeros from object files.

absolute section

Address 0 of this section is always "relocated" to runtime address 0. This is useful if you want to refer to an address that ld must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they do not change during relocation.

undefined section

This "section" is a catch-all for address references to objects not in the preceding sections.

An idealized example of three relocatable sections follows. The example uses the traditional section names `.text' and `.data'. Memory addresses are on the horizontal axis.

as Internal Sections

These sections are meant only for the internal use of as. They have no meaning at run-time. You do not really need to know about these sections for most purposes; but they can be mentioned in as warning messages, so it might be helpful to have an idea of their meanings to as. These sections are used to permit the value of every expression in your assembly language program to be a section-relative address.

`ASSEMBLER-INTERNAL-LOGIC-ERROR!`

An internal assembler logic error has been found. This means there is a bug in the assembler.

`expr section`

The assembler stores complex expression internally as combinations of symbols. When it needs to represent an expression as a symbol, it puts it in the expr section.

Sub-Sections

Assembled bytes conventionally fall into two sections: text and data. You may have separate groups of data in named sections text or data that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. as allows you to use subsections for this purpose. Within each section, there can be numbered subsections with values from 0 to 8192. Objects assembled into the same subsection go into the object file together with other objects in the same subsection. For example, a compiler

might want to store constants in the text section, but might not want to have them interspersed with the program being assembled. In this case, the compiler could issue a `.text 0' before each section of code being output, and a `.text 1' before each group of constants being output.

Subsections are optional. If you do not use subsections, everything goes in subsection number zero.

Each subsection is zero-padded up to a multiple of four bytes. (Subsections may be padded a different amount on different flavors of as.)

Subsections appear in your object file in numeric order, lowest numbered to highest. (All this to be compatible with other people's assemblers.) The object file contains no representation of subsections; ld and other programs that manipulate object files see no trace of them. They just see all your text subsections as a text section, and all your data subsections as a data section.

To specify which subsection you want subsequent statements assembled into, use a numeric argument to specify it, in a `.text expression' or a `.data expression' statement. When generating COFF output, you can also use an extra subsection argument with arbitrary named sections: `.section name, expression'. Expression should be an absolute expression. (See section Expressions.) If you just say `.text' then `.text 0' is assumed. Likewise `.data' means `.data 0'. Assembly begins in text 0. For instance:

```
.text 0     # The default subsection is text 0 anyway.
.ascii "This lives in the first text subsection. *"
.text 1
.ascii "But this lives in the second text subsection."
.data 0
.ascii "This lives in the data section,"
.ascii "in the first data subsection."
.text 0
.ascii "This lives in the first text section,"
.ascii "immediately following the asterisk (*)."
```

Each section has a location counter incremented by one for every byte assembled into that section. Because subsections are merely a convenience restricted to as there is no concept of a subsection location counter. There is no way to directly manipulate a location counter--but the .align directive changes it, and any label definition captures its current value. The location counter of the section where statements are being assembled is said to be the active location counter.

bss Section

The bss section is used for local common variable storage. You may allocate address space in the bss section, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the bss section are zeroed bytes.

Addresses in the bss section are allocated with special directives; you may not assemble anything directly into the bss section. Hence there are no bss subsections. See section .comm symbol , length , see section .lcomm symbol , length.

Symbols

Symbols are a central concept: the programmer uses symbols to name things, the linker uses symbols to link, and the debugger uses symbols to debug.

> Warning: as does not place symbols in the object file in the same order they were declared. This may break some debuggers.

Labels

A label is written as a symbol immediately followed by a colon `:'. The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

On the HPPA, the usual form for a label need not be immediately followed by a colon, but instead must start in column zero. Only one label may be defined on a single line. To work around this, the HPPA version of as also provides a special directive .label for defining labels more flexibly.

Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol, followed by an equals sign `=', followed by an expression (see section Expressions). This is equivalent to using the .set directive. See section .set symbol, expression.

Symbol Names

Symbol names begin with a letter or with one of `._'. On most machines, you can also use $ in symbol names; exceptions are noted in section Machine Dependent Features. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in section Machine Dependent Features), and underscores. For the AMD 29K family, `?' is also allowed in the body of a symbol name, though not at its beginning.

Case of letters is significant: foo is a different symbol name than Foo.

Each symbol has exactly one name. Each name in an assembly language program refers to exactly one symbol. You may use that symbol name any number of times in a program.

Local Symbol Names

Local symbols help compilers and programmers use names temporarily. There are ten local symbol names, which are re-used throughout the program. You may refer to them using the names `0' `1' ... `9'. To define a local symbol, write a label of the form `N:' (where N represents any digit). To refer to the most recent previous definition of that symbol write `Nb', using the same digit as when you defined the label. To refer to the next definition of a local label, write `Nf'---where N gives you a choice of 10 forward references. The `b' stands for "backwards" and the `f' stands for "forwards".

