sources:gnu-doc/guide/gas

COLLABORATORS

	TITLE : sources:gnu-doc/guide/gas	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 16, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

sources:gnu-doc/guide/gas

1.1 sources:gnu-doc/guide/gas.guide

```
                Using {No Value For "AS"}
*************************

   This file is a user guide to the GNU assembler `{No Value For "AS"}'.
This version of the file describes `{No Value For "AS"}' configured to
generate code for {No Value For "TARGET"} architectures.



                Overview
                                    Overview

                Invoking
                                    Command-Line Options

                Syntax
                                      Syntax

                Sections
                                    Sections and Relocation

                Symbols
                                     Symbols

                Expressions
                                  Expressions

                Pseudo Ops
                                     Assembler Directives
  Machine Dependencies          Machine Dependent Features
  Acknowledgements              Who Did What
  Index                         Index
```

1.2 gas.guide/Overview

```
                Overview
********

   Here is a brief summary of how to invoke `{No Value For "AS"}'.  For
details, see
                Comand-Line Options
                .

     {No Value For "AS"} [ -a[dhlns] ] [ -D ] [ -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

     You may combine these options; for example, use `-aln' for assembly
     listing without forms processing.  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.

`-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'
```

```
     This option is accepted but has no effect on the {No Value For
     "TARGET"} family.

‘-L’
     Keep (in the symbol table) local symbols, starting with ‘L’.

‘-o OBJFILE’
     Name the object-file output from ‘{No Value For "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’
‘-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.



                Manual
                                      Structure of this Manual

                GNU Assembler
                               {No Value For "AS"}, the GNU Assembler

                Object Formats
                              Object File Formats

                Command Line
                               Command Line

                Input Files
                                Input Files

                Object
                                      Output (Object) File
```

Errors
Error and Warning Messages

1.3 gas.guide/Manual

Structure of this Manual
========================

This manual is intended to describe what you need to know to use GNU ‘{No Value For "AS"}’. We cover the syntax expected in source files, including notation for symbols, constants, and expressions; the directives that ‘{No Value For "AS"}’ understands; and of course how to invoke ‘{No Value For "AS"}’.

We also cover special features in the {No Value For "TARGET"} configuration of ‘{No Value For "AS"}’, including assembler directives.

On the other hand, this manual is *not* intended as an introduction to programming in assembly language--let alone programming in general! In a similar vein, we make no attempt to introduce the machine architecture; we do *not* describe the instruction set, standard mnemonics, registers or addressing modes that are standard to a particular architecture.

1.4 gas.guide/GNU Assembler

{No Value For "AS"}, the GNU Assembler
======================================

GNU ‘as’ is really a family of assemblers. This manual describes ‘{No Value For "AS"}’, a member of that family which is configured for the {No Value For "TARGET"} architectures. 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.

‘{No Value For "AS"}’ is primarily intended to assemble the output of the GNU C compiler ‘{No Value For "GCC"}’ for use by the linker ‘{No Value For "LD"}’. Nevertheless, we’ve tried to make ‘{No Value For "AS"}’ assemble correctly everything that other assemblers for the same machine would assemble.

Unlike older assemblers, ‘{No Value For "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 ‘.org’).

1.5 gas.guide/Object Formats

```
                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

```
                Symbol Attributes
```

. On the {No Value For "TARGET"}, ‘{No Value For "AS"}’ is configured to produce {No Value For "OBJ-NAME"} format object files.

1.6 gas.guide/Command Line

```
Command Line
============
```

After the program name ‘{No Value For "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 ‘{No Value For "AS"}’ to assemble.

Except for ‘--’ any command line argument that begins with a hyphen (‘-’) is an option. Each option changes the behavior of ‘{No Value For "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:

```
{No Value For "AS"} -o my-object-file.o mumble.s
{No Value For "AS"} -omy-object-file.o mumble.s
```

1.7 gas.guide/Input Files

