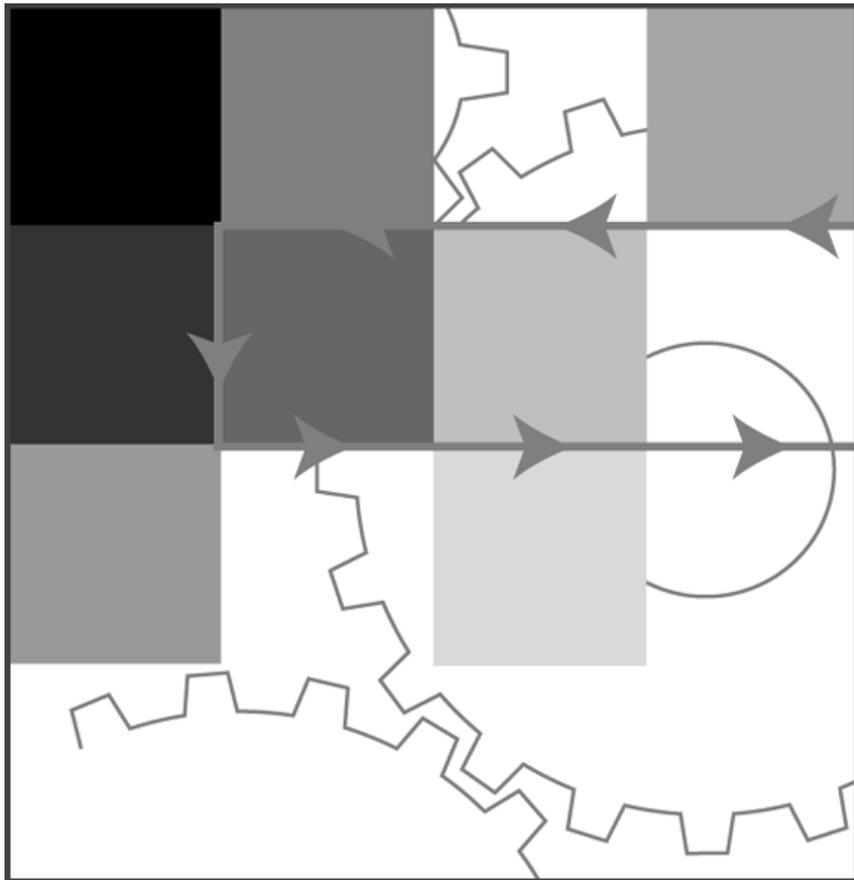


GNUPro® Toolkit

GNUPro Auxiliary Development Tools

- ***Using as***
- ***Using binutils***
- ***Using cygwin***
- ***Using info***



GNUPro 2001

Copyright © 1991-2001 Red Hat[®], Inc. All rights reserved.

Red Hat[®], GNUPro[®], the Red Hat Shadow Man logo[®], Source-Navigator[™], Insight[™], Cygwin[™], eCos[™], and Red Hat Embedded DevKit[™] are all trademarks or registered trademarks of Red Hat, Inc. ARM[®], Thumb[®], and ARM Powered[®] are registered trademarks of ARM Limited. SA[™], SA-110[™], SA-1100[™], SA-1110[™], SA-1500[™], SA-1510[™] are trademarks of ARM Limited. All other brands or product names are the property of their respective owners. “ARM” is used to represent any or all of ARM Holdings plc (LSE; ARM: NASDAQ; ARMHY), its operating company, ARM Limited, and the regional subsidiaries ARM INC., ARM KK, and ARM Korea Ltd.

AT&T[®] is a registered trademark of AT&T, Inc.

Hitachi[®], SuperH[®], and H8[®] are registered trademarks of Hitachi, Ltd.

IBM[®], PowerPC[®], and RS/6000[®] are registered trademarks of IBM Corporation.

Intel[®], Pentium[®], Pentium II[®], and StrongARM[®] are registered trademarks of Intel Corporation.

Linux[®] is a registered trademark of Linus Torvalds.

Matsushita[®], Panasonic[®], PanaX[®], and PanaXSeries[®] are registered trademarks of Matsushita, Inc.

Microsoft[®] Windows[®] CE, Microsoft[®] Windows NT[®], Microsoft[®] Windows[®] 98, and Win32[®] are registered trademarks of Microsoft Corporation.

MIPS[®] is a registered trademark and MIPS I[™], MIPS II[™], MIPS III[™], MIPS IV[™], and MIPS16[™] are all trademarks or registered trademarks of MIPS Technologies, Inc.

Mitsubishi[®] is a registered trademark of Mitsubishi Electric Corporation.

Motorola[®] is a registered trademark of Motorola, Inc.

Sun[®], SPARC[®], SunOS[™], Solaris[™], and Java[™], are trademarks or registered trademarks of Sun Microsystems, Inc..

UNIX[®] is a registered trademark of The Open Group.

NEC[®], VR5000[™], VRC5074[™], VR5400[™], VR5432[™], VR5464[™], VRC5475[™], VRC5476[™], VRC5477[™], VRC5484[™] are trademarks or registered trademarks of NEC Corporation.

All other brand and product names, services names, trademarks and copyrights are the property of their respective owners.

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information within the documentation. For licenses and use information, see “General Licenses and Terms for Using GNUPro Toolkit” in the GNUPro Toolkit *Getting Started Guide*.

How to contact Red Hat

Use the following means to contact Red Hat.

Red Hat Corporate Headquarters

2600 Meridian Parkway

Durham, NC 27713 USA

Telephone (toll free): +1 888 REDHAT 1 (+1 888 733 4281)

Telephone (main line): +1 919 547 0012

Telephone (FAX line): +1 919 547 0024

Website: <http://www.redhat.com/>

Contents

Overview of GNUPro Auxiliary Development Tools	1
--	---

Using `as`

Overview of <code>as</code>, the GNU Assembler	5
Overview of Special Features for <code>as</code>	5
Invoking <code>as</code>, the GNU Assembler	7
Using <code>as</code> and Its Options	8
<code>as</code> Command Line Usage	8
Object File Formats	11
Input files	11
Output (Object) File	12
Error and Warning Messages	12
Command Line Options	15
Enable Listings: <code>-a[cdhlns]</code> Options	16
Conform with Other Assemblers: <code>-D</code> Option	16
Work Faster: <code>-f</code> Option	17
Search Path for <code>.include</code> Specifications: <code>-I path</code> Option	17
Warn for Difference Tables: <code>-K</code> Option	17
Include Local Labels: <code>-L</code> Option	17
Assemble in MRI Compatibility Mode: <code>-M</code> Option	18
Generate Dependency Tracking: <code>--MD</code> Option	20

Name the Object File: <code>-o</code> Option	20
Join Data and Text Sections: <code>-R</code> Option.....	20
Display Assembly Statistics: <code>--statistics</code> Option	20
Announce Version: <code>-v</code> Option	21
Suppress Warnings: <code>-w</code> Option.....	21
Generate Object File in Spite of Errors: <code>-z</code> Option.....	21
Syntax	23
Preprocessing	23
Whitespace	24
Comments	24
Symbols.....	25
Statements	26
Constants	26
Character Constants	27
Strings	27
Characters.....	28
Number Constants.....	28
Integers.....	28
Bignums	29
Flonums	29
Sections and Relocation	31
1d Sections	33
as Internal Sections.....	34
Sub-sections	34
bss Section.....	35
Symbols for the GNU Assembler	37
Labels	37
Giving Symbols Other Values	38
Symbol Names	38
Local Symbol Names.....	38
The Special Dot Symbol	39
Symbol Attributes	39
Value.....	39
Type	40
Symbol Attributes for <code>a.out</code> File Format.....	40
Symbol Attributes for COFF Format.....	40
Symbol Attributes for SOM Format	40
Expressions	41
Empty Expressions.....	41
Integer Expressions	42
Arguments.....	42
Operators.....	42

Prefix Operators	42
Infix Operators	43
Assembler Macro Directives	45
.abort Directive	48
.ABORT Directive	48
.align <i>abs-expr</i> , <i>abs-expr</i> , <i>abs-expr</i> Directive	48
.ascii " <i>string</i> "... Directive.....	49
.asciz " <i>string</i> "... Directive.....	49
.balign[<i>wl</i>] <i>abs-expr</i> , <i>abs-expr</i> , <i>abs-expr</i> Directive	49
.byte <i>expressions</i> Directive.....	50
.comm <i>symbol</i> , <i>length</i> Directive.....	50
.data <i>subsection</i> Directive	50
.def <i>name</i> Directive	50
.desc <i>symbol</i> , <i>abs-expression</i> Directive.....	51
.dim Directive	51
.double <i>flonums</i> Directive	51
.eject Directive.....	51
.else Directive.....	51
.elseif Directive	51
.end Directive	52
.endif Directive	52
.endfunc Directive	52
.endif Directive	52
.equ <i>symbol</i> , <i>expression</i> Directive.....	52
.equiv <i>symbol</i> , <i>expression</i> Directive.....	52
.err Directive.....	53
.exitm Directive.....	53
.extern Directive	53
.fail <i>expression</i> Directive	53
.file <i>string</i> Directive	53
.fill <i>repeat</i> , <i>size</i> , <i>value</i> Directive.....	53
.float <i>flonums</i> Directive	54
.func <i>name</i> , <i>label</i> Directive	54
.global <i>symbol</i> , <i>.globl symbol</i> Directive.....	54
.hword <i>expressions</i> Directive.....	54
.ident Directive	55
.if <i>absolute expression</i> Directive.....	55
.include " <i>file</i> Directive	56
.int <i>expressions</i> Directive	56
.irp <i>symbol</i> , <i>values</i> ... Directive.....	56
.irpc <i>symbol</i> , <i>values</i> ... Directive.....	57
.lcomm <i>symbol</i> , <i>length</i> Directive.....	57

<code>.lflags</code> Directive	57
<code>.line</code> <i>line-number</i> Directive	58
<code>.linkonce</code> <i>type</i> Directive	58
<code>.list</code> Directive	59
<code>.ln</code> <i>line-number</i> Directive	59
<code>.long</code> <i>expressions</i> Directive	59
<code>.macro</code> Directive	59
<code>.mri</code> <i>val</i> Directive	60
<code>.nolist</code> Directive	60
<code>.octa</code> <i>bignums</i> Directive	61
<code>.org</code> <i>new-lc, fill</i> Directive	61
<code>.p2align</code> [<i>wl</i>] <i>abs-expr, abs-expr, abs-expr</i> Directive	61
<code>.print</code> <i>string</i> Directive	62
<code>.psize</code> <i>lines, columns</i> Directive	62
<code>.purgem</code> <i>name</i> Directive	62
<code>.quad</code> <i>bignums</i> Directive	62
<code>.rept</code> <i>count</i> Directive	63
<code>.sbt1</code> " <i>subheading</i> " Directive	63
<code>.scl</code> <i>class</i> Directive	63
<code>.section</code> <i>name</i> Directive	63
<code>.set</code> <i>symbol, expression</i> Directive	65
<code>.short</code> <i>expressions</i> Directive	65
<code>.single</code> <i>flonums</i> Directive	65
<code>.size</code> Directive	65
<code>.sleb128</code> <i>expressions</i> Directive	66
<code>.skip</code> <i>size, fill</i> Directive	66
<code>.space</code> <i>size, fill</i> Directive	66
<code>.stabd, .stabn, and .stabs</code> Directives	66
<code>.string</code> " <i>str</i> " Directive	67
<code>.struct</code> <i>expression</i> Directive	67
<code>.symver</code> Directive	68
<code>.tag</code> <i>structname</i> Directive	68
<code>.text</code> <i>subsection</i> Directive	68
<code>.title</code> " <i>heading</i> " Directive	69
<code>.type</code> <i>int</i> Directive	69
<code>.val</code> <i>addr</i> Directive	69
<code>.uleb128</code> <i>expressions</i> Directive	69
<code>.word</code> <i>expressions</i> Directive	69
<code>.zero</code> <i>size</i> Directive	70
Machine Dependent Features for the GNU Assembler	71
AMD 29K Dependent features	73
ARC Dependent Features	77

ARM Dependent Features	79
AT&T and Intel x86 Dependent Features	83
Hitachi H8/300 Dependent Features	91
Hitachi H8/500 Dependent Features	93
Hitachi SH Dependent Features	95
HPPA Dependent Features	99
Intel StrongARM Dependent Features	105
Intel 960 Dependent Features	107
Intel x86 and IA64 Dependent Features	111
MIPS Dependent Features	113
Mitsubishi D10V Dependent Features	119
Mitsubishi D30V Dependent Features	123
Mitsubishi M32R Dependent Features	129
Motorola 68K Dependent Features	133
NEC V850 Dependent Features	141
PowerPC Dependent Features	147
Sun Dependent Features	149
Vax Dependent Features	153
Zilog Z8000 Dependent Features	159
Acknowledgments for the GNU Assembler	163

Using binutils

Overview of binutils, the GNU Binary Utilities	167
Controlling <code>ar</code> on the Command Line	169
Controlling <code>ar</code> with a Script	172
Selecting the Target System	213
Target Selection	214
objdump Target	214
objcopy and strip Input Target	214
objcopy and strip Output Target	214
nm , size , and strings Target	215
Linker Input Target	215
Linker Output Target	215
Architecture Selection	216
objdump Architecture	216
objcopy , nm , size , strings Architecture	216
Linker input Architecture	216
Linker output Architecture	216
Linker Emulation Selection	217

Using Cygwin

Windows Development with Cygwin: a Win32 Porting Layer	221
Porting UNIX Tools to Win32	222
Goals of Cygwin	223
Harnessing the Power of the Web for Cygwin	224
The Cygwin Architecture.....	225
Process Creation for Cygwin	228
Future Work for Cygwin.....	232
Compatibility Issues with Cygwin	233
Cygwin's Compatibility with ANSI Standards	233
Cygwin's Compatibility with POSIX.1 Standards	234
Cygwin's Compatibility with Other Miscellaneous Standards.....	235
Setting up Cygwin	237
Directory Structure for Cygwin	238
Microsoft Windows NT security and <code>ntsec</code> usage.....	238
Mapping leak	243
Cygwin API Calls	244
The <code>setuid</code> Concept	245
Environment Variables for Cygwin	247
Mount Table	250
Text and Binary Modes.....	250
Programming	253
File Permissions	253
Special File Names	254
Using GCC with Cygwin	255
Console Mode Applications.....	255
GUI Mode Applications.....	255
Debugging Cygwin Programs	256
Building and Using DLLs with Cygwin	257
Building DLLs	258
Defining Microsoft Windows Resources for Cygwin.....	258
Cygwin Utilities	262
Cygwin Functions	276

Using `info`

Overview of <code>info</code>, the GNU Online Documentation	283
Using the <code>info</code> program.....	284
Reading <code>info</code> Files	285
GNU <code>info</code> Command Line Options.....	285

Moving the Cursor	287
Moving Text within a Window	288
Selecting a New Node	289
Searching an <code>info</code> File	291
Selecting Cross References	292
Parts of a Cross Reference	292
Selecting Cross References	293
Manipulating Multiple Windows	293
The Mode Line	294
Window Commands	294
The Echo Area	295
Printing Out Nodes	297
Miscellaneous Commands	298
Manipulating Variables	299
Making <code>info</code> files from <code>Texinfo</code> files	303
Controlling Paragraph Formats	303
<code>makeinfo</code> Command Line Options	304
What Makes a Valid <code>info</code> File?	305
Defaulting the <code>Prev</code> , <code>Next</code> , and <code>Up</code> Pointers	306
Index	309

Overview of GNUPro Auxiliary Development Tools

The following list details the contents of the *GNUPro Auxiliary Development Tools* documentation.

- “Using as”
(see “Overview of as, the GNU Assembler” on page 5)
- “Using binutils”
(see “Overview of binutils, the GNU Binary Utilities” on page 167)
- “Using Cygwin”
(see “Windows Development with Cygwin: a Win32 Porting Layer” on page 221)
- “Using info”
(see “Overview of info, the GNU Online Documentation” on page 283)

Using as

Copyright © 1991-2000 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information within the documentation.

For licenses and use information, see “General Licenses and Terms for Using GNUPro Toolkit” on page 105; specifically, see “GNU General Public License” on page 106, “GNU Lesser General Public License” on page 111, and “Tcl/Tk Tool Command Language and Windowing Toolkit License” on page 118 in *Getting Started Guide*.

Free Software Foundation

59 Temple Place / Suite 330
Boston, MA 02111-1307 USA

ISBN: 1-882114-66-3

1

Overview of `as`, the GNU Assembler

The following documentation serves as a user guide to the GNU assembler, `as`, and is not an introduction to programming in assembly language. It describes what you need to know to use GNU `as`, the syntax expected in source files, including symbols, constants, expressions, and the general directives.

- “Invoking `as`, the GNU Assembler” on page 7
- “Command Line Options” on page 15
- “Syntax” on page 23
- “Sections and Relocation” on page 31
- “Symbols for the GNU Assembler” on page 37
- “Expressions” on page 41
- “Assembler Macro Directives” on page 45
- “Machine Dependent Features for the GNU Assembler” on page 71

Overview of Special Features for `as`

The following documentation points to some of the special machine-dependent features of the assembler. This is not an attempt to introduce each machine’s architecture, and neither is it a description of the instruction set, standard mnemonics,

registers or addressing modes that are standard to the specific architecture. Consult the manufacturer's machine architecture documentation for such information.

- “AMD 29K Dependent features” on page 73
- “ARC Dependent Features” on page 77
- “ARM Dependent Features” on page 79
- “AT&T and Intel x86 Dependent Features” on page 83
- “Hitachi H8/300 Dependent Features” on page 91
- “Hitachi H8/500 Dependent Features” on page 93
- “Hitachi SH Dependent Features” on page 95
- “HPPA Dependent Features” on page 99
- “Intel StrongARM Dependent Features” on page 105
- “Intel 960 Dependent Features” on page 107
- “Intel x86 and IA64 Dependent Features” on page 111
- “MIPS Dependent Features” on page 113
- “Mitsubishi D10V Dependent Features” on page 119
- “Mitsubishi D30V Dependent Features” on page 123
- “Mitsubishi M32R Dependent Features” on page 129
- “Motorola 68K Dependent Features” on page 133
- “NEC V850 Dependent Features” on page 141
- “PowerPC Dependent Features” on page 147
- “Sun Dependent Features” on page 149
- “Vax Dependent Features” on page 153
- “Zilog Z8000 Dependent Features” on page 159

2

Invoking `as`, the GNU Assembler

The GNU assembler, `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. See the following documentation for more general discussion of the GNU assembler.

- “Using `as` and Its Options” on page 8
- “`as` Command Line Usage” on page 8
- “Object File Formats” on page 11
- “Input files” on page 11
- “Output (Object) File” on page 12
- “Error and Warning Messages” on page 12

`as` is primarily intended to assemble the output of the GNU C compiler, `gcc`, for use by the GNU linker, `ld`. Nevertheless, `as` assembles correctly everything that other assemblers for the same machine would assemble. Any exceptions are documented explicitly (for specific processors and their families, see “Machine Dependent Features for the GNU Assembler” on page 71). This doesn’t mean `as` always uses the same syntax as another assembler for the same architecture; for example, there are several incompatible versions of Motorola 680x0 assembly language syntax.

Using `as` and Its Options

The following example summarizes how to invoke `as`, along with showing assembler options for compiler output. For the basic descriptions of what the options do, see “`as` Command Line Usage” (below), and, for more details, see “Command Line Options” on page 15.

```
as [ -a[cdhlmns][=file] ] [ -D ] [ --defsym sym=value ] \
  [ --dump-config ] [ --emulation=name ] [ -f ] [ --gstabs ] \
  [ --gdwarf2 ] [ --help ] [ -I dir ] [ -J ] [ -K ] [ -L ] \
  [ --keep-locals ] [ --listing-lhs-width ] [ --listing-lhs-width2 ] \
  [ --listing-rhs-width ] [ --listing-cont-lines ] [ -M ] [ --mri ] \
  [ -MD file ] [ -o objfile ] [ -R ] [ --statistics ] \
  [ --strip-local-absolute ] [ -traditional-format ] [ -t ] \
  [ --itbl file ] [ -v ] [ -version ] [ --version ] [ -W ] [ -warn ] \
  [ -fatal-warnings ] [ -Z ] [ -- | files ... ]
```

`as` Command Line Usage

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.