Local symbols are not emitted by the current GNU C compiler.

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler uses them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have these parts:

`L`

All local labels begin with `L'. Normally both as and ld forget symbols that start with `L'. These labels are used for symbols you are never intended to see. If you use the `-L' option then as retains these symbols in the object file. If you also instruct ld to retain these symbols, you may use them in debugging.

`digit`

If the label is written `0:' then the digit is `0'. If the label is written `1:' then the digit is `1'. And so on up through `9:'.

`^A`

This unusual character is included so you do not accidentally invent a symbol of the same name. The character has ASCII value `\001'.

`ordinal number`

This is a serial number to keep the labels distinct. The first `0:' gets the number `1'; The 15th `0:' gets the number `15'; etc.. Likewise for the other labels `1:' through `9:'.

For instance, the first 1: is named L1^A1, the 44th 3: is named L3^A44.

The Special Dot Symbol

The special symbol `.' refers to the current address that as is assembling into. Thus, the expression `melvin: .long .' defines melvin to contain its own address. Assigning a value to . is treated the same as a .org directive. Thus, the expression `.=.+4' is the same as saying `.space 4'.

Symbol Attributes

Every symbol has, as well as its name, the attributes "Value" and "Type". Depending on output format, symbols can also have auxiliary attributes.

If you use a symbol without defining it, as assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

- Symbol Value: Value
- Symbol Type: Type
- a.out Symbols: Symbol Attributes: a.out
- a.out Symbols: Symbol Attributes: a.out, b.out
- COFF Symbols: Symbol Attributes for COFF
- SOM Symbols: Symbol Attributes for SOM

Value

The value of a symbol is (usually) 32 bits. For a symbol which labels a location in the text, data, bss or absolute sections the value is the number of addresses from the start of that section to the label. Naturally for text, data and bss sections the value of a symbol changes as ld changes section base addresses during linking. Absolute symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source file, and ld tries to determine its value from other files linked into the same program. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a .comm common declaration. The value is how much common storage to reserve, in bytes (addresses). The symbol refers to the first address of the allocated storage.

Type

The type attribute of a symbol contains relocation (section) information, any flag settings indicating that a symbol is external, and (optionally), other information for linkers and debuggers. The exact format depends on the object-code output format in use.

Symbol Attributes: a.out

Symbol Desc: Descriptor

> This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a .desc statement (see section .desc symbol, abs-expression). A descriptor value means nothing to as.

Symbol Other: Other

> This is an arbitrary 8-bit value. It means nothing to as.

Symbol Attributes for COFF

The COFF format supports a multitude of auxiliary symbol attributes; like the primary symbol attributes, they are set between .def and .endef directives.

Primary Attributes

> The symbol name is set with .def; the value and type, respectively, with .val and .type.

Auxiliary Attributes

> The as directives .dim, .line, .scl, .size, and .tag can generate auxiliary symbol table information for COFF.

Symbol Attributes for SOM

The SOM format for the HPPA supports a multitude of symbol attributes set with the .EXPORT and .IMPORT directives.

The attributes are described in HP9000 Series 800 Assembly Language Reference Manual (HP 92432-90001) under the IMPORT and EXPORT assembler directive documentation.

Expressions

An expression specifies an address or numeric value. Whitespace may precede and/or follow an expression.

The result of an expression must be an absolute number, or else an offset into a particular section. If an expression is not absolute, and there is not enough information when as sees the expression to know its section, a second pass over the source program might be necessary to interpret the expression--but the second pass is currently not implemented. as aborts with an error message in this situation.

Empty Expressions

An empty expression has no value: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression, and as assumes a value of (absolute) 0. This is compatible with other assemblers.

Integer Expressions

An integer expression is one or more arguments delimited by operators.

Arguments

Arguments are symbols, numbers or subexpressions. In other contexts arguments are sometimes called "arithmetic operands". In this manual, to avoid confusing them with the "instruction operands" of the machine language, we use the term "argument" to refer to parts of expressions only, reserving the word "operand" to refer only to machine instruction operands.

Symbols are evaluated to yield {section NNN} where section is one of text, data, bss, absolute, or undefined. NNN is a signed, 2's complement 32 bit integer.

Numbers are usually integers.

A number can be a flonum or bignum. In this case, you are warned that only the low order 32 bits are used, and as pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Subexpressions are a left parenthesis `(' followed by an integer expression, followed by a right parenthesis `)'; or a prefix operator followed by an argument.