```
                Input Files
===========
```

We use the phrase "source program", abbreviated "source", to describe the program input to one run of ‘{No Value For "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 '{No Value For "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 '{No Value For "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 '{No Value For "AS"}' no file names it attempts to read one input file from the '{No Value For "AS"}' standard input, which is normally your terminal. You may have to type ctl-D to tell '{No Value For "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, '{No Value For "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
Error and Warning Messages
.

"Physical files" are those files named in the command line given to '{No Value For "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 '{No Value For "AS"}' source is itself synthesized from other files. See '.app-file'.

1.8 gas.guide/Object

Output (Object) File
====================

Every time you run '{No Value For "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'. 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 ‘{No Value For "LD"}’. It contains assembled program code, information to help ‘{No Value For "LD"}’ integrate the assembled program into a runnable file, and (optionally) symbolic information for the debugger.

1.9 gas.guide/Errors

Error and Warning Messages
==========

‘{No Value For "AS"}’ may write warnings and error messages to the standard error file (usually your terminal). This should not happen when a compiler runs ‘{No Value For "AS"}’ automatically. Warnings report an assumption made so that ‘{No Value For "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 ‘.app-file’) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (see ‘.line’) 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.

1.10 gas.guide/Invoking

Command-Line Options
****

This chapter describes command-line options available in *all* versions of the GNU assembler; see Machine Dependencies, for options specific to the {No Value For "TARGET"}.

If you are invoking '{No Value For "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 to each compilation pass, including the assembler.)

a
: -a[dhlns] enable listings

D
: -D for compatibility

f
: -f to work faster

I
: -I for .include search path

K
: -K for compatibility

L
: -L to retain local labels

o
: -o to name the object file

R
: -R to join data and text sections

statistics
: -statistics to see statistics about assembly

v
: -v to announce version

W
: -W to suppress warnings

Z
: -Z to make object file even after errors

1.11 gas.guide/a

```
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'.
```

1.12 gas.guide/D

```
'-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
'{No Value For "AS"}'.
```

1.13 gas.guide/f

```
                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

                Preprocessing
                .

     *Warning:* if you use '-f' when the files actually need to be
     preprocessed (if they contain comments, for example), '{No Value
     For "AS"}' does not work correctly.
```

1.14 gas.guide/I

```
‘.include’ search path: ‘-I’ PATH
=================================
```

Use this option to add a PATH to the list of directories ‘{No Value For "AS"}’ searches for files specified in ‘.include’ directives (see ‘.include’). 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, ‘{No Value For "AS"}’ searches any ‘-I’ directories in the same order as they were specified (left to right) on the command line.

1.15 gas.guide/K

```
Difference Tables: ‘-K’
=======================
```

On the {No Value For "TARGET"} family, this option is allowed, but has no effect. It is permitted for compatibility with the GNU assembler on other platforms, where it can be used to warn when the assembler alters the machine code generated for ‘.word’ directives in difference tables. The {No Value For "TARGET"} family does not have the addressing limitations that sometimes lead to this alteration on other platforms.

1.16 gas.guide/L

```
                Include Local Labels: ‘-L’
==========================
```

Labels beginning with ‘L’ (upper case only) are called "local labels". See
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 ‘{No Value For "AS"}’ and ‘{No Value For "LD"}’ discard such labels, so you do not normally debug with them.

This option tells ‘{No Value For "AS"}’ to retain those ‘L...’ symbols in the object file. Usually if you do this you also tell the linker ‘{No Value For "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.

1.17 gas.guide/o

Name the Object File: ‘-o’
==========

There is always one object file output when you run ‘{No Value For "AS"}’. By default it has the name ‘a.out’. You use this option (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, ‘{No Value For "AS"}’ overwrites any existing file of the same name.

1.18 gas.guide/R

Join Data and Text Sections: ‘-R’
==========

‘-R’ tells ‘{No Value For "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
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 ‘{No Value For "AS"}’. In future, ‘-R’ may work this way.

1.19 gas.guide/statistics

Display Assembly Statistics: ‘--statistics’
==========

Use ‘--statistics’ to display two statistics about the resources used by ‘{No Value For "AS"}’: the maximum amount of space allocated during the assembly (in bytes), and the total execution time taken for the assembly (in CPU seconds).

1.20 gas.guide/v

```
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.

1.21 gas.guide/W

```
Suppress Warnings: ‘-W’
=======================
```

   ‘{No Value For "AS"}’ should never give a warning or error message when assembling compiler output. But programs written by people often cause ‘{No Value For "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 ‘{No Value For "AS"}’ assembles your file. Errors, which stop the assembly, are still reported.

1.22 gas.guide/Z

```
Generate Object File in Spite of Errors: ‘-Z’
=============================================
```

   After an error message, ‘{No Value For "AS"}’ normally produces no output. If for some reason you are interested in object file output even after ‘{No Value For "AS"}’ gives an error message on your program, use the ‘-Z’ option. If there are any errors, ‘{No Value For "AS"}’ continues anyways, and writes an object file after a final warning message of the form ‘N errors, M warnings, generating bad object file.’