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 (which is the GNU standard).

The following two command lines are equivalent.

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

`--` (two hyphens), by themselves, name 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 hyphen followed by one or more letters; the *case* of the letter is important.

Unlike older assemblers, `as` assemble a source program in one pass of the source file, with a subtle impact on the `.org` directive (see “.org new-lc, fill Directive” on page 61).

`-a[cdhlmns]`

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

- `-ac`
Omit false conditionals

- `-ad`
Omit debugging directives
- `-ah`
Include high-level source
- `-al`
Include assembly
- `-am`
Include macro expansions
- `-an`
Omit forms processing
- `-as`
Include symbols
- `-a=file`
Set the name of the listing file.

You may combine any of the previous options; for example, use `-aln` for assembly listing without forms processing. The `-a=file` option, if used, must be the last one. By itself, `-a` defaults to `-ahls`.

- `-D`
Enable some internal assembler debugging.
- `--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.
- `--dump-config`
Display the configuration settings used to create the assembler and then exit.
- `--emulation=name`
Set the assembler’s emulation (the output and format) to *name*.. Use this option only if the assembler has been built to support multiple emulations.
- `-f`
“fast”—skip whitespace and comment preprocessing (assume source is compiler output).
- `--gstabs`
Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.
- `--gdwarf2`
Generate DWARF2 debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.
- `--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-locals
Keep (in the symbol table) local symbols. On traditional `a.out` systems these start with `L`, but different systems have different local label prefixes.
- listing-lhs-width=*number*
Set the width, in words, of a listing's output data column. Sets the maximum width, in words, of the first line of the hex byte dump (as *number*).
- listing-lhs-width2=*number*
Sets the maximum width, in words, of any further lines of the hex byte dump (as *number*) for a given input source line. If this value is not specified, it defaults to being the same as the value specified for `--listing-lhs-width`. If neither switch is used, the default is to one.
- listing-rhs-width=*number*
Sets the maximum width, in characters, of the source line that is displayed alongside the hex dump (as *number*). The default value for this parameter is 100. The source line is displayed on the right hand side of the listing output.
- listing-cont-lines=*number*
Sets the maximum number of continuation lines of hex dump (as *number*) that will be displayed for a given single line of source input. The default value is 4.
- M
--mri
Assemble in MRI compatibility mode for Microtec compilers, the C and C++ compilers available from Mentor Graphics' Embedded Software Division (formerly Microtec Research, Inc.).
- MD *file*
Write dependency information to *file*.
- o *objfile*
Name the object-file output from assembly *objfile*.
- R
Fold the data section into the text section.
- statistics
Print e maximum space (in bytes) and total time (in seconds) used by assembly.
- strip-local-absolute
Remove local absolute symbols from the outgoing symbol table.
- t
--itbl *file*
Extend the instruction set to include instructions matching the specifications defined in *file*.

```
--traditional-format
    Use the same format as the native assembler, where possible.
-v
-version
    Print the assembler version.
--version
    Print the assembler version and exit.
-W
--no-warn
    Suppress warning messages.
--fatal-warnings
    Treat warnings as errors.
--warn
    Don't suppress warning messages or treat them as errors.
-z
    Generate an object file even after errors.
-- | files ...
    Standard input, or source files (files ...) to assemble.
```

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 page 39.

Input files

The phrase, *source program*, abbreviated *source*, describes 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 Ctrl-D to tell `as` there is no more program to assemble.

Use `--` (two hyphens) 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.

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” on page 12.

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 “.file string Directive” on page 53.

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 GNU linker, `ld`. It contains assembled program code, information to help `ld` integrate the assembled program into a file that can run, and optionally provides symbolic information for the GNU 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 following format (where *NNN* is a line number).

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

If a logical file name has been given (see “.file string Directive” on page 53) 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 line-number Directive” on page 58), 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 UNIX tradition).

Error messages have the following format.

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

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

3

Command Line Options

The following documentation describes command-line options available in all versions of the GNU assembler; see “Using `as` and Its Options” on page 8 for a complete invocation declaration example.

- “Enable Listings: `-a[cdhlms]` Options” on page 16
- “Conform with Other Assemblers: `-D` Option” on page 16
- “Work Faster: `-f` Option” on page 17
- “Search Path for `.include` Specifications: `-I` path Option” on page 17
- “Warn for Difference Tables: `-K` Option” on page 17
- “Include Local Labels: `-L` Option” on page 17
- “Assemble in MRI Compatibility Mode: `-M` Option” on page 18
- “Generate Dependency Tracking: `--MD` Option” on page 20
- “Name the Object File: `-o` Option” on page 20
- “Join Data and Text Sections: `-R` Option” on page 20
- “Display Assembly Statistics: `--statistics` Option” on page 20
- “Announce Version: `-v` Option” on page 21
- “Suppress Warnings: `-W` Option” on page 21
- “Generate Object File in Spite of Errors: `-Z` Option” on page 21

See also “Machine Dependent Features for the GNU Assembler” on page 71 in order

to find documentation specific to a processor's or an architecture's particular options.

If you are invoking `as` using the GNU C compiler, you can use the `-wa` option to pass arguments through to the assembler. The assembler arguments must be separated from the `-wa` compiler option using commas. Use the following input for example.

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

This input 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.)

Enable Listings: `-a [cdhlns]` Options

This grouping of options enables listing output from the assembler. By itself, `-a` requests high-level, assembly, and symbols listing. For enabling listings, `-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 also assembly listings (`-al`) be requested. Use the `-ad` option to omit debugging directives from the listing. Use the `-ac` option to omit false conditionals from a listing. Any lines which are not assembled because of a false `.if` (or `.ifdef`, or any other conditional), or a true `.if` followed by a `.else`, will be omitted from the listing. 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 `.sbt1`. 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, such as `-aln`.

If the assembler source is coming from the standard input (for instance, because it is being created by GCC using the `-pipe` command line option) then the listing will not contain any comments or preprocessor directives, since the listing code buffers input source lines from standard input only after they have been preprocessed by the assembler. This reduces memory usage and makes the code more efficient.

Conform with Other Assemblers: `-D` Option

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 Option

-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” on page 23.

WARNING! If you use -f when the files actually need to be pre-processed (if they contain comments, for example), as does not work correctly.

Search Path for .include Specifications: -I path Option

Use this option to add a *path* to the list of directories as searches for files specified in .include directives (see “.include “file Directive” on page 56). 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.

Warn for Difference Tables: -K Option

as sometimes alters the code emitted for directives of the form .word *sym1-sym2*; see “.word expressions Directive” on page 69. You can use the -K option if you want a warning issued when this is done.

Include Local Labels: -L Option

Labels beginning with L (upper case only) are called *local labels*; for more information, see “Symbol Names” on page 38. 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\$; local labels begin with ; for the ARM family.

Assemble in MRI Compatibility Mode: `-M` Option

For compatibility with the Microtec Research Inc. (MRI) mode, use the `-M` or `--mri` option. 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 MRI. The exact nature of the MRI syntax will not be documented here; see MRI documentation for more information.

The purpose of the `-M` or `--mri` 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 formats have the following usage.

- *global symbols in common section*
The Motorola 68K MRI assembler supports common sections, which are merged by the linker. Other object file formats do not support this functionality. `as` handles common sections by treating them as a single common symbol, permitting the definition of local symbols within a common section, but without supporting global symbols (since it has no way to describe them).
- *complex relocations*
The MRI assemblers support relocations against a negated section address, and relocations that combine the start addresses of two or more sections; these are not supported 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 Motorola 68K MRI `ORG` pseudo-op begins an absolute section at a given address; this differs from the usual `.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.

The following documentation details 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.

Generate Dependency Tracking: `--MD` Option

`as` can generate a dependency file for the file that it creates. This file consists of a single rule suitable for `make`, describing the dependencies for the main source file.

The rule is written to the file named in its argument.

This feature is used in the automatic updating of files.

Name the Object File: `-o` Option

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` Option

`-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 “Sections and Relocation” on page 31.) 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` Option

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` Option

You can find out what version of `as` is running by including the `-v` option (or `-version`) on the command line.

Suppress Warnings: `-W` Option

`as` should never give a warning or error message when assembling compiler output. However, 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, `-w`, no warnings are issued. This option only affects the warning messages and it does not change any particular of how `as` assembles your file. Errors that stop the assembly are still reported.

Generate Object File in Spite of Errors: `-Z` Option

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, and writes an object file after a final warning message of the following form.

```
n errors, m warnings, generating bad object file.
```


4

Syntax

The following documentation describes the machine-independent syntax allowed in a source file.

- “Preprocessing” (below)
- “Whitespace” on page 24
- “Comments” on page 24
- “Symbols” on page 25
- “Statements” on page 26
- “Constants” on page 26
- “Character Constants” on page 27
- “Strings” on page 27
- “Characters” on page 28

`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 has the following functionality.

- 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 “file Directive” on page 56). 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” in *Using gcc* in **GNUPro Compiler Tools**.

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 “Character Constants” on page 27), 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, as in the following statement.

```
/*
   The only way to include a newline ('\n') in a comment
   is to use this sort of comment.
*/
```

This means you may not nest the following types of comments.

```
/* 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 ‘#’ for the Vax; ‘#’ for the i960; ‘!’ for the SPARC family; ‘|’ for the Motorola 68K family; ‘;’ for the AMD 29K family; ‘;’ for the H8/300 family; ‘!’ for the H8/500 family; ‘;’ for the HPPA family; ‘!’ for the Hitachi SH family; ‘!’ for the Z8000 family, and ‘;’ for the ARC family; ‘#’ for the V850 processor; see “Machine Dependent Features for the GNU Assembler” on page 71 to locate specific families.

The V850 assembler also supports a double dash as starting a comment that extends to the end of the line, as the following example shows.

```
--;
```

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 “Expressions” on page 41): the logical line number of the *next* line. Then a string (see “Strings” on page 27) 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 for each architecture; see “Machine Dependent Features for the GNU Assembler” on page 71 for specific processor families.

WARNING! 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 for the GNU Assembler” on page 37.

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. Exceptions are noted for each architecture; see “Machine Dependent Features for the GNU Assembler” on page 71.) The newline or separator character is considered part of its preceding statement.

IMPORTANT! 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 “Labels” on page 37.

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

```
label:          .directivefollowed 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 the following input example.

```
.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 us to interpret the second character literally as a backslash (which prevents us 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 the escapes represent, varies widely among assemblers. The current set is what the BSD 4.2 assembler recognizes, as a subset of what most C compilers recognize. If 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 \. The quote is an ‘acute’ accent (‘), not a ‘grave’ accent (’). A newline 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; see “Flonums” on page 29.

Integers

A binary integer is `0b` or `0B` followed by one or more of the binary digits, 01.

`0b` invalid

`0b0` valid

An octal integer is 0 followed by zero or more of the octal digits (0, 1, 2, 3, 4, 5, 6, 7).

A decimal integer starts with a non-zero digit followed by zero or more digits (0, 1, 2, 3, 4, 5, 6, 7, 8, 9).

A hexadecimal integer is `0x`, or `0X`, followed by zero 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” on page 42).

```
0x      valid=0x0
0x1    valid
```

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 uses (in order) the following input:

- 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: `D`, `F`, `P`, `R`, `S`, or `X` (in uppercase or lower case).
 - On the ARC, the letter must be one of the letters: `D`, `F`, `R`, or `S` (in uppercase or lowercase).
 - On the Intel 960 architecture, the letter must be one of the letters: `D`, `F`, or `T` (in uppercase or lower case).
 - On the HPPA architecture, the letter must be `E` (uppercase 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 the following elements:
 - 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`.

5

Sections and Relocation

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 following documentation discusses sections and their relocation.

- “ld Sections” on page 33
- “as Internal Sections” on page 34
- “Sub-sections” on page 34
- “bss Section” on page 35

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 of relocation, 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 name Directive” on page 63). 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, each time an address in the object file is mentioned, `ld` must know the following criteria.

- 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 $(\text{address}) - (\text{start-address of section})$?
- Is the reference to an address “Program-Counter relative”?

In fact, every address `as` ever uses is expressed as the following.

$$(\text{section}) + (\text{offset into section}).$$

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 documentation, we use the notation `{secname N}` to mean “offset *N* into a specified 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 v}`— where *v* 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 by the assembler and have no meaning except during assembly.

ld Sections

`ld` deals with just four kinds of sections, summarized by the following documentation.

- **named sections, text sections, data sections**
These sections hold your program. `as` and `ld` treat them as separate but equal sections; anything you can say of one section is true for 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, such as constants. The data section of a running program is usually alterable; for example, C variables would be stored in the data section.
- **bss sections**
This section contains zeroed bytes when your program begins running. It is used to hold uninitialized 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 sections**
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 sections**
A catch-all for address references to objects not in preceding sections.

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

horizontal axis.

Partial program #1

text	data	bss
ttttt	dddd	00

Partial program #2

text	data	bss
TTT	DDDD	000

Linked program

text		data	bss
TTT	ttttt	dddd	DDDD 00000 ...

Addresses: 0...

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 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. Sub-sections 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 "Expressions" on page 41.) If you just say `.text`, then `.text 0` is assumed. Likewise, `.data` means `.data 0`. Assembly begins in `text 0`. For instance, use the following example.

```
.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 also ".comm symbol, length Directive" on page 50.

6

Symbols for the GNU Assembler

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. The following documentation discusses more about symbols and the assembler.

- “Labels” (below)
- “Giving Symbols Other Values” on page 38
- “Symbol Names” on page 38
- “The Special Dot Symbol” on page 39
- “Symbol Attributes” on page 39

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 equal sign (=), followed by an expression (see “Expressions” on page 41). This is equivalent to using the `.set` directive. See “.set symbol, expression Directive” on page 65.

Symbol Names

Symbol names begin with a letter or with either a dot (`.`) or an underscore (`_`); on most machines, you can also use `$` in symbol names; exceptions are noted for each architecture in the documentation; see “Machine Dependent Features for the GNU Assembler” on page 71 to locate a specific architecture. That character may be followed by any string of digits, letters, dollar signs (unless exceptions are noted for each architecture), 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 ‘*Mb*’, 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 the following 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 into which `as` is assembling.

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 of `.=.+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.

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 `.common` 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 for a.out File Format

The following documentation discusses symbol attributes for the `a.out` file format.

Descriptor

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (see “.desc symbol, abs-expression Directive” on page 51). A descriptor value means nothing to `as`.

Other

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

Symbol Attributes for COFF Format

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 Format

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) with the “IMPORT and EXPORT assembler directive” documentation.

7

Expressions

The following documentation describes the assembler syntax parts such as expressions, arguments, symbols and operators. An *expression* specifies an address or numeric value. Whitespace may precede and/or follow an expression. An *argument* is like an operand; however, with `as`, the argument has a different meaning. An *operator* is an arithmetic function. See the following documentation for more specific discussion.

- “Empty Expressions” (below)
- “Integer Expressions” on page 42

The result of an expression must be an absolute number, or else an off-set 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 sub-expressions. In other contexts, *arguments* are sometimes called *arithmetic operands*. In this documentation, 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.

Sub-expressions use 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 (see Prefix Operators). Infix operators appear between their arguments (see Infix Operators). Operators may be preceded and/or followed by whitespace.

Prefix Operators

`as` has the following *prefix operators*, each taking one argument, an 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.

- *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.

- *Intermediate Precedence*

- | (Bitwise Inclusive Or)

- & (Bitwise And)

- ^ (Bitwise Exclusive Or)

- ! (Bitwise Or Not)

- *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.

8

Assembler Macro Directives

The following list is of the directives that are available regardless of the target machine configuration for `as`, the GNU assembler. All assembler directives have names that begin with a period (`.`). The rest of the name is letters, usually in lower case. With each directive is the location of their descriptions. For additional directives pertinent to each architecture, see “Machine Dependent Features for the GNU Assembler” on page 71.

- “.abort Directive” on page 48
- “.ABORT Directive” on page 48
- “.align abs-expr, abs-expr, abs-expr Directive” on page 48
- “.ascii “string”... Directive” on page 49
- “.asciz “string”... Directive” on page 49
- “.balign[wl] abs-expr, abs-expr, abs-expr Directive” on page 49
- “.byte expressions Directive” on page 50
- “.comm symbol, length Directive” on page 50
- “.data subsection Directive” on page 50
- “.def name Directive” on page 50
- “.desc symbol, abs-expression Directive” on page 51
- “.dim Directive” on page 51

- “.double flonums Directive” on page 51
- “.eject Directive” on page 51
- “.else Directive” on page 51
- “.elseif Directive” on page 51
- “.end Directive” on page 52
- “.endif Directive” on page 52
- “.endfunc Directive” on page 52
- “.endif Directive” on page 52
- “.equ symbol, expression Directive” on page 52
- “.equiv symbol, expression Directive” on page 52
- “.err Directive” on page 53
- “.exitm Directive” on page 53
- “.extern Directive” on page 53
- “.fail expression Directive” on page 53
- “.file string Directive” on page 53
- “.fill repeat, size, value Directive” on page 53
- “.float flonums Directive” on page 54
- “.func name,label Directive” on page 54
- “.global symbol, .globl symbol Directive” on page 54
- “.hword expressions Directive” on page 54
- “.ident Directive” on page 55
- “.if absolute expression Directive” on page 55
- “.include “file Directive” on page 56
- “.int expressions Directive” on page 56
- “.irp symbol, values... Directive” on page 56
- “.irpc symbol, values... Directive” on page 57
- “.lcomm symbol, length Directive” on page 57
- “.lflags Directive” on page 57
- “.line line-number Directive” on page 58
- “.linkonce type Directive” on page 58
- “.list Directive” on page 59
- “.ln line-number Directive” on page 59
- “.long expressions Directive” on page 59
- “.macro Directive” on page 59
- “.mri val Directive” on page 60

- “.nolist Directive” on page 60
- “.octa bignums Directive” on page 61
- “.org new-lc, fill Directive” on page 61
- “.p2align[w] abs-expr, abs-expr, abs-expr Directive” on page 61
- “.print string Directive” on page 62
- “.psize lines, columns Directive” on page 62
- “.purgem name Directive” on page 62
- “.quad bignums Directive” on page 62
- “.rept count Directive” on page 63
- “.sbttl “subheading” Directive” on page 63
- “.scl class Directive” on page 63
- “.section name Directive” on page 63
- “.set symbol, expression Directive” on page 65
- “.short expressions Directive” on page 65
- “.single flonums Directive” on page 65
- “.size Directive” on page 65
- “.sleb128 expressions Directive” on page 66
- “.skip size, fill Directive” on page 66
- “.space size, fill Directive” on page 66
- “.stabd, .stabn, and .stabs Directives” on page 66
- “.string “str” Directive” on page 67
- “.struct expression Directive” on page 67
- “.symver Directive” on page 68
- “.tag structname Directive” on page 68
- “.text subsection Directive” on page 68
- “.title “heading” Directive” on page 69
- “.type int Directive” on page 69
- “.val addr Directive” on page 69
- “.uleb128 expressions Directive” on page 69
- “.word expressions Directive” on page 69
- “.zero size Directive” on page 70

IMPORTANT! One day the `.abort` and `.line` directives won't work. They are included for compatibility with older assemblers.

.abort Directive

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 to tell the assembler to quit also. One day `.abort` will not be supported.

.ABORT Directive

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*, *abs-expr* Directive

Pad the location counter (in the current subsection) to a particular storage boundary.

The first expression (*abs-expr*, which must be absolute) is the alignment required, as the following discussion details.