Operators

Operators are arithmetic functions, like + or %. Prefix operators are followed by an argument. Infix operators appear between their arguments. Operators may be preceded and/or followed by whitespace.

Prefix Operator

as has the following prefix operators. They each take one argument, which must be absolute.

- Negation. Two's complement negation.

~ Complementation. Bitwise not.

Infix Operators

Infix operators take two arguments, one on either side. Operators have precedence, but operations with equal precedence are performed left to right. Apart from + or -, both arguments must be absolute, and the result is absolute.

1. Highest Precedence

* Multiplication.

/ Division. Truncation is the same as the C operator `/'

% Remainder.

<

<< Shift Left. Same as the C operator `<<'.

>

>> Shift Right. Same as the C operator `>>'.

2. Intermediate precedence

| Bitwise Inclusive Or.

& Bitwise And.

^ Bitwise Exclusive Or.

! Bitwise Or Not.

3. Lowest Precedence

+ Addition. If either argument is absolute, the result has the section of the other argument. You may not add together arguments from different sections.

- Subtraction. If the right argument is absolute, the result has the section of the left argument. If both arguments are in the same section, the result is absolute. You may not subtract arguments from different sections.

In short, it's only meaningful to add or subtract the offsets in an address; you can only have a defined section in one of the two arguments.

Assembler Directives

All assembler directives have names that begin with a period (`.'). The rest of the name is letters, usually in lower case.

This chapter discusses directives that are available regardless of the target machine configuration for the GNU assembler. Some machine configurations provide additional directives. See section Machine Dependent Features.

.abort	This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembly language source would be piped into the assembler. If the sender of the source quit, it could use this directive tells as to quit also. One day .abort will not be supported.
.ABORT	When producing COFF output, as accepts this directive as a synonym for `.abort'. When producing b.out output, as accepts this directive, but ignores it.
.align `abs-expr, abs-expr`	Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment required, as described below. The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero. The way the required alignment is specified varies from system to system. For the a29k, hppa, m86k, m88k, w65, sparc, and Hitachi SH, and i386 using ELF format, the first expression is the alignment request in bytes. For example `.align 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. For other systems, including the i386 using a.out format, it is the number of low-order zero bits the location counter must have after advancement. For example `.align 3' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed. This inconsistency is due to the different behaviors of the various native assemblers for these systems which GAS must emulate. GAS also provides .balign and .p2align directives, described later, which have a consistent behavior across all architectures (but are specific to GAS).
.app-file `string`	.app-file (which may also be spelled `.file') tells as that we are about to start a new logical file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes `"'; but if you wish to specify an empty file name is permitted, you must give the quotes--"". This statement may go away in future: it is only recognized to be compatible with old as programs.
.ascii `"string" ...`	.ascii expects zero or more string literals (see section Strings) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

.asciz "string" ...	.asciz is just like .ascii, but each string is followed by a zero byte. The "z" in `.asciz' stands for "zero".
.balign abs-expr, abs-expr	Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the alignment request in bytes. For example `.balign 8' advances the location counter until it is a multiple of 8. If the location counter is already a multiple of 8, no change is needed. The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero.
.byte expressions	.byte expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.
.comm symbol, length	.comm declares a named common area in the bss section. Normally ld reserves memory addresses for it during linking, so no partial program defines the location of the symbol. Use .comm to tell ld that it must be at least length bytes long. ld allocates space for each .comm symbol that is at least as long as the longest .comm request in any of the partial programs linked. length is an absolute expression. The syntax for .comm differs slightly on the HPPA. The syntax is `symbol .comm, length'; symbol is optional.

.data subsection	.data tells as to assemble the following statements onto the end of the data subsection numbered subsection (which is an absolute expression). If subsection is omitted, it defaults to zero.
.def name	Begin defining debugging information for a symbol name; the definition extends until the .endef directive is encountered. This directive is only observed when as is configured for COFF format output; when producing b.out, `.def' is recognized, but ignored.
.desc symbol, abs-expr	This directive sets the descriptor of the symbol (see section Symbol Attributes) to the low 16 bits of an absolute expression. The `.desc' directive is not available when as is configured for COFF output; it is only for a.out or b.out object format. For the sake of compatibility, as accepts it, but produces no output, when configured for COFF.
.dim	This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside .def/.endef pairs. `.dim' is only meaningful when generating COFF format output; when as is generating b.out, it accepts this directive but ignores it.
.double flonums	.double expects zero or more flonums, separated by commas. It assembles floating point numbers. The exact kind of floating point numbers emitted depends on how as is configured. See section Machine Dependent Features.
.eject	Force a page break at this point, when generating assembly listings.
.else	.else is part of the as support for conditional assembly; see section .if absolute expression. It marks the beginning of a section of code to be assembled if the condition for the preceding .if was false.
.endef	This directive flags the end of a symbol definition begun with .def. `.endef' is only meaningful when generating COFF format output; if as is configured to generate b.out, it accepts this directive but ignores it.
.endif	.endif is part of the as support for conditional assembly; it marks the end of a block of code that is only assembled conditionally. See section .if absolute expression.
.equ symbol, expression	This directive sets the value of symbol to expression. It is synonymous with `.set'; see section .set symbol, expression.  The syntax for equ on the HPPA is `symbol .equ expression'.
.extern	.extern is accepted in the source program—for compatibility with other assemblers--but it is ignored. as treats all undefined symbols as external.
.file	.file (which may also be spelled `.app-file') tells as that we are about to start a new logical