1.23 gas.guide/Syntax

```
                Syntax
******
```

   This chapter describes the machine-independent syntax allowed in a source file. ‘{No Value For "AS"}’ syntax is similar to what many other assemblers use; it is inspired by the BSD 4.2 assembler.

```
Preprocessing
              Preprocessing

Whitespace
                  Whitespace

Comments
                    Comments

Symbol Intro
                Symbols

Statements
                  Statements

Constants
                   Constants
```

1.24 gas.guide/Preprocessing

```
Preprocessing
=============

   The `{No Value For "AS"}' internal preprocessor:
   * adjusts and removes extra whitespace.  It leaves one space or tab
     before the keywords on a line, and turns any other whitespace 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 `.include').
You can use the GNU C compiler driver to get other "CPP" style
preprocessing, by giving the input file a `.S' suffix.  See
Options Controlling the Kind of Output.

   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.
```

1.25 gas.guide/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

                Character Constants
                ), any whitespace means the same as exactly one
space.
```

1.26 gas.guide/Comments

```
                Comments
========

   There are two ways of rendering comments to `{No Value For "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 is see
See Machine Dependencies.

   To be compatible with past assemblers, lines that begin with `#'
have a special interpretation.  Following the `#' should be an absolute
expression (see
                Expressions
                ): the logical line number of the *next*
line.  Then a string (see
                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 `{No Value For "AS"}'.
```

1.27 gas.guide/Symbol Intro

```
                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
`_.$'.  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
                Symbols
                .
```

1.28 gas.guide/Statements

```
                Statements
==========

   A "statement" ends at a newline character (`\n') or at a semicolon
(`;').  The newline or semicolon is considered part of the preceding
statement.  Newlines and semicolons 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 `{No Value For "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.

   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
                Labels
                .

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

1.29 gas.guide/Constants

```
                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.


                 Characters
                                   Character Constants

                 Numbers
                                      Number Constants
```

1.30 gas.guide/Characters

```
                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
Strings

Chars
Characters

1.31 gas.guide/Strings

```
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 '{No Value For "AS"}' to
interpret the second character literally as a backslash (which prevents
'{No Value For "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.

'\\'
     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 '{No Value For
```

"AS"}' has no other interpretation, so '{No Value For "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.

1.32 gas.guide/Chars

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 semicolon ';') 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. '{No Value For "AS"}' assumes your character code is ASCII: ''A' means 65, ''B' means 66, and so on.

1.33 gas.guide/Numbers

Number Constants

'{No Value For "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
Integers

Bignums
Bignums

Flonums
Flonums

1.34 gas.guide/Integers

```
                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

```
                Prefix Operators
                ).
```

1.35 gas.guide/Bignums

```
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.

1.36 gas.guide/Flonums

```
Flonums
.......
```

A "flonum" represents a floating point number. The translation is indirect: a decimal floating point number from the text is converted by ‘{No Value For "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 ‘{No Value For "AS"}’ specialized to that computer.

A flonum is written by writing (in order)

* The digit ‘0’.

* A letter, to tell ‘{No Value For "AS"}’ the rest of the number is

```
     a flonum.

   * 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.

   `{No Value For "AS"}' does all processing using integers.  Flonums
are computed independently of any floating point hardware in the
computer running `{No Value For "AS"}'.
```

1.37 gas.guide/Sections

```
                Sections and Relocation
************************


                Secs Background
                             Background

                Ld Sections
                                {No Value For "LD"} Sections

                As Sections
                                {No Value For "AS"} Internal Sections

                Sub-Sections
                               Sub-Sections

                bss
                                        bss Section
```

1.38 gas.guide/Secs Background

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 ‘{No Value For "LD"}’ reads many object files (partial programs) and combines their contents to form a runnable program. When ‘{No Value For "AS"}’ emits an object file, the partial program is assumed to start at address 0. ‘{No Value For "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 ‘{No Value For "AS"}’ uses sections.

‘{No Value For "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.

An object file written by ‘{No Value For "AS"}’ has at least three sections, any of which may be empty. These are named "text", "data" and "bss" sections.

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

To let ‘{No Value For "LD"}’ know which data changes when the sections are relocated, and how to change that data, ‘{No Value For "AS"}’ also writes to the object file details of the relocation needed. To perform relocation ‘{No Value For "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 ‘{No Value For "AS"}’ ever uses is expressed as
  (SECTION) + (OFFSET INTO SECTION)

Further, most expressions ‘{No Value For "AS"}’ computes have this section-relative nature.

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 `{No Value For "LD"}' mixes partial programs, addresses in the absolute section remain unchanged. For example, address `{absolute 0}' is "relocated" to run-time address 0 by `{No Value For "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. `{No Value For "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 `{No Value For "LD"}'; others are invented for use of `{No Value For "AS"}' and have no meaning except during assembly.

1.39 gas.guide/Ld Sections