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 normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. To omit the fill value (the second argument) entirely, use two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The way the required alignment is specified varies from system to system. For some systems 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 is 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 `as` must emulate. `as` also provides `.balign` and `.p2align` directives, which have a consistent behavior across all

architectures (specific to `as`); see “.`balign[wl]` *abs-expr*, *abs-expr*, *abs-expr* Directive” on page 49 and “.`p2align[wl]` *abs-expr*, *abs-expr*, *abs-expr* Directive” on page 61.

.ascii “string”... Directive

`.ascii` expects zero or more string literals (see “Strings” on page 27) separated by commas. It assembles each string (with no automatic trailing zero byte) into consecutive addresses.

.asciz “string”... Directive

`.asciz` is just like `.ascii`, but each string is followed by a zero byte. The `z` in `.asciz` stands for *zero*.

.balign[wl] *abs-expr*, *abs-expr*, *abs-expr* Directive

Pad the location counter (in the current subsection) to a particular storage boundary.

The first expression (*abs-expr*, 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 normally zero. However, on some systems, if the section is marked as containing code and the fill value is omitted, the space is filled with no-op instructions.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. To omit the fill value (the second argument) entirely, use two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.balignw` and `.balignl` directives are variants of the `.balign` directive. The `.balignw` directive treats the fill pattern as a two byte word value. The `.balignl` directive treats the fill pattern as a four byte longword value. For example, `.balignw 4, 0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value `0x368d` (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

.byte *expressions* Directive

`.byte` expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

.comm *symbol, length* Directive

`.comm` declares a common symbol named *symbol*. When linking, a common symbol in one object file may be merged with a defined or common symbol of the same name in another object file. If `ld` does not see a definition for the symbol—just one or more common symbols—it will allocate *length* bytes of uninitialized memory. *length* must be an absolute expression. If `ld` sees multiple common symbols with the same name, and they do not all have the same size, it will allocate space using the largest size.

When using ELF, the `.comm` directive takes an optional third argument. This is the desired alignment of the symbol, specified as a byte boundary (for example, an alignment of 16 means that the least significant 4 bits of the address should be 0). The alignment must be an absolute expression, and it must be a power of 2. If `ld` allocates uninitialized memory for the common symbol, it will use the alignment when placing the symbol. If no alignment is specified, `as` will set the alignment to the largest power of 2 less than or equal to the size of the symbol, up to a maximum of 16.

The syntax for `.comm` differs slightly on the HPPA. The syntax is *symbol.comm, length*; *symbol* is optional.

.data *subsection* Directive

`.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* Directive

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-expression* Directive

This directive sets the descriptor of the symbol (see “Symbol Attributes” on page 39) 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 Directive

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* Directive

`.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. For each architecture’s documentation, see “Machine Dependent Features for the GNU Assembler” on page 71.

.eject Directive

Force a page break at this point, when generating assembly listings.

.else Directive

`.else` is part of the `as` support for conditional assembly; see “`.if` absolute expression Directive” on page 55. It marks the beginning of a section of code to be assembled if the condition for the preceding `.if` was false.

.elseif Directive

`.elseif` is for conditional assembly, shorthand for beginning a new `.if` block that would otherwise fill the entire `.else` section. see “`.if` absolute expression Directive” on page 55.

.end Directive

`.end` marks the end of the assembly file. `as` does not process anything in the file past the `.end` directive.

.endef Directive

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.

.endfunc Directive

`.endfunc` marks the end of a function specified with the `.func` directive; see “`.func name,label Directive`” on page 54.

.endif Directive

`.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 “`.if absolute expression Directive`” on page 55.

.equ *symbol*, *expression* Directive

This directive sets the value of *symbol* to *expression*. It is synonymous with `.set`; see “`.set symbol, expression Directive`” on page 65. The syntax for `equ` on the HPPA is `symbol.equ expression`.

.equiv *symbol*, *expression* Directive

The `.equiv` directive is like the `.equ` and the `.set` directives, except the assembler will signal an error if *symbol* is already defined.

Except for the contents of the error message, the following input is roughly equivalent.

```
.ifdef SYM
.err
.endif
.equ SYM,VAL
```

.err Directive

If `as` assembles a `.err` directive, it will print an error message and, unless the `-z` option was used, it will not generate an object file. This can be used to signal error an unconditionally compiled code.

.exitm Directive

Exit early from the current macro definition. See “`.macro` Directive” on page 59.

.extern Directive

`.extern` is accepted in the source program—for compatibility with other assemblers—but it is ignored. `as` treats all undefined symbols as external.

.fail *expression* Directive

Generates an error or a warning. If the value of *expression* is 500 or more, `as` will print a warning message. If the value is less than 500, `as` will print an error message. The message will include the value of *expression*. This can occasionally be useful inside complex nested macros or conditional assembly.

.file *string* Directive

`.file` (which may also be spelled `.app-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, 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. For each architecture’s documentation, see “Machine Dependent Features for the GNU Assembler” on page 71.

.fill *repeat, size, value* Directive

result, *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* Directive

.float 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. For each architecture's documentation, see "Machine Dependent Features for the GNU Assembler" on page 71.

.func *name, label* Directive

.func emits debugging information to denote function, *name*, and is ignored unless the file is assembled with debugging enabled. Only *--gstabs* is currently supported.

label is the entry point of the function and, if omitted, *name* prepends with leading char.

'leading char' is usually an underscore (`_`) or nothing, depending on the target. All functions are currently defined to have `void` return type. The function must be terminated with *.endfunc*.

.global *symbol, .globl symbol* Directive

.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 "HPPA Dependent Features" on page 99.

.hword *expressions* Directive

.hword expects zero or more expressions, and emits a 16 bit number for each

expression.

This directive is a synonym for `.short`; depending on the target architecture, it may also be a synonym for `.word`.

`.ident` Directive

`.ident` 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` Directive

`.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 “`.endif` Directive” on page 52); optionally, you may include code for the alternative condition, flagged by `.else` (see “`.else` Directive” on page 51). The following variants of `.if` are also supported:

`.ifdef symbol`

Assembles the preceding section of code if the specified *symbol* has been defined.

`.ifc string1,string2`

Assembles the preceding section of code if the two strings are the same. The strings may be optionally quoted with single quotes. If they are not quoted, the first string stops at the first comma, and the second string stops at the end of the line. Strings which contain whitespace should be quoted. The string comparison is case sensitive.

`.ifeq absolute expression`

Assembles the preceding section of code if the argument is zero.

`.ifeqs string1,string2`

Another form of `.ifc`. The strings must be quoted using double quotes.

`.ifge absolute expression`

Assembles the preceding section of code if the argument is greater than or equal to zero.

`.ifgt absolute expression`

Assembles the preceding section of code if the argument is greater than zero.

`.iflt absolute expression`

Assembles the preceding section of code if the argument is less than or equal to zero.

`.iflt absolute expression`

Assembles the preceding section of code if the argument is less than zero.

`.ifnc string1, string2`

Like `.ifc`, but the sense of the test is reversed; this assembles the preceding section of code if the two strings are not the same.

`.ifndef symbol`

`.ifnotdef symbol`

Assembles the preceding section of code if the specified *symbol* has not been defined. Both spelling variants are equivalent.

`.ifne absolute expression`

Assembles the preceding section of code if the argument is not equal to zero (in other words, this is equivalent to `.if`).

`.ifnes string1, string2`

Like `.ifeqs`, but the sense of the test is reversed; this assembles the preceding section of code if the two strings are not the same.

`.include` *file* Directive

`.include` 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 “Search Path for `.include` Specifications: `-I path Option`” on page 17). Quotation marks are required around *file*.

`.int expressions` Directive

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 uses.

`.irp symbol, values...` Directive

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`.

```
.irp      param,1,2,3
move    d\param,sp@-
```

```
.endr
```

The previous statement is equivalent with assembling the following statement.

```
move    d1, sp@-
move    d2, sp@-
move    d3, sp@-
```

`.irpc symbol, values... Directive`

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`.

```
.irpc    param, 123
move     d\param, sp@-
.endr
```

For example, assembling the previous statement is equivalent to assembling the following declaration.

```
move     d1, sp@-
move     d2, sp@-
move     d3, sp@-7.32
```

`.lcomm symbol, length Directive`

Reserve *length* (an absolute expression) bytes for a local common denoted by *symbol*. The 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 “`.global symbol, .globl symbol Directive`” on page 54), so it’s normally not visible to `ld`.

Some targets permit a third argument with `.lcomm`. This argument specifies the desired alignment of the symbol in the bss section.

The syntax for `.lcomm` differs slightly on the HPPA. The syntax is `symbol.lcomm, length; symbol` is optional.

`.lflags Directive`

`as` accepts this directive, for compatibility with other assemblers, but ignores it.

`.line` *line-number* Directive

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. This directive may be obsolete; it is recognized only for compatibility with existing assembler programs.

WARNING! For the AMD29K configuration, 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.

`.linkonce` *type* Directive

Mark the current section so that the linker only includes a single copy of it. This may be used to include the same section in several different object files, but ensure that the linker will only include it once in the final output file. The `.linkonce` pseudo-op must be used for each instance of the section. Duplicate sections are detected based on the section name, so it should be unique. This directive is only supported by a few object file formats; as of this writing, the only object file format which supports it is the Portable Executable format used on Windows NT. The *type* argument is optional. If specified, it must be one of the strings in the following list. Not all types may be supported on all object file formats.

- `discard`
Silently discard duplicate sections. This is the default.
- `one_only`
Warn if there are duplicate sections, but still keep only one copy.
- `same_size`
Warn if any of the duplicates have different sizes.
- `same_contents`
Warn if any of the duplicates do not have exactly the same contents.

The following example shows the usage.

```
.linkonce same_size
```

.list Directive

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 “Enable Listings: `-a[cdhlms]` Options” on page 16), the initial value of the listing counter is one.

.ln *line-number* Directive

`.ln` is a synonym for `.line`.

.long *expressions* Directive

`.long` is the same as `.int`, see “`.int` expressions Directive” on page 56.

.macro Directive

The `.macro` and `.endm` commands allow you to define macros that generate assembly output. For example, the following definition specifies a macro `sum` that puts a sequence of numbers into memory.

```
.macro      sum
.long      \from
.if        \to-\from
sum        "(\from+1)", \to
.endif     from=0, to=5
endm
```

With that definition, `SUM 0,5` is equivalent to the following 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 `=default`. For example, the following 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`
Begins the definition of a macro called `plus1`, taking 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` has 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*.

`.mri val` Directive

If `val` is non-zero, this tells `as` to enter MRI mode. If `val` is zero, this tells `as` to exit MRI mode. This change affects code assembled until the next `.mri` directive, or until the end of the file. See “Assemble in MRI Compatibility Mode: -M Option” on page 18.

`.nolist` Directive

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` Directive

`.octa` 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-loc, fill` Directive

Advance the location counter of the current section to *new-loc*. *new-loc* 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-loc* has the wrong section, the `.org` directive is ignored. To be compatible with former assemblers, if the section of *new-loc* is absolute, `as` issues a warning, then pretends the section of *new-loc* 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-loc* may not be undefined. If you really detest this restriction we eagerly await a chance to share your improved assembler.

IMPORTANT! The origin is relative to the start of the section, not to the start of the subsection. This is compatible with other 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[w1] abs-expr, abs-expr, abs-expr` Directive

`.p2align[w1]` 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 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.

The third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing

the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. Omit the fill value (the second argument) entirely by using two commas after the required alignment; this can be useful if you want the alignment to be filled with no-op instructions when appropriate.

The `.p2alignw` and `.p2alignl` directives are variants of the `.p2align` directive. The `.p2alignw` directive treats the fill pattern as a two byte word value. The `.p2alignl` directive treats the fill pattern as a four byte longword value. For example, `.p2alignw 2,0x368d` will align to a multiple of 4. If it skips two bytes, they will be filled in with the value `0x368d` (the exact placement of the bytes depends upon the endianness of the processor). If it skips 1 or 3 bytes, the fill value is undefined.

`.print` *string* Directive

`as` prints *string* on standard output during assembly; put *string* in double quotes.

`.psize` *lines, columns* Directive

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 except for those explicitly specified with `.eject`.

`.purgem` *name* Directive

Undefine the macro name, so that later uses of the string will not be expanded; see “`.macro` Directive” on page 59.

`.quad` *bignums* Directive

`.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* Directive

Repeat the sequence of lines between the `.rept` directive and the next `.endr` directive *count* times (where *count* stands for the appropriate sequence).

```
.rept          3
.long         0
.endr
```

For example, assembling the previous statement is equivalent to assembling the following directive.

```
.long         0
.long         0
.long         0
```

.sbt1 "*subheading*" Directive

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.

.sc1 *class* Directive

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 `.sc1` directive is primarily associated with COFF output; when configured to generate `b.out` output format, as accepts this directive but ignores it.

.section *name* Directive

Use the `.section` directive to assemble the following code into a section called *name*. 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. For COFF targets, the `.section` directive is used in one of the following ways.

```
.section name[,"flags"]
.section name[, subsegment]
```

If the optional argument is quoted, it is taken as flags to use for the section. Each flag is a single character. The following flags are recognized.

b	bss section (uninitialized data)
n	section is not loaded
w	writable section
d	data section
r	read-only section
x	executable section

If no flags are specified, the default flags depend upon the section name. If the section name is not recognized, the default will be for the section to be loaded and writable.

If the optional argument to the `.section` directive is not quoted, it is taken as a subsequent number (see “Sub-sections” on page 34).

For ELF targets, the `.section` directive is used in the following way.

```
.section name[, "flags"[, @type]]
```

The optional *flags* argument is a quoted string which may contain any combination of the following characters.

a	section is allocatable
w	section is writable
x	section is executable

The optional *type* argument may contain one of the following constants.

```
@progbits
```

section contains data

```
@nobits
```

section does not contain data (that is, section only occupies space)

If no flags are specified, the default flags depend on the section name. If the section name is not recognized, the default will be for the section to have none of the previously described flags; it will not be allocated in memory, nor will it be writable or executable. The section will contain data.

For ELF targets, the assembler supports another type of `.section` directive for compatibility with the Solaris assembler, as with the following declaration.

```
.section "name"[, flags...]]
```

Notice that the section name is quoted. There may be a sequence of the following comma-separated flags.

```
#alloc
    section is allocatable
#write
    section is writable
#excinstr
    section is executable
```

`.set` *symbol, expression* Directive

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 “Symbol Attributes” on page 39.) 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* Directive

`.short` is normally the same as `.word`. See “.word expressions Directive” on page 69.

In some configurations, however, `.short` and `.word` generate numbers of different lengths; for a specific architecture's documentation, see “Machine Dependent Features for the GNU Assembler” on page 71.

`.single` *flonums* Directive

`.single` 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; for a specific architecture's documentation, see “Machine Dependent Features for the GNU Assembler” on page 71.

`.size` Directive

This directive is generated by compilers to include auxiliary debugging information in the symbol table. It is only permitted inside `.def/.endif` pairs. `.size` is only meaningful when generating COFF format output; when `as` is generating `b.out`, it accepts this directive but ignores it.

.sleb128 *expressions* Directive

`.sleb128` stands for “signed little endian base 128.” This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See also “`.uleb128 expressions Directive`” on page 69.

.skip *size, fill* Directive

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. This is the same as `.space`.

.space *size, fill* Directive

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

WARNING! `.space` has a completely different meaning for HPPA targets; use `.block` as a substitute. See *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001) for the meaning of the `.space` directive. See “HPPA Dependent Features” on page 99.

This directive is for compatibility with other AMD 29K assemblers.

WARNING! In most versions of the GNU assembler, the `.space` directive has the effect of `.block`. For each architecture’s documentation, see “Machine Dependent Features for the GNU Assembler” on page 71.

.stabd, .stabn, and .stabs Directives

There are three directives that begin `.stab`. All emit symbols for use by symbolic debuggers (see “Symbols” on page 25). 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, as the following descriptions clarify.

- *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 set to the low 8 bits of this expression. Any bit pattern is permitted, but `1d` and debuggers choke on silly bit patterns.

- *other*
An absolute expression, the symbol's other attribute set to the low 8 bits of this expression.
- *desc*
An absolute expression, the symbol's descriptor 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 any of the following `.stabd`, `.stabn`, or `.stabs` statements, 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, relocatability. 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" Directive

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 "Strings" on page 27

.struct *expression* Directive

Switch to the absolute section, and set the section offset to *expression*, which must be an absolute expression. The following example shows the usage.

```
.struct 0
field1:
    .struct field1 + 4
field2:
    .struct field2 + 4
field3:
```

The previous example's input would define the symbol, `field1`, to have the value 0, the symbol, `field2`, to have the value 4, and the symbol, `field3`, to have the value 8.

Assembly would be left in the absolute section, and you would need to use a `.section` directive of some sort to change to some other section before further assembly.

`.symver` Directive

Use the `.symver` directive to bind symbols to specific version nodes within a source file. This is only supported on ELF platforms, and is typically used when assembling files to be linked into a shared library. Use this in objects to be bound into an application itself so as to override a versioned symbol from a shared library. For ELF targets, the `.symver` directive is used like the following declaration shows

```
.symver name, name2@nodename
```

The symbol, *name*, must exist and be defined within the file being assembled. The `.versym` directive effectively creates a symbol alias with the name *name2@nodename*, and in fact the main reason that we just don't try and create a regular alias is that the `@` character isn't permitted in symbol names. The *name2* part of the name is the actual name of the symbol by which it will be externally referenced. The name, *name*, itself is merely a name of convenience that is used so that it is possible to have definitions for multiple versions of a function within a single source file, and so that the compiler can unambiguously know which version of a function is being mentioned. The *nodename* portion of the alias should be the name of a node specified in the version script supplied to the linker when building a shared library. If you are attempting to override a versioned symbol from a shared library, then *nodename* should correspond to the nodename of the symbol you are trying to override.

`.tag structname` Directive

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` Directive

`.text` 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"* Directive

Use *heading* as the title (second line, immediately after the source file name and pagenumber) when generating assembly listings. `.title` affects subsequent pages, as well as the current page if it appears within ten lines of the top of a page.

.type *int* Directive

This directive, permitted only within `.def/.endif` 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* Directive

This directive, permitted only within `.def/.endif` 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.

.uleb128 *expressions* Directive

`uleb128` stands for “unsigned little endian base 128.” This is a compact, variable length representation of numbers used by the DWARF symbolic debugging format. See “`.sleb128 expressions Directive`” on page 66.

.word *expressions* Directive

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 for which the assembly builds.

WARNING! To support compilers on machines with a 32-bit address space, having less than 32-bit addressing, this requires special treatment. If the machine of interest to you does 32-bit addressing (or doesn't require it), ignore this issue. For documentation for specific processors, see “Machine Dependent Features for the GNU Assembler” on page 71. In order to assemble compiler output into something that works, `as` occasionally 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.

`.zero` *size* Directive

This directive fills a space *size* bytes long with a value of zero.

9

Machine Dependent Features for the GNU Assembler

The machine instruction sets are (almost by definition) different on each machine where the GNU assembler, `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.

There is discussion on most of these differences, though there aren't details on any machine's instruction set. For details on that subject, see the specific hardware manufacturer's documentation.

The following architectures are discussed.

- “AMD 29K Dependent features” on page 73
- “ARC Dependent Features” on page 77
- “ARM Dependent Features” on page 79
- “AT&T and Intel x86 Dependent Features” on page 83
- “Hitachi H8/300 Dependent Features” on page 91
- “Hitachi H8/500 Dependent Features” on page 93
- “Hitachi SH Dependent Features” on page 95
- “HPPA Dependent Features” on page 99
- “Intel StrongARM Dependent Features” on page 105

- “Intel 960 Dependent Features” on page 107
- “Intel x86 and IA64 Dependent Features” on page 111
- “MIPS Dependent Features” on page 113
- “Mitsubishi D10V Dependent Features” on page 119
- “Mitsubishi D30V Dependent Features” on page 123
- “Mitsubishi M32R Dependent Features” on page 129
- “Motorola 68K Dependent Features” on page 133
- “NEC V850 Dependent Features” on page 141
- “PowerPC Dependent Features” on page 147
- “Sun Dependent Features” on page 149
- “Vax Dependent Features” on page 153
- “Zilog Z8000 Dependent Features” on page 159

10

AMD 29K Dependent features

The following documentation discusses the features pertinent to the AMD (Advanced Micro Devices, Inc.) 29000 machines regarding the GNU assembler.