`string`	file. string is the new file name. In general, the filename is recognized whether or not it is surrounded by quotes `""; but if you wish to specify an empty file name, you must give the quotes--"". This statement may go away in future: it is only recognized to be compatible with old as programs. In some configurations of as, .file has already been removed to avoid conflicts with other assemblers. See section Machine Dependent Features.
.fill `repeat , size , value`	repeat, size and value are absolute expressions. This emits repeat copies of size bytes. Repeat may be zero or more. Size may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are value rendered in the byte-order of an integer on the computer as is assembling for. Each size bytes in a repetition is taken from the lowest order size bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers. size and value are optional. If the second comma and value are absent, value is assumed zero. If the first comma and following tokens are absent, size is assumed to be 1.
.float `flonums`	This directive assembles zero or more flonums, separated by commas. It has the same effect as .single. The exact kind of floating point numbers emitted depends on how as is configured. See section Machine Dependent Features.
.global `symbol`, **.globl** `symbol`	.global makes the symbol visible to ld. If you define symbol in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, symbol takes its attributes from a symbol of the same name from another file linked into the same program. Both spellings (`.globl' and `.global') are accepted, for compatibility with other assemblers. On the HPPA, .global is not always enough to make it accessible to other partial programs. You may need the HPPA-only .EXPORT directive as well. See section HPPA Assembler Directives.
.hword `expressions`	This expects zero or more expressions, and emits a 16 bit number for each. This directive is a synonym for `.short'; depending on the target architecture, it may also be a synonym for `.word'.
.ident	This directive is used by some assemblers to place tags in object files. as simply accepts the directive for source-file compatibility with such assemblers, but does not actually emit anything for it.
.if `absolute expression`	.if marks the beginning of a section of code which is only considered part of the source program being assembled if the argument (which must be an absolute expression) is non-zero. The end of the conditional section of code must be marked by .endif (see section .endif); optionally, you may include code for the alternative condition, flagged by .else (see section .else. The following variants of .if are also supported: `.ifdef symbol`

	Assembles the following section of code if the specified symbol has been defined.
	
	`.ifndef symbol`
	`ifnotdef symbol`
	
	Assembles the following section of code if the specified symbol has not been defined. Both spelling variants are equivalent.

`.include` `"file"`	This directive provides a way to include supporting files at specified points in your source program. The code from file is assembled as if it followed the point of the .include; when the end of the included file is reached, assembly of the original file continues. You can control the search paths used with the `-I' command-line option (see section Command-Line Options). Quotation marks are required around file.
`.int` `expressions`	Expect zero or more expressions, of any section, separated by commas. For each expression, emit a number that, at run time, is the value of that expression. The byte order and bit size of the number depends on what kind of target the assembly is for.

`.irp` `symbol, values...`	Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the .irp directive, and is terminated by an .endr directive. For each value, symbol is set to value, and the sequence of statements is assembled. If no value is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to symbol within the sequence of statements, use \symbol.
	
	For example, assembling
	`.irp param,1,2,3`
	`move d\param,sp@-`
	`.endr`
	is equivalent to assembling
	`move d1,sp@-`
	`move d2,sp@-`
	`move d3,sp@-`

`.irpc` `symbol, values...`	Evaluate a sequence of statements assigning different values to symbol. The sequence of statements starts at the .irpc directive, and is terminated by an .endr directive. For each character in value, symbol is set to the character, and the sequence of statements is assembled. If no value is listed, the sequence of statements is assembled once, with symbol set to the null string. To refer to symbol within the sequence of statements, use \symbol.
	