```
{No Value For "LD"} Sections
============================
```

`{No Value For "LD"}' deals with just four kinds of sections, summarized below.

These sections hold your program. `{No Value For "AS"}' and `{No Value For "LD"}' treat them as separate but equal sections. Anything you can say of one section is true another.

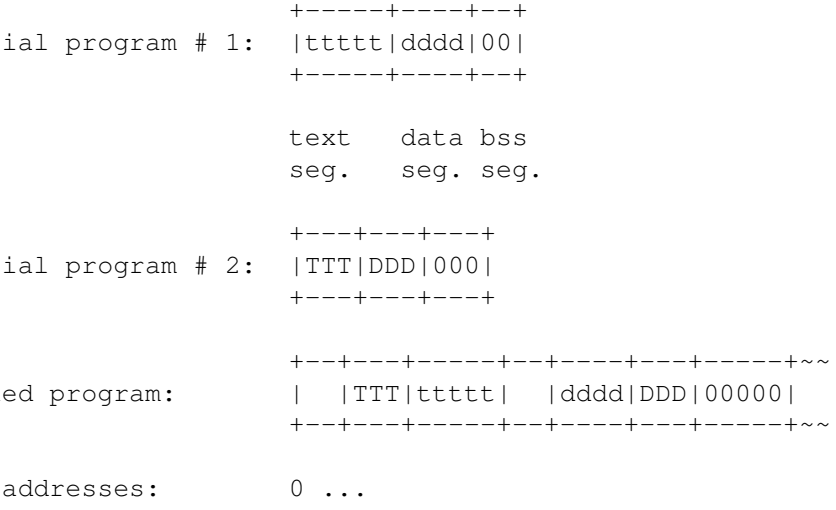
bss section
: This section contains zeroed bytes when your program begins running. It is used to hold unitialized variables or common storage. The length of each partial program's bss section is important, but because it starts out containing zeroed bytes there is no need to store explicit zero bytes 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 `{No

Value For "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. Memory addresses are on the horizontal axis.



1.40 gas.guide/As Sections

{No Value For "AS"} Internal Sections
=====================================

These sections are meant only for the internal use of '{No Value For "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 '{No Value For "AS"}' warning messages, so it might be helpful to have an idea of their meanings to '{No Value For "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.

1.41 gas.guide/Sub-Sections

```
                Sub-Sections
============
```

You may have separate groups of data in named sections that you want to end up near to each other in the object file, even though they are not contiguous in the assembler source. ‘{No Value For "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.

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; ‘{No Value For "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. EXPRESSION should be an absolute expression. (See

```
                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 ‘{No Value For "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.

1.42 gas.guide/bss

```
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 ‘.comm’, see ‘.lcomm’.

1.43 gas.guide/Symbols

```
                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:* ‘{No Value For "AS"}’ does not place symbols in the object file in the same order they were declared. This may break some debuggers.

```
Labels
                        Labels

Setting Symbols
              Giving Symbols Other Values

Symbol Names
                 Symbol Names

Dot
                           The Special Dot Symbol

Symbol Attributes
            Symbol Attributes
```

1.44 gas.guide/Labels

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.

1.45 gas.guide/Setting Symbols

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

Expressions
). This is equivalent to using the ‘.set’ directive. See ‘.set’.

1.46 gas.guide/Symbol Names

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 See Machine Dependencies. That character may be followed by any string of digits, letters, dollar signs (unless otherwise noted in See Machine Dependencies), and underscores.

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 '{No Value For "AS"}' and '{No Value For "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 '{No Value For "AS"}' retains these symbols in the object file. If you also instruct '{No Value For "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 'L?A1', the 44th '3:' is named 'L?A44'.

1.47 gas.guide/Dot

```
The Special Dot Symbol
======================
```

The special symbol '.' refers to the current address that '{No Value For "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'.

1.48 gas.guide/Symbol Attributes

```
                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, ‘{No Value For "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
```

1.49 gas.guide/Symbol Value

```
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 ‘{No Value For "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 ‘{No Value For "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.

1.50 gas.guide/Symbol Type

```
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.

1.51 gas.guide/Expressions

```
                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 ‘{No Value For "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. ‘{No Value For "AS"}’ aborts with an error message in this situation.

```
 Empty Exprs
                   Empty Expressions

 Integer Exprs
                 Integer Expressions
```

1.52 gas.guide/Empty Exprs

```
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 ‘{No Value For "AS"}’ assumes a value of (absolute) 0. This is compatible with other assemblers.

1.53 gas.guide/Integer Exprs

```
                Integer Expressions
===================
```

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

1.54 gas.guide/Arguments

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 '{No Value For "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.

1.55 gas.guide/Operators

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.

1.56 gas.guide/Prefix Ops

Prefix Operator

   ‘{No Value For "AS"}’ has the following "prefix operators". They
each take one argument, which must be absolute.

‘-’
     "Negation". Two’s complement negation.

‘~’
     "Complementation". Bitwise not.

1.57 gas.guide/Infix Ops

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.

1.58 gas.guide/Pseudo Ops

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.