`as` has no additional command line options for the AMD 29K family.

The macro syntax used on the AMD 29K is like syntax in the AMD 29K Family Macro Assembler Specification. Normal GNU assembler macros should still work. For more information on the AMD 29K family, see *AMD 29000 User's Manual*, published by AMD.

`;` is the line comment character.

The question mark character, `?`, is permitted in identifiers (but *may not begin* an identifier).

General-purpose registers are represented by predefined symbols of the form `GR nnn` (for global registers) or `LR nnn` (for local registers), where nnn represents a number between 0 and 127, written with no leading zeros. The leading letters may be in either upper or lower case; for example, `gr13` and `LR7` are both valid register names.

You may also refer to general-purpose registers by specifying the register number as the result of an expression (prefixed with `%%` to flag the expression as a register number), as in the following example.

```
%% expression
```

`expression` in the previous example's case must be an absolute expression evaluating

to a number between 0 and 255. The range [0, 127] refers to global registers, and the range [128, 255] to local registers.

In addition, `as` understands the protected special-purpose register names for the AMD 29K family like the following names.

<code>vab</code>	<code>chd</code>	<code>pc0</code>
<code>ops</code>	<code>chc</code>	<code>pc1</code>
<code>cps</code>	<code>rbp</code>	<code>pc2</code>
<code>cfg</code>	<code>tmc</code>	<code>mmu</code>
<code>cha</code>	<code>tmr</code>	<code>lru</code>

These unprotected special-purpose register names are also recognized:

<code>ipc</code>	<code>alu</code>	<code>fpe</code>
<code>ipa</code>	<code>bp</code>	<code>inte</code>
<code>ipb</code>	<code>fc</code>	<code>fps</code>
<code>q</code>	<code>cr</code>	<code>exop</code>

The AMD 29K family uses IEEE floating-point numbers.

The following documentation discusses the features pertinent to the AMD 29K regarding the machine directives for the GNU assembler.

`.block size, fill`

This directive emits *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero. In other versions of the GNU assembler, this directive is called `.space`.

`.cputype`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.file`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers. In other versions of the GNU assembler, `.file` is used for the directive called `.app-file` in the AMD 29K assembler.

`.line`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.sect`

This directive is ignored; it is accepted for compatibility with other AMD 29K assemblers.

`.use section name`

Establishes the section and subsection for the following code; section name may be one of `.text`, `.data`, `.data1`, or `.lit`. With one of the first three *section name* options, `.use` is equivalent to the machine directive, *section name*; the remaining case, `.use .lit`, is the same as `.data 200`. This directive is accepted only on non-COFF targetted versions of the AMD 29K toolchain.

`.space`

This directive is ignored; it is accepted for compatibility with other AMD 29K

assemblers. In other versions of the GNU assembler, `.space` is used for the `.block` directive for the AMD 29K assembler.

`as` implements all the standard AMD 29K opcodes. No additional pseudo-instructions are needed on this family.

ARC Dependent Features

The following documentation discusses the features pertinent to the ARC processor regarding the GNU assembler.

The ARC chip family includes several successive levels (or other variants) of chip, using the same core instruction set, but including a few additional instructions at each level.

By default, `as` assumes the core instruction set (ARC base). The `.cpu` pseudo-op is intended to be used to select the variant.

`-EB`

Generate big-endian format output

`-EL`

Generate little-endian.format output.

The ARC processor family currently does not have hardware floating point support. Software floating point support is provided by `gcc` and uses IEEE floating-point numbers.

The ARC version of `as` supports the following additional machine directive.

`.cpu name`

This must be followed by the desired CPU. The ARC is intended to be customizable, `.cpu` is used to select the desired variant (although, currently, `arc` is the only substitution for `name`).

ARM Dependent Features

The following documentation discusses the features pertinent to the ARM and StrongARM processors regarding the GNU assembler.

The ARM syntax is based on the syntax in ARM's *ARM7 Architecture Manual*. For a list of available generic assembler options, see "Using `as` and Its Options" on page 8.

Both the assembler and the compiler support hardware floating point. For detailed information on the ARM7/7T machine instruction set, see *ARM7 Series Instruction Manual*. The GNU assembler implements all the standard opcodes for the ARM7/7T machine instruction set. For detailed information on the StrongARM machine instruction set, see the *ARM Architecture Reference Manual* and Intel's *StrongARM Reference Manual*. The GNU assembler implements all the standard opcodes, including both the standard ARM opcodes and Intel's extensions.

@ is for indicating the start of a comment that extends to the end of the line.

The following command line options are ARM and StrongARM specific assembler command line options.

```
-m[arm][1|2|250|3|6|7[t][d][m][i]|8[10]|9[20][tdmi]]  
-mstrongarm[110[0]]
```

Select processor variant.

```
-m[arm]v[2|2a|3|3m|4|4t|5[t][e]]
```

Select architecture variant.

- `-mthumb`
Only allow THUMB instructions.
- `-marm`
Allow any instruction.
- `-mfpa10`
Select the v1.0 floating point architecture.
- `-mfpa11`
Select the v1.1 floating point architecture.
- `-mfpe-old`
Don't allow floating-point multiple instructions.
- `-mno-fpu`
Don't allow any floating-point instructions.
- `-mthumb-interwork`
Specify that the code has been generated for interworking between THUMB and ARM code.
- `-mapcs-32`
Mark the code as supporting the 32 bit variant of the ARM procedure calling standard. This is the default.
- `-mapcs-26`
Mark the code as supporting the 26 bit variant of the ARM procedure calling standard.
- `-EB`
Assemble code for a big endian CPU.
- `-EL`
Assemble code for a little endian CPU. This is the default.
- `-moabi`
Selects the old ABI for calling between functions.
- `-mapcs-float` | `-mapcs-reentrant`
Select which ABI variant is in use.
- `-k`
Specify that PIC code has been generated.

The following machine directives are for the ARM and StrongARM families of processors.

- `.arm`
The subsequent code uses the ARM instruction set.
- `.thumb`
The subsequent code uses the THUMB instruction set.
- `.code 16`
An alias for `.thumb`.
- `.code 32`
An alias for `.arm`.

.force_thumb

Subsequent code uses the THUMB instruction set, and should be assembled even if the target processor does not support THUMB instructions.

.thumb_func

Subsequent label is the name of function which has been encoded using THUMB instructions, rather than ARM instructions.

ldr *register*, = *expression*

Loads the value of *expression* into *register*. If the value is one that can be constructed by a MOV or MVN instruction then this will be used. Otherwise, the value will be placed into the nearest literal pool (if it is not there already) and a PC relative LDR instruction will be used to load the value into the register.

.ltorg

Dumps the current accumulated literal pool entries into the current section. This directive does not generate any jump instructions around the pool.

.pool

Synonym for **.ltorg**.

.req

Creates an alias for a register name. The following example shows syntax.

```
alias .req register_name
```

The following example shows usage.

```
overflow .req r1
```

Once the alias has been created, it can be used in the assembler sources at any place where a register name would be expected.

.align ABS-EXPR, ABS-EXPR, ABS-EXPR

Pads the location counter (in the current subsection) to a particular storage boundary. The first expression, **ABS-EXPR** (which must be absolute), is the alignment required, expressed as the number of low-order zero bits the location counter must have after advancement. For example, **.align 3** 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, **ABS-EXPR** (also absolute), gives the fill 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 third expression is also absolute, and is also optional. If it is present, it is the maximum number of bytes that should be skipped by this alignment directive. If doing the alignment would require skipping more bytes than the specified maximum, then the alignment is not done at all. You can omit the fill value (the second argument) entirely by simply using two commas after the required alignment. There is one special case. If the first expression evaluates to 0 it is treated as if it were 2. This is for compatibility with ARM's own assembler, which uses **.align 0** to mean align to a word boundary.

.even

Align the following code or data on an even boundary (that is, 2 byte).

`.thumb-set`

Like the standard `.set` directive, except that the created symbol is also marked as being a THUMB function symbol.

The following tables list the register names supported for ARM, using the *register name, register number* format.

Table 1: General registers

r0: 0	r1: 1	r2: 2	r3: 3
r4: 4	r5: 5	r6: 6	r7: 7
r8: 8	r9: 9	r10: 10	r11: 11
r12: 12	r13: 13:	r14: 14	r15: 15

Table 2: APCS names for the general registers

a1: 0	a2: 1	a3: 2	a4: 3:
v1: 4	v2: 5	v3: 6	v4: 7:
v5: 8	v6: 9	sb: 9	v7: 10
s1: 10	fp: 11	ip: 12	sp: 13
lr: 14	pc: 15		

Table 3: Floating point registers

f0: 16	f1: 17	f2: 18	f3: 19
f4: 20	f5: 21	f6: 22	f7: 23
c0: 32	c1: 33	c2: 34	c3: 35
c4: 36	c5: 37	c6: 38	c7: 39
c8: 40	c9: 41	c10: 42	c11: 43
c12: 44	c13: 45	c14: 46	c15: 47
cr0: 32	cr1: 33	cr2: 34	cr3: 35
cr4: 36	cr5: 37	cr6: 38	cr7: 39
cr8: 40	cr9: 41	cr10: 42	cr11: 43
cr12: 44	cr13: 45	cr14: 46	cr15: 47

Table 4: Accumulators

acc0: 0

13

AT&T and Intel x86 Dependent Features

The following documentation discusses the features pertinent to the AT&T and Intel processors regarding the GNU assembler. See also “Intel x86 and IA64 Dependent Features” on page 111 for more information pertaining to the Intel usage for the GNU assembler.

In order to maintain compatibility with the output of GCC, the GNU assembler supports AT&T System V/386 assembler syntax. This is quite different from Intel syntax. We mention these differences because almost all 80386 documents used only Intel syntax. Notable differences between the two syntaxes are:

- AT&T immediate operands are preceded by `$`; Intel immediate operands are undelimited (Intel `push 4` is AT&T `pushl $4`). AT&T register operands are preceded by `%`; Intel register operands are undelimited. AT&T absolute (as opposed to PC relative) `jump/call` operands are prefixed by `*`; they are undelimited in Intel syntax.
- AT&T and Intel syntax use the opposite order for source and destination operands. Intel `add eax, 4` is `addl $4, %eax`. The `source, dest` convention is maintained for compatibility with previous Unix assemblers.
- In AT&T syntax the size of memory operands is determined from the last character of the opcode name. Opcode suffixes of `b`, `w`, and `l` specify byte (8-bit), word (16-bit), and long (32-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (*not* the opcodes themselves) with `byte ptr`,

word ptr, and dword ptr. Thus, Intel `mov al, byte ptr foo` is `movb foo, %al` in AT&T syntax.

- Immediate form long jumps and calls are `lcall/ljmp $ section, $offset` in AT&T syntax; the Intel syntax is `call/jmp far section: offset`. Also, the far return instruction is `lret $stack-adjust` in AT&T syntax; Intel syntax is `ret far stack-adjust`.
- The AT&T assembler does not provide support for multiple section programs. Unix style systems expect all programs to be single sections.

Instruction mnemonics are suffixed with one character modifiers which specify the size of operands. `b`, `w`, and `l` specify byte, word, and long operands. `as` tries to fill in the missing suffix based on the destination register operand (the last one by convention). Thus, `mov %ax, %bx` is equivalent to `movw %ax, %bx`; also, `mov $1, %bx` is equivalent to `movw $1, %bx`; this is incompatible with the AT&T UNIX assembler, which assumes that a missing mnemonic suffix implies long operand size (this incompatibility does not affect compiler output since compilers always explicitly specify the mnemonic suffix.) Almost all opcodes have the same names in AT&T and Intel format. There are a few exceptions. The sign extend and zero extend instructions need two sizes to specify them. They need a size to sign/zero extend from and a size to zero extend *to*. This is accomplished by using two opcode suffixes in AT&T syntax. Base names for sign extend and zero extend are `movs...` and `movz...` in AT&T syntax (`movsx` and `movzx` in Intel syntax). The opcode suffixes are tacked on to this base name, the *from* suffix before the *to* suffix. Thus, `movsbl %al, %edx` is AT&T syntax for “move sign extend *from* `%al` *to* `%edx`.” Possible suffixes are `bl` (from byte to long), `bw` (from byte to word), and `wl` (from word to long). The following Intel syntax conversion instructions are called `cbtw`, `cwtl`, `cwtd`, and `cltd` in AT&T naming convention. `as` accepts either naming for the following instructions.

- `cbw`
sign-extend byte in `%al` to word in `%ax`
- `cwde`
sign-extend word in `%ax` to long in `%eax`
- `cwd`
sign-extend word in `%ax` to long in `%dx:%ax`
- `cdq`
sign-extend dword in `%eax` to quad in `%edx:%eax`

Far call/jump instructions are `lcall` and `ljmp` in AT&T syntax, but `call far` and `jump far` in Intel convention.

Register operands are always prefixed with `%`. The 80386 registers consist of the following register operands.

- 8 32-bit registers:
`%eax` (the accumulator), `%ebx`, `%ecx`, `%edx`, `%edi`, `%esi`, `%ebp` (the frame pointer), and `%esp` (the stack pointer).

- *8 16-bit low-ends:*
%ax, %bx, %cx, %dx, %di, %si, %bp, and %sp.
- *8 8-bit registers*
%ah, %al, %bh, %bl, %ch, %cl, %dh, and %dl (these are the high-bytes and low-bytes of %ax, %bx, %cx, and %dx)
- *6 section registers:*
%cs (*code section*), %ds (*data section*), %ss (*stack section*), %es, %fs, and %gs.
- *3 processor control registers:*
%cr0, %cr2, and %cr3.
- *6 debug registers:*
%db0, %db1, %db2, %db3, %db6, and %db7.
- *2 test registers:*
%tr6 and %tr7.
- *8 floating point register stack:*
%st, or, equivalently, %st(0), %st(1), %st(2), %st(3), %st(4), %st(5), %st(6), and %st(7).

Instruction prefixes are used to modify the following instruction. They are used to repeat string instructions, to provide section overrides, to perform bus lock operations, and to change operand and address size. Most instructions that normally operate on 32-bit operands will use 16-bit operands if the instruction has an ‘operand size’ prefix. Instruction prefixes are best written on the same line as the instruction upon which they act. For example, the ‘scas’ (scan string) instruction is repeated with the following declaration.

```
repne
scas
```

The following is a list of instruction prefixes:

- Section override prefixes: *cs*, *ds*, *ss*, *es*, *fs* and *gs*. These are automatically added by using the *section: memory-operand* form for memory references.
- Operand/address size prefixes *data16* and *addr16* change 32-bit operands/addresses into 16-bit operands/addresses, while *data32* and *addr32* change 16-bit ones (in a *.code16* section) into 32-bit operands/addresses. These prefixes must appear on the same line of code as the instruction they modify. For instance, in a 16-bit *.code16* section, you might write an instruction like the following example’s usage.

```
addr32 jmp1 *(%ebx)
```
- The *bus lock prefix*, *lock*, inhibits interrupts during execution of the instruction it precedes. (This is only valid with certain instructions; see a 80386 manual for details).
- The *wait for coprocessor prefix*, *wait*, waits for the coprocessor to complete the current instruction. This should never be needed for the 80386/80387

combination.

- The `rep`, `repe`, and `repne` prefixes are added to string instructions to make them repeat `%ecx` times.

An Intel syntax indirect memory reference of the following form translates into the AT&T syntax like the following declaration.

```
section:[base + index*scale + disp]
```

It then outputs like the following example.

```
section:disp(base, index, scale)
```

base and *index* are the optional 32-bit base and index registers, *disp* is the optional displacement, and *scale*, taking the values 1, 2, 4, and 8, multiplies *index* to calculate the address of the operand. If no *scale* is specified, *scale* is taken to be 1. *section* specifies the optional section register for the memory operand, and may override the default section register (see a 80386 manual for section register defaults). Section overrides in AT&T syntax *must* have be preceded by a `%`. If you specify a section override which coincides with the default section register, `as` does *not* output any section register override prefixes to assemble the given instruction. Thus, section overrides can be specified to emphasize which section register is used for a given memory operand. The following examples show Intel and AT&T style memory references:

- AT&T: `-4(%ebp)`; Intel: `[ebp - 4]`
base is `%ebp`; *disp* is `-4`; *section* is missing, and the default section is used (`%ss` for addressing with `%ebp` as the base register). *index*, *scale* are both missing.
- AT&T: `foo(,%eax,4)`; Intel: `[foo + eax*4]`
index is `%eax` (scaled by a *scale* 4); *disp* is `foo`; all other fields are missing. `%ds` is the section register by default.
- AT&T: `foo(,1)`; Intel: `[foo]`
This uses the value pointed to by `foo` as a memory operand. Note that *base* and *index* are both missing, but there is only *one* period (`,`), as a syntactic exception.
- AT&T: `%gs:foo`; Intel: `gs:foo`
This selects the contents of the `foo` variable with section register (*section*) being `%gs`.

Absolute (as opposed to PC relative) call and jump operands must be prefixed with `*`. If no `*` is specified, `as` always chooses PC relative addressing for jump/call labels. Any instruction that has a memory operand *must* specify its size (byte, word, or long) with an opcode suffix (`b`, `w`, or `l`, respectively).

Jump instructions are always optimized to use the smallest possible displacements. This is accomplished by using byte (8-bit) displacement jumps whenever the target is sufficiently close. If a byte displacement is insufficient a long (32-bit) displacement is used. We do not support word (16-bit) displacement jumps (i.e., prefixing the jump

instruction with the `addr16` opcode prefix), since the 80386 insists upon masking `%eip` to 16 bits after the word displacement is added.

IMPORTANT! `jcxz`, `jecxz`, `loop`, `loopz`, `loope`, `loopnz`, and `loopne` instructions only come in byte displacements, so that if you use these instructions (GCC does not use them) you may get an error message (and incorrect code). The AT&T 80386 assembler tries to get around this problem by expanding `jcxz foo` to the following example's form.

```

                                jcxz  cx_zero
                                jmp   cx_nonzero
cx_zero:                        jmp   foo
cx_nonzero:
```

All 80387 floating point types except packed BCD are supported. BCD support may be added without much difficulty. These data types are 16-, 32-, and 64-bit integers, and single (32-bit), double (64-bit), and extended (80-bit) precision floating point. Each supported type has an opcode suffix and a constructor associated with it. Opcode suffixes specify operand's data types. Constructors build these data types into memory.

- *Floating point constructors:*

`.float` or `.single`, `.double`, and `.tfloat` for 32-, 64-, and 80-bit formats. These correspond to opcode suffixes, `s`, `l`, and `t`. `t` stands for *temporary real*, and that the 80387 only supports this format via the `fldt` (*load temporary real to stack top*) and `fstpt` (*store temporary real and pop stack*) instructions.

- *Integer constructors:*

`.word`, `.long` or `.int`, and `.quad` for the 16-, 32-, and 64-bit integer formats. The corresponding *opcode suffixes* are `s` (single), `l` (long), and `q` (quad). As with the *temporary real* format, the 64-bit `q` format is only present in the `fldq` (*load quad integer to stack top*) and `fistpq` (*store quad integer and pop stack*) instructions.

Register to register operations do not require opcode suffixes, so that `fst %st, %st(1)`, is equivalent to `fstl %st, %st(1)`.

Since the 80387 automatically synchronizes with the 80386, `fwait` instructions are almost never needed (this is not the case for the 80286/80287 and 8086/8087 combinations). Therefore, `as` suppresses the `fwait` instruction whenever it is implicitly selected by one of the `fn` the instructions. For example, `fsave` and `fnsave` are treated identically. In general, all the `fn` the instructions are made equivalent to `f` instructions. If `fwait` is desired, it must be explicitly coded.