	For example, assembling
	`.irpc param,123`
	`move d\param,sp@-`
	`.endr`
	
	is equivalent to assembling
	`move d1,sp@-`
	`move d2,sp@-`
	`move d3,sp@-`

`.lcomm`	Reserve length (an absolute expression) bytes for a local common denoted by symbol. The

`symbol , length`	section and value of symbol are those of the new local common. The addresses are allocated in the bss section, so that at run-time the bytes start off zeroed. Symbol is not declared global (see section .global symbol, .globl symbol), so is normally not visible to ld. The syntax for .lcomm differs slightly on the HPPA. The syntax is `symbol .lcomm, length'; symbol is optional.
`.lflags`	as accepts this directive, for compatibility with other assemblers, but ignores it.
`.line` `line-number` `or` **`.ln`** `line-number`	Change the logical line number. line-number must be an absolute expression. The next line has that logical line number. Therefore any other statements on the current line (after a statement separator character) are reported as on logical line number line-number - 1. One day as will no longer support this directive: it is recognized only for compatibility with existing assembler programs. Warning: In the AMD29K configuration of as, this command is not available; use the synonym .ln in that context. Even though this is a directive associated with the a.out or b.out object-code formats, as still recognizes it when producing COFF output, and treats `.line' as though it were the COFF `.ln' if it is found outside a .def/.endef pair. Inside a .def, `.line' is, instead, one of the directives used by compilers to generate auxiliary symbol information for debugging.
`.list`	Control (in conjunction with the .nolist directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). .list increments the counter, and .nolist decrements it. Assembly listings are generated whenever the counter is greater than zero. By default, listings are disabled. When you enable them (with the `-a' command line option; see section Command-Line Options), the initial value of the listing counter is one.
`.long` `expressions`	.long is the same as `.int', see section .int expressions.
`.macro`	The commands .macro and .endm allow you to define macros that generate assembly output. For example, this definition specifies a macro sum that puts a sequence of numbers into memory: `.macro sum from=0, to=5` `.long \from` `.if \to-\from` `sum "(\from+1)",\to` `.endif` `.endm` With that definition, `SUM 0,5' is equivalent to this assembly input: `.long 0` `.long 1` `.long 2` `.long 3` `.long 4`

	`.long 5` **.macro macname** **.macro macname macargs ...** Begin the definition of a macro called macname. If your macro definition requires arguments, specify their names after the macro name, separated by commas or spaces. You can supply a default value for any macro argument by following the name with `=deflt'. For example, these are all valid .macro statements: **.macro comm** Begin the definition of a macro called comm, which takes no arguments. **.macro plus1 p, p1** **.macro plus1 p p1** Either statement begins the definition of a macro called plus1, which takes two arguments; within the macro definition, write `\p' or `\p1' to evaluate the arguments. **.macro reserve_str p1=0 p2** Begin the definition of a macro called reserve_str, with two arguments. The first argument has a default value, but not the second. After the definition is complete, you can call the macro either as `reserve_str a,b' (with `\p1' evaluating to a and `\p2' evaluating to b), or as `reserve_str ,b' (with `\p1' evaluating as the default, in this case `0', and `\p2' evaluating to b).  When you call a macro, you can specify the argument values either by position, or by keyword. For example, `sum 9,17' is equivalent to `sum to=17, from=9'.  **.endm** Mark the end of a macro definition.  **.exitm** Exit early from the current macro definition.  **\@**  as maintains a counter of how many macros it has executed in this pseudo-variable; you can copy that number to your output with `\@', but only within a macro definition.
.nolist	Control (in conjunction with the .list directive) whether or not assembly listings are generated. These two directives maintain an internal counter (which is zero initially). .list increments the counter, and .nolist decrements it. Assembly listings are generated whenever the counter is greater than zero.
.octa bignums	This directive expects zero or more bignums, separated by commas. For each bignum, it emits a 16-byte integer. The term "octa" comes from contexts in which a "word" is two bytes; hence octa-word for 16 bytes.

.org new-lc , fill	Advance the location counter of the current section to new-lc. new-lc is either an absolute expression or an expression with the same section as the current subsection. That is, you can't use .org to cross sections: if new-lc has the wrong section, the .org directive is ignored. To be compatible with former assemblers, if the section of new-lc is absolute, as issues a warning, then pretends the section of new-lc is the same as the current subsection. .org may only increase the location counter, or leave it unchanged; you cannot use .org to move the location counter backwards. Because as tries to assemble programs in one pass, new-lc may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler. Beware that the origin is relative to the start of the section, not to the start of the subsection. This is compatible with other people's assemblers. When the location counter (of the current subsection) is advanced, the intervening bytes are filled with fill which should be an absolute expression. If the comma and fill are omitted, fill defaults to zero.
.p2align abs-expr , abs-expr	Pad the location counter (in the current subsection) to a particular storage boundary. The first expression (which must be absolute) is the number of low-order zero bits the location counter must have after advancement. For example `.p2align 3' advances the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed. The second expression (also absolute) gives the value to be stored in the padding bytes. It (and the comma) may be omitted. If it is omitted, the padding bytes are zero.
.psize lines , columns	Use this directive to declare the number of lines--and, optionally, the number of columns--to use for each page, when generating listings. If you do not use .psize, listings use a default line-count of 60. You may omit the comma and columns specification; the default width is 200 columns. as generates formfeeds whenever the specified number of lines is exceeded (or whenever you explicitly request one, using .eject). If you specify lines as 0, no formfeeds are generated save those explicitly specified with .eject.
.quad bignums	.quad expects zero or more bignums, separated by commas. For each bignum, it emits an 8-byte integer. If the bignum won't fit in 8 bytes, it prints a warning message; and just takes the lowest order 8 bytes of the bignum. The term "quad" comes from contexts in which a "word" is two bytes; hence quad-word for 8 bytes.