`as` supports Intel's MMX instruction set, the Single Instruction Multiple Data (SIMD) instructions for integer data, available on Intel's Pentium MMX processors and Pentium II processors, AMD's K6 and K6-2 processors, Cyrix's M2 processor, and others.

`as` also supports AMD's 3DNow! instruction set, SIMD instructions for 32-bit floating point data, available on AMD's K6-2 processor and possibly others in the future. Currently, `as` does not support Intel's floating point SIMD, Katmai (KNI). The eight 64-bit MMX operands, also used by 3DNow!, are called `%mm0`, `%mm1`, up to `%mm7`. They contain eight 8-bit integers, four 16-bit integers, two 32-bit integers, one 64-bit integer, or two 32-bit floating point values.

The MMX registers cannot be used at the same time as the floating point stack.

See Intel and AMD documentation, keeping in mind that the operand order in instructions is reversed from the Intel syntax.

While `as` normally writes only *pure* 32-bit i386 code, it has limited support for writing code to run in real mode or in 16-bit protected mode code segments; to do this, insert a `.code16` directive before the assembly language instructions to be run in 16-bit mode. You can switch `as` back to writing normal 32-bit code with the `.code32` directive. The code which `as` generates in 16-bit mode will not necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you must refrain from using any 32-bit constructs which require `as` to output address or operand size prefixes.

Writing 16-bit code instructions by explicitly specifying a prefix or an instruction mnemonic suffix within a 32-bit code section generates different machine instructions than those generated for a 16-bit code segment. In a 32-bit code section, the following code generates the machine opcode bytes `66 6a 04`, which pushes the value '4' onto the stack, decrementing `%esp` by 2.

```
pushw $4
```

The same code in a 16-bit code section would generate the machine opcode bytes `6a 04` (without the operand size prefix), which is correct since the processor default operand size is assumed to be 16 bits in a 16-bit code section.

The UnixWare assembler, and probably other AT&T derived i386 Unix assemblers, generate floating point instructions with reversed source and destination registers in certain cases. Unfortunately, GCC and possibly many other programs use this reversed syntax, so we're stuck with it. The following syntax serves as an example.

```
fsub %st,%st(3)
```

Such input results in `%st(3)` being updated to `%st - %st(3)` rather than the expected `%st(3) - %st`. This happens with all the non-commutative arithmetic floating point operations with two register operands where the source register is `%st` and the destination register is `%st(i)`.

There is some trickery concerning the `mul` and `imul` instructions that deserves mention. The 16-, 32-, and 64-bit expanding multiplies (base opcode `0xf6`; extension 4 for `mul` and 5 for `imul`) can be output only in the one operand form. Thus, `imul %ebx, %eax` does *not* select the expanding multiply; the expanding multiply would clobber the `%edx` register, and this would confuse gcc output. Use `imul %ebx` to get the 64-bit product in `%edx:%eax`. There is a two operand form of `imul` when the first

operand is an immediate mode expression and the second operand is a register. This is just a shorthand, so that, multiplying `%eax` by 69, for example, can be done with `imul $69, %eax` rather than `imul $69, %eax, %eax`.

The x86 has the following machine dependent options.

- q Disable some warning messages.
- v Display the assembler version number.
- Q
- K
- S Ignored; accepted for compatibility with other assemblers.

14

Hitachi H8/300 Dependent Features

The following documentation discusses the features pertinent to the Hitachi H8/300 processor regarding the GNU assembler.

`as` has no additional command line options for the Hitachi H8/300 family.

`;` is the line comment character. `$` can be used instead of a newline to separate statements. Therefore, *you may not use \$ in symbol names* on the H8/300.

You can use predefined symbols of the form `rnh` and `rn1` to refer to the H8/300 registers `as` sixteen 8-bit general-purpose registers. `n` is a digit from 0 to 7; for instance, both `r0h` and `r71` are valid register names.

You can also use the eight predefined symbols `rn` to refer to the H8/300 registers `as` 16-bit registers (you must use this form for addressing).

On the H8/300H, you can also use the eight predefined symbols `ern` (`er0 . . . er7`) to refer to the 32-bit general purpose registers. The two control registers are called `pc` (program counter; a 16-bit register, except on the H8/300H where it is 24 bits) and `ccr` (condition code register; an 8-bit register). `r7` is used as the stack pointer, and can also be called `sp`. `as` understands the following addressing modes for the H8/300.

`rn`
Register direct

`@rn`
Register indirect

@(*d*, *rn*)

@(*d*:16, *rn*)

@(*d*:24, *rn*)

Register indirect: 16-bit or 24-bit displacement, *d*, from register, *n*. (24-bit displacements are only meaningful on the H8/300H.)

@*rn*+

Register indirect with post-increment.

@-*rn*

Register indirect with pre-decrement.

@*aa*

@*aa*:8

@*aa*:16

@*aa*:24

Absolute address *aa*. (The address size :24 only makes sense on the H8/300H.)

#*xx*

#*xx*:8

#*xx*:16

#*xx*:32

Immediate data. You may specify the :8, :16, or :32 for clarity, if you wish; but `as` neither requires this nor uses it, since required data size is taken from context.

@@*aa*

@@*aa*:8

Memory indirect. You may specify the :8 for clarity, if you wish; but `as` neither requires this nor uses it.

For detailed information on the H8/300 machine instruction set, see *H8/300 Series Programming Manual* (Hitachi ADE-602-025). For information specific to the H8/300H, see *H8/300H Series Programming Manual* (Hitachi). `as` implements all the standard H8/300 opcodes. No additional pseudo-instructions are needed on this family.

The H8/300 family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools. `as` has only the following machine-dependent directive for the H8/300.

`.h8300h`

Recognize and emit additional instructions for the H8/300H variant, and also make `.int` emit 32-bit numbers rather than the usual (16-bit) for the H8/300 family.

`.sbranch`

`.lbranch`

Select the default length for branches to be either 8 bits (with `.sbranch`) or 16 bits (with `.lbranch`).

On the H8/300 family (including the H8/300H) `.word` directives generate 16-bit numbers.

15

Hitachi H8/500 Dependent Features

The following documentation discusses the features pertinent to the H8/500 processor regarding the GNU assembler.

`as` has no additional command line options for the Hitachi H8/500 family.

`!` is the line comment character.

`;` can be used instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

You can use the predefined symbols `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, and `r7` to refer to the H8/500 registers.

The H8/500 also has the following control registers.

<code>cp</code>	code pointer
<code>dp</code>	data pointer
<code>bp</code>	base pointer
<code>tp</code>	stack top pointer
<code>ep</code>	extra pointer
<code>sr</code>	status register
<code>ccr</code>	condition code register

All registers are 16 bits long. To represent 32 bit numbers, use two adjacent registers; for distant memory addresses, use one of the segment pointers (`cp` for the program counter; `dp` for `r0-r3`; `ep` for `r4` and `r5`; and `tp` for `r6` and `r7`).

`as` understands the following addressing modes for the H8/500:

`Rn`

Register direct.

`@Rn`

Register indirect.

`@(d:8, Rn)`

Register indirect with 8 bit signed displacement.

`@(d:16, Rn)`

Register indirect with 16 bit signed displacement.

`@-Rn`

Register indirect with pre-decrement.

`@Rn+`

Register indirect with post-increment.

`@aa:8`

8 bit absolute address.

`@aa:16`

16 bit absolute address.

`#xx:8`

8 bit immediate.

`#xx:16`

16 bit immediate.

The H8/500 family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools.

`as` has no machine-dependent directives for the H8/500. However, on this platform the `.int` and `.word` directives generate 16-bit numbers.

For detailed information on the H8/500 machine instruction set, see ***H8/500 Series Programming Manual*** (Hitachi M21T001). `as` implements all the standard H8/500 opcodes. No additional pseudo-instructions are needed on this family.

16

Hitachi SH Dependent Features

The following documentation discusses the features pertinent to the Hitachi SH processor regarding the GNU assembler.

`as` has no additional command line options for the Hitachi SH family.

`!` is the line comment character.

You can use `;` instead of a newline to separate statements.

Since `$` has no special meaning, you may use it in symbol names.

You can use the `r0`, `r1`, `r2`, `r3`, `r4`, `r5`, `r6`, `r7`, `r8`, `r9`, `r10`, `r11`, `r12`, `r13`, `r14`, and `r15` predefined symbols to refer to the SH registers.

The SH also has the following control registers.

```
pr
    procedure register (holds return address)
pc
    program counter
mach
macl
    high and low multiply accumulator registers
sr
    status register
gbr
```

`global`
base register

`vbr`
vector base register (for interrupt vectors)

`as` understands the following addressing modes for the SH. `Rn` in the following refers to any of the numbered registers, but not the control registers.

`Rn`
Register direct.

`@Rn`
Register indirect.

`@-Rn`
Register indirect with pre-decrement.

`@Rn+`
Register indirect with post-increment.

`@(disp, Rn)`
Register indirect with displacement.

`@(R0, Rn)`
Register indexed.

`@(disp, GBR)`
GBR offset.

`@(R0, GBR)`
GBR indexed.

`addr`

`@(disp, PC)`
PC relative address (for branch or for addressing memory). The `as` implementation allows you to use the simpler form `addr` anywhere a PC relative address is called for; the alternate form is supported for compatibility with other assemblers.

`#imm`
Immediate data.

The SH family has no hardware floating point, but the `.float` directive generates IEEE floating-point numbers for compatibility with other development tools.

`as` has the following machine-dependent options for the SH.

`-little`
Flag the output as little endian.

`-relax`
Alter jump instructions for long displacements.

`-small`
Align sections to a 4 byte boundary, not 16.

`-dsp`
Accept the SH DSP instructions, and disallow SH3e and SH4 instructions.

For detailed information on the SH machine instruction set, see *SH-Microcomputer User's Manual* (published by Hitachi Micro Systems, Inc.).

`as` implements all the standard SH opcodes. No additional pseudo-instructions are needed on this family. Note, however, that because `as` supports a simpler form of PC-relative addressing, you may simply write the following (for example).

```
mov.l bar,r0
```

Other assemblers might require an explicit displacement to `bar` from the program counter:

```
mov.l @(disp, PC)
```

`as` has the following machine-dependent directive for the SH.

```
.vaword expr
```

Like the `.word` directive, except that the assembler will not force the value to be word aligned.

HPPA Dependent Features

The following documentation discusses the features pertinent to the HPPA processor regarding the GNU assembler.

As a back end for GCC, `as` has been thoroughly tested and should work extremely well for the HPPA targets. It has been tested only minimally on hand-written assembly code and no one has tested it much on the assembly output from the HP compilers.

The format of the debugging sections has changed since the original `as` port (version 1.3x) was released; therefore, you must rebuild all HPPA objects and libraries with the new assembler so that you can debug the final executable.

The HPPA `as` port generates a small subset of the relocations available in the SOM and ELF object file formats. Additional relocation support will be added as it becomes necessary.

`as` has the following machine-dependent command line option for the HPPA.

`-C`

Generate a warning if a comment is found.

`-v`

Display the assembler's version number.

The assembler syntax closely follows the HPPA instruction set reference manual; assembler directives and general syntax closely follow the HPPA assembly language reference manual, with a few noteworthy differences. First, a colon may immediately follow a label definition. This is simply for compatibility with how most assembly

language programmers write code. Some obscure expression parsing problems may affect hand written code which uses the `spop` instructions, or code which makes significant use of the `!` line separator. `as` is much less forgiving about missing arguments and other similar oversights than the HP assembler. `as` notifies you of missing arguments as syntax errors; this is regarded as a feature, not a bug. Finally, `as` allows you to use an external symbol without explicitly importing the symbol; In the future this allowance will be an error for HPPA targets.

Special characters for HPPA targets include the following.

`;` is the line comment character.

`!` can be used instead of a newline to separate statements.

Since `§` has no special meaning, you may use it in symbol names.

The HPPA family uses IEEE floating-point numbers.

`as` for the HPPA supports many additional directives for compatibility with the native assembler. The following documentation only briefly describes them. For detailed information on HPPA-specific assembler directives, see *HP9000 Series 800 Assembly Language Reference Manual* (HP 92432-90001).

`as` does *not* support the following assembler directives described in the HP manual:

```
.endm      listoff      macro
.enter     liston
.leave     locct
```

Beyond those implemented for compatibility, `as` supports one additional assembler directive for the HPPA: `.param`. It conveys register argument locations for static functions. Its syntax closely follows the `.export` directive.

The following are the additional directives in `as` for the HPPA:

```
.begin_brtab
    Mark the start of a branch table; SOM format only.

.block n
.blockz n
    Reserve n bytes of storage, and initialize them to zero.

.call
    Mark the beginning of a procedure call. Only the special case with no arguments
    is allowed.

.callinfo [param=value, ...][flag, ...]
    Specify a number of parameters and flags that define the environment for a
    procedure. param may be any of frame (frame size), entry_gr (end of general
    register range), entry_fr (end of float register range), entry_sr (end of space
    register range). The values for flag are calls or caller (proc has subroutines),
no_calls (proc does not call subroutines), save_rp (preserve return pointer),
save_sp (proc preserves stack pointer), no_unwind (do not unwind this proc),
hpux_int (proc is interrupt routine).
```

-
- `.code`
Assemble into the standard section called `$TEXT$`, subsection `$CODE$`.
 - `.compiler sourcefile language version`
Create a compilation unit auxiliary header; SOM format only.
 - `.copyright "string"`
In the SOM object format, insert `string` into the object code, marked as a copyright string.
 - `.end_brtab`
Mark the end of a branch table; SOM format only.
 - `.enter`
Not yet supported; the assembler rejects programs containing this directive.
 - `.entry`
Mark the beginning of a procedure.
 - `.exit`
Mark the end of a procedure.
 - `.export name[,typ][,param=r]`
Make a procedure `name` available to callers. `typ`, if present, must be one of `absolute`, `code` (ELF only, not SOM), `data`, `entry`, `data`, `entry`, `millicode`, `plabel`, `pri_prog`, or `sec_prog`.
`param`, if present, provides either relocation information for the procedure arguments and result, or a privilege level. `param` may be `argw n` (where `n` ranges from 0 to 3, and indicates one of four one-word arguments); `rtnval` (the procedure's result); or `priv_lev` (privilege level). For arguments or the result, `r` specifies how to relocate, and must be one of `no` (not relocatable), `gr` (argument is in general register), `fr` (in floating point register), or `'fu'` (upper half of float register). For `priv_lev`, `r` is an integer.
 - `.half n`
Define a two-byte integer constant `n`; synonym for the portable `as` directive, `.short`.
 - `.import name[,typ]`
Converse of `.export`; make a procedure available to call. The arguments use the same conventions as the first two arguments for `.export`.
 - `.label name`
Define `name` as a label for the current assembly location.
 - `.leave`
Not yet supported; the assembler rejects programs containing this directive.
 - `.origin lc`
Advance location counter to `lc`. Synonym for the {No value for `'as'`} portable directive `.org`.
 - `.param name[,typ][,param=r]`
Similar to `.export`, but used for static procedures.

- `.proc`
Use preceding the first statement of a procedure.
- `.procend`
Use following the last statement of a procedure.
- `label.reg expr`
Synonym for `.equ`; define *label* with the absolute expression *expr* as its value.
- `.space secname[, params]`
Switch to section *secname*, creating a new section by that name if necessary. You may only use *params* when creating a new section, not when switching to an existing one. *secname* may identify a section by number rather than by name. If specified, the list *params* declares attributes of the section, identified by keywords. The keywords recognized are `spnum=exp` (identify this section by the number *exp*, an absolute expression), `sort=exp` (order sections according to this sort key when linking; *exp* is an absolute ex-pression), `unloadable` (section contains no loadable data), `notdefined` (this section defined elsewhere), and `private` (data in this section not available to other programs).
- `.spnum secnam`
Allocate four bytes of storage, and initialize them with the section number of the section named *secnam*. (You can define the section number with the HPPA `.space` directive.)
- `.string "str"`
Copy the characters in the string *str* to the object file. See “Strings” on page 27 for information on escape sequences you can use in `as` strings.

WARNING! The HPPA version of `.string` differs from the usual `as` definition: it does *not* write a zero byte after copying *str*.

- `.stringz "str"`
Like `.string`, but appends a zero byte after copying *str* to object file.
- `.subspa name[, params]`
`.nsubspa name[, params]`
Similar to `.space`, but selects a subsection name within the current section. You may only specify *params* when you create a subsection (in the first instance of `.subspa` for this *name*). If specified, the list *params* declares attributes of the subsection, identified by keywords. The keywords recognized are `quad=expr` (“quadrant” for this subsection), `align=expr` (alignment for beginning of this subsection; a power of two), `access=expr` (value for “access rights” field), `sort=expr` (sorting order for this subspace in link), `code_only` (subsection contains only code), `unloadable` (subsection cannot be loaded into memory), `common` (subsection is common block), `dup_comm` (initialized data may have duplicate names), or `zero` (subsection is all zeros, do not write in object file).
`.nsubspa` always creates a new subspace with the given name, even if one with the same name already exists.

```
.version "str"
```

Write *str* as version identifier in object code.

For detailed information on the HPPA machine instruction set, see ***PA-RISC Architecture and Instruction Set Reference Manual*** (published by Hewlett-Packard [HP 09740- 90039]).

Intel StrongARM Dependent Features

For detailed information on the GNU assembler for the Intel StrongARM processor, see “ARM Dependent Features” on page 79; for general information on the Intel StrongARM machine instruction set, see the *ARM Architecture Reference Manual* and Intel’s *StrongARM Reference Manual*.

19

Intel 960 Dependent Features

The following documentation discusses the features pertinent to the Intel 960 processor regarding the GNU assembler.

The following options are for Intel 80960 processors regarding the GNU assembler.

`-ACA` | `-AKA` | `-AKB` | `-AKC` | `-AMC` | `-ASA` | `-ASB` | `-AHX` | `-AJX`

Select the 80960 architecture. Instructions or features not supported by the selected architecture cause fatal errors.

`-ACA` is equivalent to `-ACA_A`; `-AKC` is equivalent to `-AMC`.

Synonyms are provided for compatibility with other tools.

If you do not specify any of these options, `as` generates code for any instruction or feature that is supported by *some* version of the i960 (even if this means mixing architectures!).

In principle, `as` attempts to deduce the minimal sufficient processor type if none is specified; depending on the object code format, the processor type may be recorded in the object file. If it is critical that the `as` output match a specific architecture, specify that architecture explicitly.

`-b`

Add code to collect information about conditional branches taken, for later optimization using branch prediction bits. (The conditional branch instructions have branch prediction bits in the CA, CB, and CC architectures.) If `BR` represents

a conditional branch instruction, the following represents the code generated by the assembler when `-b` is specified.

```

                call      increment  routine
                word     0          # pre-counter
Label:         BR
                call    increment  routine
                .word   0          # post-counter

```

The counter following a branch records the number of times that branch was *not* taken; the difference between the two counters is the number of times the branch *was* taken. A table of every such `Label` is also generated, so that the external postprocessor `gbr960` (supplied by Intel) can locate all the counters. This table is always labeled `__BRANCH_TABLE__`; this is a local symbol to permit collecting statistics for many separate object files. The table is word aligned, and begins with a two-word header. The first word, initialized to 0, is used in maintaining linked lists of branch tables. The second word is a count of the number of entries in the table, which follow immediately: each is a word, pointing to one of the labels illustrated in the previous example.

```

*NEXT      COUNT: N      *BRLAB 1      ...      *BRLAB N
                __BRANCH_TABLE__ layout

```

The first word of the header is used to locate multiple branch tables, since each object file may contain one. Normally the links are maintained with a call to an initialization routine, placed at the beginning of each function in the file. The GNU C compiler generates these calls automatically when you give it a `-b` option. For further details, see the documentation of `gbr960`.

`-no-relax`

Normally, Compare-and-Branch instructions with targets that require displacements greater than 13 bits (or that have external targets) are replaced with the corresponding compare (or `chkbit`) and branch instructions. You can use the `-no-relax` option to specify that `as` should generate errors instead, if the target displacement is larger than 13 bits.