.rept count	Repeat the sequence of lines between the .rept directive and the next .endr directive count times.
	
	For example, assembling
	`.rept 3`
	`.long 0`

	`.endr`
	
	is equivalent to assembling
	`.long 0`
	`.long 0`
	`.long 0`

`.sbttl` `"subheading"`	Use subheading as the title (third line, immediately after the title line) when generating assembly listings. This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.
`.scl` `class`	Set the storage-class value for a symbol. This directive may only be used inside a .def/.endef pair. Storage class may flag whether a symbol is static or external, or it may record further symbolic debugging information. The `.scl' directive is primarily associated with COFF output; when configured to generate b.out output format, as accepts this directive but ignores it.
`.section` `name, subsection`	Assemble the following code into end of subsection numbered subsection in the COFF named section name. If you omit subsection, as uses subsection number zero. `.section .text' is equivalent to the .text directive; `.section .data' is equivalent to the .data directive. This directive is only supported for targets that actually support arbitrarily named sections; on a.out targets, for example, it is not accepted, even with a standard a.out section name as its parameter.
`.set` `symbol, expression`	Set the value of symbol to expression. This changes symbol's value and type to conform to expression. If symbol was flagged as external, it remains flagged. (See section Symbol Attributes.) You may .set a symbol many times in the same assembly. If you .set a global symbol, the value stored in the object file is the last value stored into it. The syntax for set on the HPPA is `symbol .set expression'.
`.short` `expressions`	.short is normally the same as `.word'. See section .word expressions. In some configurations, however, .short and .word generate numbers of different lengths; see section Machine Dependent Features.
`.single` `flonums`	This directive assembles zero or more flonums, separated by commas. It has the same effect as .float. The exact kind of floating point numbers emitted depends on how as is configured. See section Machine Dependent Features.
`.size`	This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside .def/.endef pairs. `.size' is only meaningful when generating COFF format output; when as is generating b.out, it accepts this directive but ignores it.
`.space`	This directive emits size bytes, each of value fill. Both size and fill are absolute expressions.

`size , fill`	If the comma and fill are omitted, fill is assumed to be zero. Warning: In most versions of the GNU assembler, the directive .space has the effect of .block See section Machine Dependent Features.

`.stabd, .stabn, .stabs`	There are three directives that begin `.stab'. All emit symbols (see section Symbols), for use by symbolic debuggers. The symbols are not entered in the as hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:
	
	string
	
	This is the symbol's name. It may contain any character except `\000', so is more general than ordinary symbol names. Some debuggers used to code arbitrarily complex structures into symbol names using this field.
	
	type
	
	An absolute expression. The symbol's type is set to the low 8 bits of this expression. Any bit pattern is permitted, but ld and debuggers choke on silly bit patterns.
	
	other
	
	An absolute expression. The symbol's "other" attribute is set to the low 8 bits of this expression.
	
	desc
	
	An absolute expression. The symbol's descriptor is set to the low 16 bits of this expression.
	
	value
	
	An absolute expression which becomes the symbol's value.
	
	If a warning is detected while reading a .stabd, .stabn, or .stabs statement, the symbol has probably already been created; you get a half-formed symbol in your object file. This is compatible with earlier assemblers!
	
	`.stabd type , other , desc`
	
	The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings. The symbol's value is set to the location counter, relocatably. When your program is linked, the value of this symbol is the address of the location counter when the .stabd was assembled.
	
	`.stabn type , other , desc , value`
	
	The name of the symbol is set to the empty string "".
	