This option does not affect the Compare-and-Jump instructions; the code emitted for them is *always* adjusted when necessary (depending on displacement size), regardless of whether you use `-no-relax`.

`--link-relax`

Preserve individual alignment directives so that the linker can perform relaxation.

`as` generates IEEE floating-point numbers for the directives `.float`, `.double`, `.extended`, and `.single`.

The following documentation discusses the features pertinent to the i960 regarding the machine directives for the GNU assembler.

`.bss symbol, length, align`

Reserve `length` bytes in the bss section for a local `symbol`, aligned to the power

of two specified by *align*, *length* and *align* must be positive absolute expressions. This directive differs from `.lcomm` only in that it permits you to specify an alignment. See “.lcomm symbol, length Directive” on page 57.

`.extended flonums`

`.extended` expects zero or more flonums, separated by commas; for each flonum, `.extended` emits an IEEE extended-format (80-bit) floating-point number.

`.leafproc call-lab, bal-lab`

You can use the `.leafproc` directive in conjunction with the optimized `callj` instruction to enable faster calls of leaf procedures. If a procedure is known to call no other procedures, you may define an entry point that skips procedure prolog code (and that does not depend on system-supplied saved context), and declare it as the *bal-lab* using `.leafproc`. If the procedure also has an entry point that goes through the normal prolog, you can specify that entry point as *call-lab*.

A `.leafproc` declaration is meant for use in conjunction with the optimized `call` instruction `callj`; the directive records the data needed later to choose between converting the `callj` into a `bal` or a `call`.

call-lab is optional; if only one argument is present, or if the two arguments are identical, the single argument is assumed to be the *bal* entry point.

`.sysproc name, index`

The `.sysproc` directive defines a name for a system procedure. After you define it using `.sysproc`, you can use *name* to refer to the system procedure identified by *index* when calling procedures with the optimized `call` instruction `callj`.

Both arguments are required; *index* must be between 0 and 31 (inclusive).

All Intel 960 machine instructions are supported. Some opcodes are processed beyond simply emitting a single corresponding instruction: `callj`, and Compare-and-Branch or Compare-and- Jump instructions with target displacements larger than 13 bits.

You can write `callj` to have the assembler or the linker determine the most appropriate form of subroutine call: `call`, `bal`, or `calls`. If the assembly source contains enough information—a `.leafproc` or `.sysproc` directive defining the operand—then `as` translates the `callj`; if not, it simply emits the `callj`, leaving it for the linker to resolve.

The 960 architectures provide combined *Compare-and-Branch* instructions that permit you to store the branch target in the lower 13 bits of the instruction word itself. However, if you specify a branch target far enough away that its address won't fit in 13 bits, the assembler can either issue an error, or convert your Compare-and-Branch instruction into separate instructions to do the compare and the branch.

Whether `as` gives an error or expands the instruction depends on two choices you can make: whether you use the `-no-relax` option, and whether you use a *Compare and Branch* instruction or a *Compare and Jump* instruction. The *Jump* instructions are *always* expanded if necessary; the *Branch* instructions are expanded when necessary *unless* you specify `-no-relax`, in which case `as` gives an error instead. Table 5 shows

the Compare-and-Branch instructions, their Jump variants, and the instruction pairs into which they may expand.

Table 5: Compare-and-Branch instructions

<i>Compare and Branch</i>	<i>Compare and Jump</i>	<i>Expanded to</i>
bbc		chkbit; bno
bbs		chkbit; bo
cmpibe	cmpije	cmpi; be
cmpibg	cmpijg	cmpi; bg
cmpibge	cmpijge	cmpi; bge
cmpibl	cmpijl	cmpi; bl
cmpible	cmpijle	cmpi; ble
cmpibno	cmpijno	cmpi; bno
cmpibne	cmpijne	cmpi; bne
cmpibo	cmpijo	cmpi; bo
cmpobe	cmpoje	cmpo; be
cmpobg	cmpojg	cmpo; bg
cmpobge	cmpojge	cmpo; bge
cmpobl	cmpojl	cmpo; bl
cmpoble	cmpojle	cmpo; ble
cmpobne	cmpojne	cmpo; bne

Intel $x86$ and IA64 Dependent Features

The following documentation discusses the features pertinent to the Intel $x86$ and IA 64 processors regarding the GNU assembler. See also “AT&T and Intel $x86$ Dependent Features” on page 83 for issues relating to the AT&T issues with respect to Intel processors.

The Intel $x86$ configurations use the following GNU assembler options.

- v
Print the assembler version and exit.
- q
Suppress some warning messages.

The Intel IA64 configurations use the following GNU assembler options.

- Milp32 | -Milp64 | -Mlp64 | -Mp64
Select the data model (-Mlp64 is the default).
- mauto-pic
Label the output with the EF-IA-64-NOFUNDESCONS option (that is, no function description).
- mconstant-gp
Label the output with the EF-IA-64-CONS-GP option (that is, a fixed GP register).
- Mle | -Mbe
Select the big endian or the little endian output.

`-x | -xexplicit`

Enable dependency violation checking.

`-xauto`

Automatically remove dependency violation checker.

`-xdebug`

Display debug information from the dependency violation checker.

`as` supports Intel's MMX instruction set, the Single Instruction Multiple Data (SIMD) instructions for integer data, available on Intel's Pentium MMX processors and Pentium II processors, AMD's K6 and K6-2 processors, Cyrix's M2 processor, and others.

`as` also supports AMD's 3DNow! instruction set, SIMD instructions for 32-bit floating point data, available on AMD's K6-2 processor and possibly others in the future. Currently, `as` does not support Intel's floating point SIMD, Katmai (KNI). The eight 64-bit MMX operands, also used by 3DNow!, are called `%mm0`, `%mm1`, up to `%mm7`. They contain eight 8-bit integers, four 16-bit integers, two 32-bit integers, one 64-bit integer, or two 32-bit floating point values. The MMX registers cannot be used at the same time as the floating point stack. See Intel and AMD documentation, keeping in mind that the operand order in instructions is reversed from the Intel syntax.

While `as` normally writes only *pure* 32-bit i386 code, it has limited support for writing code to run in real mode or in 16-bit protected mode code segments; to do this, insert a `.code16` directive before the assembly language instructions to be run in 16-bit mode. You can switch `as` back to writing normal 32-bit code with the `.code32` directive. The code which `as` generates in 16-bit mode will not necessarily run on a 16-bit pre-80386 processor. To write code that runs on such a processor, you must refrain from using any 32-bit constructs which require `as` to output address or operand size prefixes.

Writing 16-bit code instructions by explicitly specifying a prefix or an instruction mnemonic suffix within a 32-bit code section generates different machine instructions than those generated for a 16-bit code segment. In a 32-bit code section, the following code generates the machine opcode bytes `66 6a 04`, which pushes the value '4' onto the stack, decrementing `%esp` by 2.

```
pushw $4
```

The same code in a 16-bit code section would generate the machine opcode bytes `6a 04` (without the operand size prefix), which is correct since the processor default operand size is assumed to be 16 bits in a 16-bit code section.

The GNU assembler has the following machine-independent directive for the Intel IA64 processors.

```
.proc name [alternate]
```

Specify *name* as the entry point for a procedure, creating it if it does not already exist. An option list of alternate symbol names can also be provided with *alternate*, and they will be marked as procedure entry points.

MIPS Dependent Features

The following documentation discusses the features pertinent to the MIPS processors regarding the GNU assembler.

MIPS processors that have support are the MIPS R2000, R3000, R4000 and R6000. For information about the MIPS instruction set, see *MIPS RISC Architecture*, by Kane and Heindrich (Prentice-Hall). For an overview of MIPS assembly conventions, see “Appendix D: Assembly Language Programming” in the same work.

The MIPS configurations of GNU `as` support the following special options.

`-nocpp`

Ignored. It is accepted for compatibility with the native tools.

`-EL`

Generate little endian format output.

`-EB`

Generate big endian format output.

`-G num`

Set the largest size of an object that can be referenced implicitly with the `gp` register. Only accepted for targets that use ECOFF format, such as a DECstation running Ultrix, the default value for `num` is 8.

`-mcpu=cpu type`

Generate code for a particular MIPS `cpu type` processor. This has little effect on the assembler, but it is passed by GCC.

`-mips1`

`-mips2`

`-mips3`

`-mips4`

Generate code for a particular MIPS Instruction Set Architecture level. `-mips1` corresponds to the R2000 and R3000 processors, `-mips2` to the R6000 processor, and `-mips3` to the R4000 processor, and `-mips4` to the r8000 and r10000 processors. You can also switch instruction sets during the assembly; see details on page 116.

`-m4650`

`-no-m4650`

Generate code for the MIPS r4650 chip. This tells the assembler to accept the `mad` and `madu` instruction, and to not schedule `nop` instructions around accesses to the `HI` and `LO` registers. `-no-m4650` turns off this option.

`-m5400`

`-mno-m5400`

With `-m5400`, generate code for the MIPS 5400 port. With `-mno-m5400`, do not generate code for the MIPS 5400 port.

`-m3900`

`-mno-m3900`

With `-m3900`, generate code for the MIPS 3900 port. With `-mno-m3900`, do not generate code for the MIPS 3900 port.

`-m4010`

`-no-m4010`

Generate code for the LSI R4010 chip. This tells the assembler to accept the R4010 specific instructions (`addciu`, `ffc`, etc.), and to not schedule `nop` instructions around accesses to the `HI` and `LO` registers. `-no-m4010` turns off this option.

`-32`

Generate output in the 32-bit MIPS ELF format (default).

`-64`

Generate output in the 64-bit MIPS ELF format.

`-mips16`

`-no-mips16`

Generate code for the MIPS 16 processor. This is equivalent to putting `.set mips16` at the start of the assembly file. `-no-mips16` turns off this option.

`-xgot`

Use 32-bit offsets when accessing the GOT () in SVR4 PIC mode. For Irix compatibility.

-
- `--trap`
 - `--no-trap`
 - `--break`
 - `--no-break`
 - Control how to deal with multiplication overflow and division by zero. `--trap` or `--no-break` (which are synonyms) take a trap exception (and only work for Instruction Set Architecture level 2 and higher); `--break` or `--no-trap` (also synonyms, and the default) take a break exception.
 - `--emulation=name`
 - This option causes `as` to emulate as configured for some other target, in all respects, including output format (choosing between ELF and ECOFF only), handling of pseudo-opcodes which may generate debugging information or store symbol table information, and default endianness. The available configuration names are `mipsecoff`, `mipself`, `mipslecoff`, `mipsbecoff`, `mipslelf`, `mipsbelf`. The first two do not alter the default endianness from that of the primary target for which the assembler was configured; the others change the default to little- or big-endian as indicated by the `b` or `l` in the name. Using `-EB` or `-EL` will override the endianness selection in any case. This option is currently supported only when the primary target for which `as` is configured is a MIPS ELF or ECOFF target. Furthermore, the primary target or others specified with `--enable-targets=name` (where `name` is the target) at configuration time must include support for the other format, if both are to be available. For example, the Irix 5 configuration includes support for both. Eventually, this option will support more configurations, with more fine-grained control over the assembler's behavior, and will be supported for more processors.
 - `-membedded-pic`
 - Generate code suitable for use in an embedded PIC environment.
 - `-mabi number`
 - Set the MIPS ABI in use to `number` (one of `32`, `n32`, `64`, `o64` or `eabi`).
 - `-mfix7000`
 - Enable a workaround for a hardware bug in the MIPS 7000; insert a NOPs around `mfhi` and `mflo` instructions when the register is referenced in the next two instructions.
 - `-KPIC`
 - `-call-shared`
 - Generate position independent code.
 - `-mpg32`
 - Assume a 32-bit global pointer.
 - `-mpg64`
 - Assume a 64-bit global pointer (default).
 - `--no-construct-float`
 - `--construct-float`
 - Disable the construction of double width floating point constants by assembling

two single width floating point constants into a pair of single width floating point registers that alias the destination double width floating point registers.

-g

Do not perform optimizations which limit full symbolic debugging.

-g2

Do not perform optimizations which limit full symbolic debugging like -g, but allow removal of unneeded NOPs.

-O0

Remove unneeded NOPs.

-O

Remove unneeded NOPs like -O0, but also allow branches to be swapped.

Assembling for a MIPS ECOFF target supports some additional sections besides the usual `.text`, `.data` and `.bss`. The additional sections are `.rdata`, used for read-only data, `.sdata`, used for small data, and `.sbss`, used for small common objects.

When assembling for ECOFF, the assembler uses the `$gp` (`$28`) register to form the address of a *small object*. Any object in the `.sdata` or `.sbss` sections is considered “small” in this sense. For external objects, or for objects in the `.bss` section, you can use the `gcc -G` command to control the size of objects addressed via `$gp`; the default value is 8, meaning that a reference to any object eight bytes or smaller uses `$gp`.

Passing `-G 0` to `as` prevents it from using the `$gp` register on the basis of object size (but the assembler uses `$gp` for objects in `.sdata` or `sbss` in any case). The size of an object in the `.bss` section is set by the `.comm` or `.lcomm` directive that defines it. The size of an external object may be set with the `.extern` directive. For example, `.extern sym,4` declares that the object at `sym` is 4 bytes in length, while leaving `sym` otherwise undefined.

Using small ECOFF objects requires linker support, and assumes that the `$gp` register is correctly initialized (normally done automatically by the startup code). MIPS ECOFF assembly code must not modify the `$gp` register.

MIPS ECOFF `as` supports several directives used for generating debugging information which are not support by traditional MIPS assemblers. These are `.def`, `.endif`, `.dim`, `.file`, `.scl`, `.size`, `.tag`, `.type`, `.val`, `.stabd`, `.stabn`, and `.stabs`. The debugging information generated by the three `.stab` directives can only be read by GDB, not by traditional MIPS debuggers (this enhancement is required to fully support C++ debugging). These directives are primarily used by compilers, not assembly language programmers!

GNU `as` supports an additional directive to change the MIPS Instruction Set Architecture level on the fly: `.set mipsn`. `n` should be a number from 0 to 3. A value from 1 to 3 makes the assembler accept instructions for the corresponding isa level, from that point on in the assembly. `.set mipsn` affects not only which instructions are permitted, but also how certain macros are expanded. `.set mips0` restores the ISA

level to its original level: either the level you selected with command line options, or the default for your configuration. You can use this feature to permit specific R4000 instructions while assembling in 32-bit mode. Use this directive with care!

The `.set mips16` directive puts the assembler into MIPS 16 mode, in which it will assemble instructions for the MIPS 16 processor. Use `.set nomips16` to return to normal 32 bit mode. Traditional MIPS assemblers do not support this directive.

By default, MIPS 16 instructions are automatically extended to 32 bits when necessary. The directive `.set noautoextend` will turn this off. When `.set noautoextend` is in effect, any 32 bit instruction must be explicitly extended with the `'e'` modifier (e.g., `'li.e $4,1000'`). The directive `.set autoextend` may be used to once again automatically extend instructions when necessary.

This directive is only meaningful when in MIPS 16 mode. Traditional MIPS assemblers do not support this directive.

The `.insn` directive tells `as` that the following data is actually instructions. This makes a difference in MIPS 16 mode: when loading the address of a label which precedes instructions, `as` automatically adds 1 to the value, so that jumping to the loaded address will do the right thing.

The directives, `.set push` and `.set pop`, may be used to save and restore the current settings for all the options which are controlled by `.set`. The `.set push` directive saves the current settings on a stack. The `.set pop` directive pops the stack and restores the settings.

These directives can be useful inside an macro which must change an option such as the ISA level or instruction reordering but does not want to change the state of the code which invoked the macro.

Traditional MIPS assemblers do not support these directives.

Mitsubishi D10V Dependent Features

The following documentation discusses the features pertinent to the Mitsubishi D10V series of processors regarding the GNU assembler.

The following options are for Mitsubishi D10V processors.

-O

The D10V can often execute two sub-instructions in parallel. When this option is used, `as` will attempt to optimize its output by detecting when instructions can be executed in parallel.

--nowarnswap

Suppresses warnings about swapping the order of instructions. To optimize execution performance, `as` will sometimes swap the order of instructions; normally this swapping generates a warning.

The D10V version of `as` uses the instruction names in the *D10V Architecture Manual*. However, the names in the manual are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. `as` will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, `as` will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either `.s` (short) or `.l` (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol defined later in your program, you can write `bra.s foo` as an instruction. `objdump` and GDB will always append `.s` or

.1 to instructions having both short and long forms.

The D10V assembler takes as input a series of instructions, either one-per-line, or in the special two-per-line format described next section. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one.

Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary.

If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols.

`;` and `#` are the line comment characters.

Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially. To specify the executing order, use the following symbols.

`->`

Sequential with instruction on the left first.

`<-`

Sequential with instruction on the right first.

`||`

Parallel.

The D10V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line.

```
abs a1 -> abs r0
```

Execute these sequentially. The instruction on the right is in the right container and is executed second.

```
abs r0 <- abs a1
```

Execute these reverse-sequentially. The instruction on the right is in the right container, and is executed first.

```
ld2w r2,@r8+ || mac a0,r0,r7
```

Execute these in parallel.

```
ld2w r2,@r8+ ||  
mac a0,r0,r7
```

Two-line format. Execute these in parallel.

```
ld2w r2,@r8+  
mac a0,r0,r7
```

Two-line format. Execute these sequentially. Assembler will put them in the proper containers.

```
ld2w r2,@r8+ ->
mac a0,r0,r7
```

Two-line format. Execute these sequentially. Same as previous set of instructions, but second instruction will always go into right container.

Since `§` has no special meaning, you may use it in symbol names.

You can use the `r0` through `r15` predefined symbols to refer to the D10V registers. You can also use `sp` as an alias for `r15`. The accumulators are `a0` and `a1`. There are special register-pair names that may optionally be used in opcodes that require even-numbered registers. Register names are not case sensitive.

The D10V also has the following register pairings.

- `r0` *with* `r1`
- `r2` *with* `r3`
- `r4` *with* `r5`
- `r6` *with* `r7`
- `r8` *with* `r9`
- `r10` *with* `r11`
- `r12` *with* `r13`
- `r14` *with* `r15`

The D10V also has predefined symbols for the following control registers and status bits.

```
psw
    Processor Status Word
bpsw
    Backup Processor Status Word
pc
    Program Counter
bpc
    Backup Program Counter
rpt_c
    Repeat Count
rpt_s
    Repeat Start address
rpt_e
    Repeat End address
mod_s
    Modulo Start address
mod_e
    Modulo End address
```

`iba`
Instruction Break Address

`f0`
Flag 0

`f1`
Flag 1

`c`
Carry flag

`as` understands the following addressing modes for the D10V. `Rn` in the following descriptions refers to any of the numbered registers, but not the control registers.

`Rn`
Register direct

`@Rn`
Register indirect

`@Rn+`
Register indirect with post-increment

`@Rn-`
Register indirect with post-decrement

`@-SP`
Register indirect with pre-decrement

`@(disp,Rn)`
Register indirect with displacement

`addr`
PC relative address (for branch or rep)

`#imm`
Immediate data (the `#` is optional and ignored)

Any symbol followed by `@word` will be replaced by the symbol's value shifted right by 2. This is used in situations such as loading a register with the address of a function (or any other code fragment). For example, if you want to load a register with the location of the function `main` then jump to that function, you could do use the following example as input.

```
ldi r2, main@word
jmp r2
```

The D10V has no hardware floating point, but the `.float` and `.double` directives generates IEEE floating-point numbers for compatibility with other development tools.

For detailed information on the D10V machine instruction set, see *D10V Architecture: A VLIW Microprocessor for Multimedia Applications* (from Mitsubishi Electric Corp.). `as` implements all the standard D10V opcodes. The only changes are those described in the section on size modifiers.

Mitsubishi D30V Dependent Features

The following documentation discusses the features pertinent to the Mitsubishi D30V processors regarding the GNU assembler.