	`.stabs string , type , other , desc , value`

	All five fields are specified.
.string "str"	Copy the characters in str to the object file. You may specify more than one string to copy, separated by commas. Unless otherwise specified for a particular machine, the assembler marks the end of each string with a 0 byte. You can use any of the escape sequences described in section Strings.
.tag structname	This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside .def/.endef pairs. Tags are used to link structure definitions in the symbol table with instances of those structures. `.tag' is only used when generating COFF format output; when as is generating b.out, it accepts this directive but ignores it.
.text subsection	Tells as to assemble the following statements onto the end of the text subsection numbered subsection, which is an absolute expression. If subsection is omitted, subsection number zero is used.
.title "heading"	Use heading as the title (second line, immediately after the source file name and pagenumber) when generating assembly listings. This directive affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.
.type int	This directive, permitted only within .def/.endef pairs, records the integer int as the type attribute of a symbol table entry. `.type' is associated only with COFF format output; when as is configured for b.out output, it accepts this directive but ignores it.
.val addr	This directive, permitted only within .def/.endef pairs, records the address addr as the value attribute of a symbol table entry. `.val' is used only for COFF output; when as is configured for b.out, it accepts this directive but ignores it.
.word expressions	This directive expects zero or more expressions, of any section, separated by commas. The size of the number emitted, and its byte order, depend on what target computer the assembly is for. Warning: Special Treatment to support Compilers Machines with a 32-bit address space, but that do less than 32-bit addressing, require the following special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it; see section Machine Dependent Features), you can ignore this issue. In order to assemble compiler output into something that works, as occasionlly does strange things to `.word' directives. Directives of the form `.word sym1-sym2' are often emitted by compilers as part of jump tables. Therefore, when as assembles a directive of the form

	`.word sym1-sym2', and the difference between sym1 and sym2 does not fit in 16 bits, as creates a secondary jump table, immediately before the next label. This secondary jump table is preceded by a short-jump to the first byte after the secondary table. This short-jump prevents the flow of control from accidentally falling into the new table. Inside the table is a long-jump to sym2. The original `.word' contains sym1 minus the address of the long-jump to sym2.  If there were several occurrences of `.word sym1-sym2' before the secondary jump table, all of them are adjusted. If there was a `.word sym3-sym4', that also did not fit in sixteen bits, a long-jump to sym4 is included in the secondary jump table, and the .word directives are adjusted to contain sym3 minus the address of the long-jump to sym4; and so on, for as many entries in the original jump table as necessary.

Machine Dependent Features

The machine instruction sets are (almost by definition) different on each machine where as runs. Floating point representations vary as well, and as often supports a few additional directives or command-line options for compatibility with other assemblers on a particular platform. Finally, some versions of as support special pseudo-instructions for branch optimization.

This chapter discusses most of these differences, though it does not include details on any machine's instruction set. For details on that subject, see the hardware manufacturer's manual.

M680x0 Options

The Motorola 680x0 version of as has a few machine dependent options. You can use the `-l' option to shorten the size of references to undefined symbols. If you do not use the `-l' option, references to undefined symbols are wide enough for a full long (32 bits). (Since as cannot know where these symbols end up, as can only allocate space for the linker to fill in later. Since as does not know how far away these symbols are, it allocates as much space as it can.) If you use this option, the references are only one word wide (16 bits). This may be useful if you want the object file to be as small as possible, and you know that the relevant symbols are always less than 17 bits away.

For some configurations, especially those where the compiler normally does not prepend an underscore to the names of user variables, the assembler requires a `%' before any use of a register name. This is intended to let the assembler distinguish between C variables and functions named `a0' through `a7', and so on. The `%' is always accepted, but is not required for certain configurations, notably `sun3'. The `--register-prefix-optional' option may be used to permit omitting the `%' even for configurations for which it is normally required. If this is done, it will generally be impossible to refer to C variables and functions with the same names as register names.

as can assemble code for several different members of the Motorola 680x0 family. The default depends upon how as was configured when it was built; normally, the default is to assemble code for the 68020 microprocessor. The following options may be used to change the default. These options control which instructions and addressing modes are permitted. The members of the 680x0 family are very similar. For detailed information about the differences, see the Motorola manuals.

`-m68000' `-m68008' `-m68302'

Assemble for the 68000. `-m68008' and `-m68302' are synonyms for `-m68000', since the chips are the same from the point of view of the assembler.

`-m68010' Assemble for the 68010.

`-m68020' Assemble for the 68020. This is normally the default.

`-m68030' Assemble for the 68030.

`-m68040' Assemble for the 68040.

`-m68060' Assemble for the 68060.

`-mcpu32' `-m68331' `-m68332' `-m68333' `-m68340' `-m68360'

Assemble for the CPU32 family of chips.

`-m68881' `-m68882'

Assemble 68881 floating point instructions. This is the default for the 68020, 68030, and the CPU32. The 68040 and 68060 always support floating point instructions.

`-mno-68881'

Do not assemble 68881 floating point instructions. This is the default for 68000 and the 68010. The 68040 and 68060 always support floating point instructions, even if this option is used.