The following assembler options are for Mitsubishi D30V processors.

- n Warn when `nops` are generated.
- N Warn when a `nop` after a 32-bit multiply instruction is generated before a load or 16-bit multiply instruction.
- O The D30V can often execute two sub-instructions in parallel. When this option is used, `as` will attempt to optimize its output by detecting when instructions can be executed in parallel.
- c By default, generate warning messages if a symbol has the same name as a register.
- C Disable behavior of `-c`.

The D30V syntax is based on the syntax in Mitsubishi's D30V architecture manual. The differences are detailed in the following documentation.

The D30V version of `as` uses the instruction names in the *D30V Architecture*

Manual. However, the names in the manual are sometimes ambiguous. There are instruction names that can assemble to a short or long form opcode. `as` will always pick the smallest form if it can. When dealing with a symbol that is not defined yet when a line is being assembled, it will always use the long form. If you need to force the assembler to use either the short or long form of the instruction, you can append either `.s` (short) or `.l` (long) to it. For example, if you are writing an assembly program and you want to do a branch to a symbol that is defined later in your program, you can write `bra.s foo` to resolve the need. `objdump` and `GDB` will always append `.s` or `.l` to instructions having both short and long forms.

The D30V assembler takes `as` input a series of instructions, either one-per-line, or in the special two-per-line format. Some of these instructions will be short-form or sub-instructions. These sub-instructions can be packed into a single instruction. The assembler will do this automatically. It will also detect when it should not pack instructions. For example, when a label is defined, the next instruction will never be packaged with the previous one. Whenever a branch and link instruction is called, it will not be packaged with the next instruction so the return address will be valid. Nops are automatically inserted when necessary. If you do not want the assembler automatically making these decisions, you can control the packaging and execution type (parallel or sequential) with the special execution symbols.

`;` and `#` are the line comment characters. Sub-instructions may be executed in order, in reverse-order, or in parallel. Instructions listed in the standard one-per-line format will be executed sequentially unless you use the `-o` option.

To specify the executing order, use the following symbols.

```
-> Sequential with instruction on the left first
<- Sequential with instruction on the right first
|| Parallel
```

The D30V syntax allows either one instruction per line, one instruction per line with the execution symbol, or two instructions per line. Use the following examples.

```
abs r2,r3 -> abs r4,r5
Execute these sequentially. The instruction on the right is in the right container
and is executed second.
```

```
abs r2,r3 <- abs r4,r5
Execute these reverse-sequentially. The instruction on the right is in the right
container, and is executed first.
```

```
abs r2,r3 || abs r4,r5
Execute these in parallel.
```

```
ldw r2,@(r3,r4) ||
mulx r6,r8,r9
```

Two-line format. Execute these in parallel.

```
mulx a0,r8,r9
stw r2,@(r3,r4)
```

Two-line format. Execute these sequentially unless `-o` option is used. If the `-o` option is used, the assembler will determine if the instructions could be done in parallel (the above two instructions can be done in parallel), and if so, emit them as parallel instructions. The assembler will put them in the proper containers. In the above example, the assembler will put the `stw` instruction in left container and the `mulx` instruction in the right container.

```
stw r2,@(r3,r4) ->
mulx a0,r8,r9
```

Two-line format. Execute the `stw` instruction followed by the `mulx` instruction sequentially. The first instruction goes in the left container and the second instruction goes into right container. The assembler will give an error if the machine ordering constraints are violated.

```
stw r2,@(r3,r4) <-
mulx a0,r8,r9
```

Same as previous example, except that the `mulx` instruction is executed before the `stw` instruction.

Since `$` has no special meaning, you may use it in symbol names.

`as` supports the full range of guarded execution directives for each instruction. Just append the directive after the instruction proper. The following directives are for the Mitsubishi D30V processors.

```
/tx
```

Execute the instruction if flag `f0` is true.

```
/fx
```

Execute the instruction if flag `f0` is false.

```
/xt
```

Execute the instruction if flag `f1` is true.

```
/xf
```

Execute the instruction if flag `f1` is false.

```
/tt
```

Execute the instruction if both flags `f0` and `f1` are true.

```
/tf
```

Execute the instruction if flag `f0` is true and flag `f1` is false.

You can use the `r0` through `r63` predefined symbols to refer to the D30V registers. You can also use `sp` as an alias for `r63` and `link` as an alias for `r62`. The accumulators are `a0` and `a1`. The D30V also has predefined symbols for the following control registers and status bits.

<code>psw</code>	Processor Status Word
<code>bpsw</code>	Backup Processor Status Word
<code>pc</code>	Program Counter
<code>bpc</code>	Backup Program Counter
<code>rpt_c</code>	Repeat Count
<code>rpt_s</code>	Repeat Start address
<code>rpt_e</code>	Repeat End address
<code>mod_s</code>	Modulo Start address
<code>mod_e</code>	Modulo End address
<code>iba</code>	Instruction Break Address
<code>f0</code>	Flag 0
<code>f1</code>	Flag 1
<code>f2</code>	Flag 2
<code>f3</code>	Flag 3
<code>f4</code>	Flag 4
<code>f5</code>	Flag 5
<code>f6</code>	Flag 6
<code>f7</code>	Flag 7
<code>s</code>	Same as flag 4 (saturation flag)
<code>v</code>	Same as flag 5 (overflow flag)

`va`
Same as flag 6 (sticky overflow flag)

`c`
Same as flag 7 (carry/borrow flag)

`b`
Same as flag 7 (carry/borrow flag)

`as` understands the following addressing modes for the D30V. `Rn` in the following refers to any of the numbered registers, but not the control registers.

`Rn`
Register direct

`@Rn`
Register indirect

`@Rn+`
Register indirect with post-increment

`@Rn-`
Register indirect with post-decrement

`@-SP`
Register indirect with pre-decrement

`@(disp, Rn)`
Register indirect with displacement

`addr`
PC relative address (for branch or rep).

`#imm`
Immediate data (the # is optional and ignored)

The D30V has no hardware floating point, but the `.float` and `.double` directives generates IEEE floating-point numbers for compatibility with other development tools.

For detailed information on the D30V machine instruction set, see ***D30V Architecture: A VLIW Microprocessor for Multimedia Applications*** (from Mitsubishi Electric Corp.). `as` implements all the standard D30V opcodes. The only changes are those described in the section on size modifiers.

Mitsubishi M32R Dependent Features

The following documentation discusses the features pertinent to the Mitsubishi M32R processor regarding the GNU assembler.

The Mitsubishi M32R processors have a few machine-dependent `as` options

`-m32rx`

`as` can assemble code for several different members of the Mitsubishi M32R family. Normally the default is to assemble code for the M32R microprocessor. This option may be used to change the default to the M32RX microprocessor, which adds some more instructions to the basic M32R instruction set, and some additional parameters to some of the original instructions.

`-warn-explicit-parallel-conflicts`

Instructs `as` to produce warning messages when questionable parallel instructions are encountered. This option is enabled by default, but GCC disables it when it invokes `as` directly. Questionable instructions are those whose behaviour would be different if they were executed sequentially. For example, a `'mv r1, r2 || mv r3, r1'` code fragment produces a different result from `'mv r1, r2 \n mv r3, r1'` since the former moves `r1` into `r3` and *then* `r2` into `r1`, whereas the latter moves `r2` into `r1` *and* `r3`.

`-Wp`

This is a shorter synonym for the `-warn-explicit-parallel-conflicts` option.

`-no-warn-explicit-parallel-conflicts`

Instructs `as` not to produce warning messages when questionable parallel instructions are encountered.

`-Wnp`

This is a shorter synonym for the `-no-warn-explicit-parallel-conflicts` option.

`-Wuh`

`--warn-unmatched-high`

`-Wnuh`

`--no-warn-unmatched-high`

`-Wuh` and `--warn-unmatched-high` generate a warning message if a high or shigh relocation is seen without a matching low relocation. The default (with `-Wuh` or `--no-warn-unmatched-high`) is to avoid producing such warnings.

The Mitsubishi M32R processors have the following machine-dependent `as` directives.

`.m32rx`

Allow the assembly of `m32rx` instructions.

`.m32r`

Disallow the assembly of `m32rx` instructions.

`.scomm`

Like the `.comm` directive, except that the values are placed in the `.scommon` section.

There are several warning and error messages that can be produced by `as` which are specific to the M32R processors.

output of 1st instruction is the same as an input to 2nd instruction - is this intentional ?

This message is only produced if warnings for explicit parallel conflicts have been enabled. It indicates that the assembler has encountered a parallel instruction in which the destination register of the left hand instruction is used as an input register in the right hand instruction. For example, a `mv r1, r2 || neg r3, r1` code fragment produces register `r1` as the destination of the `move` instruction and the input to the `neg` instruction.

output of 2nd instruction is the same as an input to 1st instruction - is this intentional ?

This message is only produced if warnings for explicit parallel conflicts have been enabled. It indicates that the assembler has encountered a parallel instruction in which the destination register of the right hand instruction is used as an input register in the left hand instruction. For example, a `mv r1, r2 || neg r2, r3` code fragment produces register `r2` as the destination of the `neg` instruction and the input to the `move` instruction.

instruction `...' is for the M32RX only

This message is produced when the assembler encounters an instruction which is

only supported by the M32Rx processor, and the `-m32rx` command line flag has not been specified to allow assembly of such instructions.

unknown instruction `...`

This message is produced when the assembler encounters an instruction which it does not recognise.

only the NOP instruction can be issued in parallel on the m32r

This message is produced when the assembler encounters a parallel instruction which does not involve a NOP instruction and the `-m32rx` command line flag has not been specified. Only the M32RX processor is able to execute two instructions in parallel.

instruction `...` cannot be executed in parallel.

This message is produced when the assembler encounters a parallel instruction which is made up of one or two instructions which cannot be executed in parallel.

Instructions share the same execution pipeline

This message is produced when the assembler encounters a parallel instruction whose components both use the same execution pipeline.

Instructions write to the same destination register.

This message is produced when the assembler encounters a parallel instruction where both components attempt to modify the same register. For example, the following code fragments will produce such a message (both write to the condition bit):

```
□ mv r1, r2 || neg r1, r3
□ jl r0 || mv r14, r1
□ st r2, @-r1 || mv r1, r3
□ mv r1, r2 || ld r0, @r1+
□ cmp r1, r2 || addx r3, r4
```

Unmatched high/shigh reloc

This message means that a `high` or an `shigh` relocation was seen without a matching `low` relocation.

Motorola 68K Dependent Features

The following documentation discusses the features pertinent to the Motorola 68K series of processors regarding the GNU assembler.

The following options are for Motorola 68K processors.

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.

Normally, the pipe character (`|`) is treated as a comment character, which means that it can not be used in expressions. The `--bitwise-or` option turns `|` into a normal character. In this mode, you must either use C style comments, or start comments with a `#` character at the beginning of a line.

If you use an addressing mode with a base register without specifying the size, as will normally use the full 32 bit value. For example, the `%a0@(%d0)` addressing mode is equivalent to the `%a0@(%d0:1)` addressing mode. Use the `--base-size-default-16` option to tell as to default to using the 16 bit value. `%a0@(%d0)` is equivalent to

`%a0@(%d0:w)` in this case. You may use the `--base-size-default-32` option to restore the default behaviour.

If you use an addressing mode with a displacement, and the value of the displacement is not known, `as` will normally assume that the value is 32 bits. For example, if the `disp` symbol has not been defined, `as` will assemble the `%a0@(disp,%d0)` addressing mode as though `disp` is a 32 bit value. You may use the `--disp-size-default-16` option to tell `as` to instead assume that the displacement is 16 bits. In this case, `as` will assemble `%a0@(disp,%d0)` as though `disp` is a 16 bit value. You may use the `--disp-size-default-32` option to restore the default behaviour.

`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.

`-l`

Shorten references to undefined symbols, to one word instead of two. 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.

`-m68000`
`-m68ec000`
`-m68hc000`
`-m68hc001`
`-m68008`
`-m68302`
`-m68306`
`-m68307`
`-m68322`
`-m68356`

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`
`-m68ec020`

Assemble for the 68020. This is normally the default.

-m68030

-m68ec030

Assemble for the 68030.

-m68040

-m68ec040

Assemble for the 68040.

-m68060

-m68ec060

Assemble for the 68060.

-mcpu32

-m68330

-m68331

-m68332

-m68333

-m68334

-m68336

-m68340

-m68341

-m68349

-m68360

Assemble for the CPU32 family of chips.

-m5200

Assemble for the ColdFire 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.

The previous 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, `mov1` is equivalent to `mov.l`.

The following options are for Motorola 68HC11 or 68HC12 series processors.

`-m68hc11` | `-m68hc12`

Specify which processor is the target. The default is defined by the configuration option when building the assembler.

`--force-long-branches`

Relative branches are turned into absolute ones. This concerns conditional branches, unconditional branches and branches to a sub-routine.

`-S` | `--short-branches`

Do not turn relative branches into absolute ones when the offset is out of range.

`--strict-direct-mode`

Do not turn the direct addressing mode into extended addressing mode when the instruction does not support direct addressing mode.

`--print-insn-syntax`

Print the syntax of instruction in case of error.

`--print-opcodes`

Print the list of instructions with syntax and then exit.

`--generate-example`

Print an example of instruction for each possible instruction and then exit. This option is only useful for testing the assembler.

The following documentation discusses the M680x0 syntax. This syntax for the Motorola 680x0 was developed at MIT. The 680x0 version of `as` uses instruction names and syntax compatible with the Sun assembler. Intervening periods are ignored; for example, `mov.l` is equivalent to `mov.l`. `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` (that is, the stack pointer). `%a6` is also known as `%fp` (that is, 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`.

The standard Motorola syntax chip differs from the syntax discussed with “Syntax” on page 23. `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 discussion, *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` (i.e., 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.

Packed decimal (P) format floating literals are not supported.

The following directives generate floating point formats.

`.float`

single precision floating point constants.

`.double`

double precision floating point constants.

`.extend`

`.ldouble`

extended precision (long double) floating point constants.

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.

Certain pseudo opcodes are permitted for branch instructions. They expand to the shortest branch instruction that reach the target. Generally these mnemonics are made by substituting *j* for *b* at the start of a Motorola mnemonic. Example 1 summarizes the pseudo-operations.

Example 1: Motorola pseudo-operations

Pseudo-Op	Displacement				
	BYTE	WORD	LONG	LONG	non-PC relative
jbsr	bsrs	bsr	bsrl	jsr	jsr
jra	bras	bra	bral	jmp	jmp
* jXX	bXXs	bXX	bXXl	bNXs; jmp1	bNXs; jmp
* dbXX	dbXX	dbXX		dbXX; bra; jmp1	
* fjXX	fbXXw	fbXXw	fbXXl		fbNXw; jmp

XX: condition
NX: negative of XX condition

The following discussion describes the specific requirements for M68K branch instructions with respect to pseudo-ops (as shown with asterisks in Example 1).

jbsr
jra

These are the simplest jump pseudo-operations; they always map to one particular machine instruction, depending on the displacement to the branch target.

JXX

Here, jxx stands for an entire family of pseudo-operations, where xx is a conditional branch or condition-code test. The full list of pseudo-ops includes jcc, jcs, jeq, jge, jgt, jhi, jle, jls, jlt, jmi, jne, jpl, jvc, and jvs: For the cases of non-PC relative displacements and long displacements on the 68000 or 68010, as issues a longer code fragment in terms of NX, the opposite condition to xx, as the following example shows.

```
jXX foo
```

For the non-PC relative case in the previous example, as gives the following output.

```
bNXs oof
jmp foo
oof:
```

dbXX

The full family of pseudo-operations includes dbcc, dbcs, dbeq, dbf, dbge, dbgt, dbhi, dble, dbls, dblt, dbmi, dbne, dbpl, dbra, dbt, dbvc, and dbvs. Other than for word and byte displacements, when the source reads dbxx foo, as emits the following output.

```
dbXX oo1
bra oo2
oo1: jmp1 foo
oo2:
```

`fjXX`

This family includes `fjeq`, `fjf`, `fjge`, `fjgl`, `fjgle`, `fjgt`, `fjt`, `fjle`, `fjlt`, `fjne`, `fjnge`, `fjnjl`, `fjnle`, `fjnlt`, `fjoge`, `fjogl`, `fjogt`, `fjole`, `fjolt`, `fjor`, `fjseq`, `fjsf`, `fjsne`, `fjst`, `fjueq`, `fjuge`, `fjugt`, `fjule`, `fjult`, and `fjun`. For branch targets that are not PC relative, `as` emits the following output when it encounters `fjXX foo`.

```
fbNX oof
jmp foo
oof:
```

The immediate character is `#` for Sun compatibility. The line-comment character is `|`. If a `#` appears at the beginning of a line, it is treated as a comment unless it looks like `# line file`, in which case it is treated normally.

NEC V850 Dependent Features

The following documentation discusses the features pertinent to the NEC V850 processors regarding the GNU assembler.

The NEC V850 configurations of GNU `as` support the following special options.

`-mwarn-signed_overflow`

Causes warnings to be produced when signed immediate values overflow the space available for them within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

`-mwarn-unsigned_overflow`

Causes warnings to be produced when unsigned immediate values overflow the space available for them within their opcodes. By default this option is disabled as it is possible to receive spurious warnings due to using exact bit patterns as immediate constants.

`-mv850`

Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

`-mv850e`

Specifies that the assembled code should be marked as being targeted at the V850E processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

`-mv850any`

Specifies that the assembled code should be marked as being targeted at the V850 processor but support instructions that are specific to the extended variants of the process. This allows the production of binaries that contain target specific code, but which are also intended to be used in a generic fashion. For example, `libgcc.a` contains generic routines used by the code produced by GCC for all versions of the v850 architecture, together with support routines only used by the V850E architecture.

`#` is the line comment character.

`as` supports the following names for registers.

Table 6: General registers for the NEC V850

general register 0 r0, zero	general register 16 r16
general register 1 r1	general register 17 r17
general register 2 r2, hp	general register 18 r18
general register 3 r3, sp	general register 19 r19
general register 4 r4, gp	general register 20 r20
general register 5 r5, tp	general register 21 r21
general register 6 r6	general register 22 r22
general register 7 r7	general register 23 r23
general register 8 r8	general register 24 r24
general register 9 r9	general register 25 r25
general register 10 r10	general register 26 r26
general register 11 r11	general register 27 r27
general register 12 r12	general register 28 r28
general register 13 r13	general register 29 r29
general register 14 r14	general register 30 r30, ep
general register 15 r15	general register 31 r31, lp

Table 7: System registers for the NEC V850

```

system register 0
eipc
system register 1
eipsw
system register 2
fepc
system register 3
fepsw
system register 4
ecr
system register 5
psw
system register 16
ctpc
system register 17
ctpsw
system register 18
dbpc
system register 19
dbpsw
system register 20
ctbp

```

The V850 family uses IEEE floating-point numbers.

The following special V850 assembler directives are available.

`.offset expression`

Moves the offset into the current section to the specified amount (*expression*).

`.section "name", type`

This is an extension to the standard `.section` directive. It sets the current section (*type*) and creates an alias for this section ("name").

`.v850`

Specifies that the assembled code should be marked as being targeted at the V850 processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

`.v850e`

Specifies that the assembled code should be marked as being targeted at the V850E processor. This allows the linker to detect attempts to link such code with code assembled for other processors.

`.sdata`

Sets the current section to `.sdata`.