`-m68851'

Assemble 68851 MMU instructions. This is the default for the 68020, 68030, and 68060. The 68040 accepts a somewhat different set of MMU instructions; `-m68851' and `-m68040' should not be used together.

`-mno-68851'

Do not assemble 68851 MMU instructions. This is the default for the 68000, 68010, and the CPU32. The 68040 accepts a somewhat different set of MMU instructions.

Syntax

This syntax for the Motorola 680x0 was developed at MIT.

The 680x0 version of as uses instructions names and syntax compatible with the Sun assembler. Intervening periods are ignored; for example, `movl' is equivalent to `mov.l'.

In the following table apc stands for any of the address registers (`%a0' through `%a7'), the program counter (`%pc'), the zero-address relative to the program counter (`%zpc'), a suppressed address register (`%za0' through `%za7'), or it may be omitted entirely. The use of size means one of `w' or `l', and it may be omitted, along with the leading colon, unless a scale is also specified. The use of scale means one of `1', `2', `4', or `8', and it may always be omitted along with the leading colon.

The following addressing modes are understood:

Immediate	`#number'
Data Register	`%d0' through `%d7'
Address Register	`%a0' through `%a7' `%a7' is also known as `%sp', i.e. the Stack Pointer. %a6 is also known as `%fp', the Frame Pointer.
Address Register Indirect	`%a0@' through `%a7@'
Address Register Postincrement	`%a0@+' through `%a7@+'
Address Register Predecrement	`%a0@-' through `%a7@-'
Indirect Plus Offset	`apc@(number)'
Index	`apc@(number,register:size:scale)' The number may be omitted.
Postindex	`apc@(number)@(onumber,register:size:scale)' The onumber or the register, but not both, may be omitted.
Preindex	`apc@(number,register:size:scale)@(onumber)' The number may be omitted. Omitting the register produces the Postindex addressing mode.
Absolute	`symbol', or `digits', optionally followed by `:b', `:w', or `:l'.

Motorola Syntax

The standard Motorola syntax for this chip differs from the syntax already discussed (see section Syntax). as can accept Motorola syntax for operands, even if MIT syntax is used for other operands in the same instruction. The two kinds of syntax are fully compatible.

In the following table apc stands for any of the address registers (`%a0' through `%a7'), the program counter (`%pc'), the zero-address relative to the program counter (`%zpc'), or a suppressed address register (`%za0' through `%za7'). The use of size means one of `w' or `l', and it may always be omitted along with the leading dot. The use of scale means one of `1', `2', `4', or `8', and it may always be omitted along with the leading asterisk.

The following additional addressing modes are understood:

Address Register Indirect	`(%a0)' through `(%a7)' `%a7' is also known as `%sp', i.e. the Stack Pointer. %a6 is also known as `%fp', the Frame Pointer.

Address Register Postincrement	`(%a0)+' through `(%a7)+'
Address Register Predecrement	`-(%a0)' through `-(%a7)'
Indirect Plus Offset	`number(%a0)' through `number(%a7)', or `number(%pc)'. The number may also appear within the parentheses, as in `(number,%a0)'. When used with the pc, the number may be omitted (with an address register, omitting the number produces Address Register Indirect mode).
Index	`number(apc,register.size*scale)' The number may be omitted, or it may appear within the parentheses. The apc may be omitted. The register and the apc may appear in either order. If both apc and register are address registers, and the size and scale are omitted, then the first register is taken as the base register, and the second as the index register.
Postindex	`([number,apc],register.size*scale,onumber)' The onumber, or the register, or both, may be omitted. Either the number or the apc may be omitted, but not both.
Preindex	`([number,apc,register.size*scale],onumber)' The number, or the apc, or the register, or any two of them, may be omitted. The onumber may be omitted. The register and the apc may appear in either order. If both apc and register are address registers, and the size and scale are omitted, then the first register is taken as the base register, and the second as the index register.

Floating Point

Packed decimal (P) format floating literals are not supported. Feel free to add the code!

The floating point formats generated by directives are these.

```
.float      Single precision floating point constants.
.double     Double precision floating point constants.
```

There is no directive to produce regions of memory holding extended precision numbers, however they can be used as immediate operands to floating-point instructions. Adding a directive to create extended precision numbers would not be hard, but it has not yet seemed necessary.

680x0 Machine Directives

In order to be compatible with the Sun assembler the 680x0 assembler understands the following directives.

`.data1` This directive is identical to a .data 1 directive.

`.data2` This directive is identical to a .data 2 directive.

`.even` This directive is a special case of the .align directive; it aligns the output to an even byte boundary.

`.skip` This directive is identical to a .space directive.