`.tdata`

Sets the current section to `.tdata`.

```
.zdata
    Sets the current section to .zdata.
.sbss
    Sets the current section to .sbss.
.tbss
    Sets the current section to .tbss.
.zbss
    Sets the current section to .zbss.
.rodata
    Sets the current section to .rodata.
.rozdata
    Sets the current section to .rozdata.
.scomm
    Sets the current section to .scomm.
.tcomm
    Sets the current section to .tcomm.
.zcomm
    Sets the current section to .zcomm.
.call_table_data
    Sets the current section to .call_table_data.
.call_table_text
    Sets the current section to .call_table_text.
```

as implements all the standard V850 opcodes as well as implementing the following pseudo ops.

hi0()

Computes the higher 16 bits of the given expression and stores it into the immediate operand field of the given instruction. Use the following example's usage.

```
`mulhi hi0(here - there), r5, r6'
```

This computes the difference between the address of labels *here* and *there*, takes the upper 16 bits of this difference, shifts it down 16 bits and then multiplies it by the lower 16 bits in register 5, putting the result into register 6.

lo()

Computes the lower 16 bits of the given expression and stores it into the immediate operand field of the given instruction. Use the following example's usage.

```
`addi lo(here - there), r5, r6'
```

This computes the difference between the address of labels *here* and *there*, takes the lower 16 bits of this difference and adds it to register 5, putting the result into

register 6.

`hi()`

Computes the higher 16 bits of the given expression and then adds the value of the most significant bit of the lower 16 bits of the expression and stores the result into the immediate operand field of the given instruction. For instance, the following code example shows how to compute the address of the `here` label, storing it into register 6.

```
'movhi hi(here), r0, r6' `movea lo(here), r6, r6'
```

The reason for this special behaviour is that `movea` performs a sign extension on its immediate operand. So, for example, if the address of `here` was `0xFFFFFFFF`, then without the special behaviour of the `hi()` pseudo-op the `movhi` instruction would put `0xFFFF0000` into `r6`, then the `movea` instruction would take its immediate operand, `0xFFFF`, sign extend it to 32 bits, `0xFFFFFFFF`, and then add it into `r6` giving `0xFFFFFFFFFF`, which is wrong (the fifth nibble is E). With the `hi()` pseudo op adding in the top bit of the `lo()` pseudo op, the `movhi` instruction actually stores 0 into `r6` (`0xFFFF + 1 = 0x0000`), so that the `movea` instruction stores `0xFFFFFFFF` into `r6`, which is the right value.

`hilo()`

Computes the 32 bit value of the given expression and stores it into the immediate operand field of the given instruction (which must be a `mov` instruction). Use the following example's usage.

```
'mov hilo(here), r6'
```

This computes the absolute address of the `here` label, putting the result into register 6.

`sdaoff()`

Computes the offset of the named variable from the start of the Small Data Area (whose address is held in register 4, the GP register) and stores the result as a 16 bit signed value in the immediate operand field of the given instruction. Use the following example's usage.

```
'ld.w sdaoff(_a_variable)[gp], r6'
```

This loads the contents of the location pointed to by the `_a_variable` label into register 6, provided that the label is located somewhere within $\pm 32\text{K}$ of the address held in the GP register.

IMPORTANT! The linker assumes that the GP register contains a fixed address set to the address of the label called `gp`. This can either be set up automatically by the linker, or specifically set by using the `--defsym __gp=<value>` command line option.

`tdaoff()`

Computes the offset of the named variable from the start of the Tiny Data Area (whose address is held in register 30, the EP register), storing the result as a 4, 5, 7

or 8 bit unsigned value in the immediate operand field of the given instruction. Use the following example's usage.

```
'sld.w tdaoff(_a_variable)[ep],r6'
```

This loads the contents of the location pointed to by the label, `_a_variable`, into register 6, provided that the label is located somewhere within +256 bytes of the address held in the EP register.

IMPORTANT! The linker assumes that the EP register contains a fixed address set to the address of the label called `ep`. This can either be set up automatically by the linker, or specifically set by using the `--defsym __ep=<value>` command line option.

`zdaoff()`

Computes the offset of the named variable from address 0, storing the result as a 16 bit signed value in the immediate operand field of the given instruction. Use the following example's usage.

```
'movea zdaoff(_a_variable),zero,r6'
```

This puts the address of the label, `_a_variable`, into register 6, assuming that the label is somewhere within the first 32K of memory. (Strictly speaking it also possible to access the last 32K of memory as well, as the offsets are signed).

`ctoff()`

Computes the offset of the named variable from the start of the Call Table Area (whose address is held in system register 20, the CTBP register), storing the result a 6 or 16 bit unsigned value in the immediate field of the given instruction or piece of data. Use the following example's usage.

```
'callt ctoff(table_func1)'
```

This will put the function whose address is held in the call table at the location, `table_func1`.

For information on the V850 instruction set, see *V850 Family 32-/16-Bit single-Chip Microcontroller Architecture Manual* from NEC, Ltd.

PowerPC Dependent Features

The following documentation discusses the features pertinent to the PowerPC processors regarding the GNU assembler.

The following options are for the PowerPC when using the GNU assembler.

`-mpwrx` | `-mpwr2` | `-mpwer` | `-m601` | `-mppc` | `-mppc32` | `-m403` | `-m603`
`-m604` | `-mpp64` | `-m620` | `-mppc64bridge`

Select processor type.

`-msolaris`

`-mno-solaris`

`-msolaris` generates code for Sun Solaris targets; `-mno-solaris` does not generate code for Sun Solaris targets.

`-mcom`

Generate Power or PowerPC common instructions.

`-many`

Generate instructions for any PowerPC architecture.

`-mregnames`

`-mno-regnames`

`-mregnames` allows symbolic names for registers; `-mno-regnames` does not allow symbolic names for registers.

`-mrelocatable`

`-mrelocatable-lib`

Flag the generated code as being relocatable.

`-mbig` | `-mbig-endian`

Flag output as big endian.

`-memb`

Flag output as embedded.

`-v`

Display the assembler's version number.

The following directives are for the PowerPC when using the GNU assembler.

`.vbyte` *count*, *value*

Inset *count* bytes with a value of *value* into the output.

`.toc`

Switch to the `.toc` subsegment.

`.stabx` *name*, *value*, *storage class*, *type*

Create a STABS debugging format symbol.

`.rename` *old-name*, *new-name*

Rename *old-name* symbol to *new-name* symbol.

`.tc` *name*, *val1*, *val2*, *val3*, *val4*

Create a TOC entry called *name*, which contains the four values: *val1*, *val2*, *val3*, and *val4*.

Sun Dependent Features

The following documentation discusses the features pertinent to the SPARC processors and the Sun PicoJava processors regarding `as`, the GNU assembler.

The SPARC chip family includes several successive levels (or other variants) of chips, using the same core instruction set, but including a few additional instructions at each level. By default, `as` assumes the core instruction set (SPARC v6), but bumps the architecture level as necessary; it switches to successively higher architectures as it encounters instructions that only exist in the higher levels.

If not configured for SPARC v9 (`sparc64-*-*`), `as` will not bump passed `sparclite` by default; an option must be passed to enable the v9 instructions.

`as` treats `sparclite` as being compatible with v8, unless an architecture is explicitly requested. SPARC v9 is always incompatible with `sparclite`.

`-Av6` | `-Av7` | `-Av8` | `-Asparclet` | `-Asparclite` | `-Av8plus` | `-Av8plusa`
`-Av9` | `-Av9a`

Use one of the `-A` options to select one of the SPARC architectures explicitly. If you select an architecture explicitly, `as` reports a fatal error if it encounters an instruction or feature requiring a higher level. `-Av8plusa` and `-Av9a` enable the SPARC V9 instruction set with UltraSPARC extensions.

`-xarch=v8plus` | `-xarch=v8plusa`

For compatibility with the Solaris v9 assembler. These options are equivalent to `-Av9` and `-Av9a`, respectively.

`-bump`

Warn whenever it is necessary to switch to another level. If an architecture level is explicitly requested, `as` will not issue warnings until that level is reached, and will then bump the level as required (except between incompatible levels).

`-32` | `-64`

Select the word size, either 32 bits or 64 bits. These options are only available with the ELF object file format, and require that the necessary BFD support has been included.

`-TSO` | `-PSO` | `-RMO`

Set the SPARC V9 memory model. Default is `-RMO`.

`-EL` | `-EB`

Set the endianness of the output to either little endian with `-EL`, or big endian with `-EB`.

`--enforce-aligned-data`

Force `.word` and other data directives to generate appropriately aligned data. Off by default so that packed structures can be created using these directives. This option is available to allow compatibility with SunOS and Solaris native assemblers.

`--little-endian-data`

Like `-EL`, except that, if the `-EB` option is used subsequently to select the big endian format, the data will still be emitted in little endian format.

`-no-undeclared-regs`

Generate an error message if an undeclared is used (by the `.register` pseudo-op). Off by default.

`-undeclared-regs`

Restore default behavior having used `-no-undeclared-regs`.

`-no-relax`

Disable the relaxation (automatic shortening) of jumps and branches.

`-relax`

Restore default behavior having used `-no-relax`.

`-KPIC`

Flag the output as being position independent code. Enables some error messages when using non-position independent constructs.

`-v`

Print assembler version number.

`as` normally permits data to be misaligned for SPARC chips. For example, it permits the `.long` pseudo-op to be used on a byte boundary. However, the native SunOS and Solaris assemblers issue an error when they see misaligned data.

You can use the `--enforce-aligned-data` option to make `as` also issue an error to SPARC chips about misaligned data, just as the SunOS and Solaris assemblers do.

The `--enforce-aligned-data` option is not the default because GCC issues

misaligned data pseudo-ops when it initializes certain packed data structures (structures defined using the `packed` attribute). You may have to assemble with `as` in order to initialize packed data structures in your own code.

The SPARC uses IEEE floating-point numbers.

The SPARC version of `as` supports the following additional machine directives:

`.align expr`

Followed by the desired alignment in bytes.

`.common name, value, type`

Followed by a symbol name, a positive number, and "bss", behaving somewhat like `.comm`, although syntax is different.

`.empty`

Suppress warnings about invalid delay slot usage.

`.half`

Functionally identical to `.short`.

`.nword expr`

Produces a 32 bit or a 64 bit value depending on whether the target architecture is a 32 bit or 64 bit architecture.

`.proc`

Ignored. Any text following it on the same line is also ignored.

`.reserve`

Followed by a symbol name, a positive number, and "bss", behaving somewhat like `.lcomm`, although syntax is different.

`.seg name`

Behaves like `.section`, except that it only accepts "text", "data", or "data1", or "text" as section names.

`.skip`

Functionally identical to the `.space` directive.

`.vahalf`

Unaligned version of the `.half` directive.

`.vaword`

Unaligned version of the `.word` directive.

`.vaxword`

Unaligned version of the `.xword` directive.

`.word`

Produces 32 bit values instead of the 16 bit values it produces on many other machines.

`.xword`

Produces 64 bit values.

The following options are for a Sun picoJava processor.

`-mbig`

Generate big endian format output.

`-mlittle`

Generate little endian format output.

Vax Dependent Features

The following documentation discusses the features pertinent to the Vax processors regarding the GNU assembler.

Conversion of flonums to floating point is correct, and compatible with previous assemblers. Rounding is towards zero if the remainder is exactly half the least significant bit. `D`, `F`, `G` and `H` floating point formats are understood. Immediate floating literals (e.g., `S`$6.9`) are rendered correctly. Again, rounding is towards zero in the boundary case. The `.float` directive produces `f` format numbers. The `.double` directive produces `d` format numbers.

All DEC mnemonics are supported. Beware that `case...` instructions have exactly 3 operands. The dispatch table that follows the `case...` instruction should be made with `.word` statements. As far as we know, this is compatible with all UNIX assemblers. The immediate character is `$` for UNIX compatibility, not `#` as DEC writes it. The indirect character is `*` for UNIX compatibility, not `@` as DEC writes it. The displacement sizing character is ``` (an accent grave) for UNIX compatibility, not `^` (a circumflex) as DEC writes it. The letter preceding ``` may have either case. `G` is not understood, but all other letters (`b`, `i`, `l`, `s`, `w`) are understood.

Register names understood are `r0` `r1` `r2...` `r15` `ap` `fp` `sp` `pc`. Upper and lower case letters are equivalent. The following example shows usage.

```
tstb *w`$4(r5)
```

Any expression is permitted in an operand. Operands are comma separated.

Vax bit fields can not be assembled with `as`. Add the required code if needed.

The Vax version of `as` accepts any of the following options, gives a warning message that the option was ignored and proceeds. These options are for compatibility with scripts designed for other people's assemblers.

`-s` (Symbol Table)

`-T` (Token Trace)

These are obsolete options used to debug old assemblers.

`-d` (Displacement size for JUMPs)

This option expects a number following the `-d`. Like options that expect filenames, the number may immediately follow the `-d` (old standard) or constitute the whole of the command line argument that follows `-d` (GNU standard).

`-v` (Virtualize Interpass Temporary File)

Some other assemblers use a temporary file. This option commanded them to keep the information in active memory rather than in a disk file. `as` always does this, so this option is redundant.

`-J` (JUMPify Longer Branches)

Many 32-bit computers permit a variety of branch instructions to do the same job. Some of these instructions are short (and fast) but have a limited range; others are long (and slow) but can branch anywhere in virtual memory. Often there are 3 flavors of branch: short, medium and long. Some other assemblers would emit short and medium branches, unless told by this option to emit short and long branches.

`-t` (Temporary File Directory)

Some other assemblers may use a temporary file, and this option takes a filename being the directory to site the temporary file. Since `as` does not use a temporary disk file, this option makes no difference. `-t` needs exactly one filename.

`-v`

Display assembler version number.

The Vax version of the assembler accepts other options when compiled for VMS.

`-hn`

External symbol or section names (used for global variables) are not case sensitive on VAX/VMS and always map to upper case. This is contrary to the C language definition which explicitly distinguishes upper and lower case. To implement a standard-conforming C compiler, names must be changed (mapped) to preserve the case information. The default mapping is to convert all lower case characters to uppercase and adding an underscore followed by a 6 digit hex value, representing a 24 digit binary value. The one digits in the binary value represent which characters are uppercase in the original symbol name.

The `-hn` option determines how to map names. This takes several values. No `-h` switch at all allows case hacking. A value of zero (`-h0`) implies names should be upper case, and inhibits the case hack. A value of 2 (`-h2`) implies names should be

all lower case, with no case hack. A value of 3 (`-h3`) implies that case should be preserved. The value 1 is unused. The `-H` option directs `as` to display every mapped symbol during assembly. Symbols whose names include a dollar sign (\$) are exceptions to the general name mapping. These symbols are normally only used to reference VMS library names. Such symbols are always mapped to upper case.

`-+`

Causes `as` to truncate any symbol name larger than 31 characters. Also prevents some code following the `_main` symbol normally added to make the object file compatible with Vax-11 “C” version.

`-1`

Ignored; for backward compatibility with `as` version 1.x.

`-H`

Causes `as` to print every symbol which was changed by case mapping.

The assembler supports the following directives for the Vax.

`.dfloat`

Expects zero or more flonums, separated by commas, and assembles Vax `d` format 64-bit floating point constants.

`.ffloat`

Expects zero or more flonums, separated by commas, and assembles Vax `f` format 32-bit floating point constants.

`.gfloat`

Expects zero or more flonums, separated by commas, and assembles Vax `g` format 64-bit floating point constants.

`.hfloat`

Expects zero or more flonums, separated by commas, and assembles Vax `h` format 128-bit floating point constants.

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that reaches the target. Generally these mnemonics are made by substituting `j` for `b` at the start of a DEC mnemonic.

This feature is included both for compatibility and to help compilers. If you do not need this feature, avoid these opcodes. What follows are the mnemonics, and the code into which they can expand.

`jbsb`

`jsb` is already an instruction mnemonic, so `jbsb` is available

(byte displacement)

`bsbb . .`

(word displacement)

`bsbw . . .`

(long displacement)

```

    jsb...
jbr
jr
    Unconditional branch.
    (byte displacement)
    brb...
    (word displacement)
    brw...
    (long displacement)
    jmp...
jCOND
    COND may be any one of the conditional branches neq, nequ, eql, eqlu, gtr, geq,
    lss, gtru, lequ, vc, vs, gequ, cc, lssu, cs. COND may also be one of the bit tests
    bs, bc, bss, bcs, bsc, bcc, bssi, bcci, lbs, lbc. NOTCOND is the opposite
    condition to COND.
    (byte displacement)
    bCOND...
    (word displacement)
    bNOTCOND foo ; brw... ; foo:
    (long displacement)
    bNOTCOND foo ; jmp... ; foo:
jacbX
    x may be one of b d f g h l w.
    (word displacement)
    OPCODE...
    (long displacement)
    OPCODE..., foo ;
    brb bar ; foo: jmp... ;
    bar:
jaobYYY
    YYY may be one of lss leq.
jsobZZZ
    ZZZ may be one of geq gtr.
    (byte displacement)
    OPCODE...
    (word displacement)
    OPCODE... , foo ;
    brb bar ;
    foo: brw destination ;
    bar:
(long displacement)
    OPCODE..., foo ;
    brb bar ;

```

```
        foo: jmp destination ;
        bar:
aobleq
aoblss
sobgeq
sobgtr
    (byte displacement)
        OPCODE...
    (word displacement)
        OPCODE... , foo ;
        brb bar ;
        foo: brw destination ;
        bar:
    (long displacement)
        OPCODE..., foo ;
        brb bar ;
        foo: jmp destination ;
        bar:
```


Zilog Z8000 Dependent Features

The following documentation discusses the features pertinent to the Zilog Z8000 processors regarding the GNU assembler.

The following command line options are available for `as` with the Zilog Z8000 processor.

`-z8001`

Generates code for the Zilog Z8001 processor.

`-z8002`

Generates code for the Zilog Z8001 processor.

There is no specific syntax for `as` with the Zilog Z8000 processor.

The Z8000 that `as` supports both members of the Z8000 family: the unsegmented Z8002, with 16 bit addresses, and the segmented Z8001 with 24 bit addresses. When the assembler is in unsegmented mode (specified with the `unsegm` directive), an address takes up one word (16 bit) sized register. When the assembler is in segmented mode (specified with the `segm` directive), a 24-bit address takes up a long (32 bit) register. See “The Z8000 port of `as` includes these additional assembler directives, for compatibility with other Z8000 assemblers. As shown, these directives do not begin with a period (unlike the ordinary `as` directives).” on page 161 for a list of other Z8000 specific assembler directives.

For detailed information on the Z8000 machine instruction set, see the ***Z8000 Technical Manual***.

! is the line comment character. Use ; instead of a newline to separate statements.

Table 8 on page 160 summarizes the Zilog Z8000 processor opcodes and their

arguments.

Table 8: Zilog Z8000 opcodes and arguments

<i>Opcode</i>	<i>Argument</i>
<i>rs</i>	16 bit source register
<i>rd</i>	16 bit destination register
<i>rbs</i>	8 bit source register
<i>rbd</i>	8 bit destination register
<i>rrs</i>	32 bit source register
<i>rrd</i>	32 bit destination register
<i>rqs</i>	64 bit source register
<i>rqd</i>	64 bit destination register
<i>addr</i>	16/24 bit address
<i>imm</i>	Immediate data

The Z8000 has sixteen 16 bit registers, numbered 0 to 15. You can refer to different sized groups of registers by register number with a prefix: *r*, for 16 bit registers, *rr* for 32 bit registers and *rq* for 64 bit registers. You can also refer to the contents of the first eight (of the sixteen 16 bit registers) by bytes. They are named *rn_h* and *rn_l*.

byte registers

r0l r0h r1h r1l r2h r2l r3h r3l r4h r4l r5h r5l r6h r6l r7h r7l

word registers

r0 r1 r2 r3 r4 r5 r6 r7 r8 r9 r10 r11 r12 r13 r14 r15

long word registers

rr0 rr2 rr4 rr6 rr8 rr10 rr12 rr14

quad word registers

rq0 rq4 rq8 rq12

as understands the following addressing modes for the Z8000.

rn

Register direct

@*rn*

Indirect register

addr

Direct: the 16 bit or 24 bit address (depending on whether the assembler is in segmented or unsegmented mode) of the operand is in the instruction.

address(rn)

Indexed: the 16 or 24 bit address is added to the 16 bit register to produce the final address in memory of the operand.

rn(#imm)

Base Address: the 16 or 24 bit register is added to the 16 bit sign extended immediate displacement to produce the final address in memory of the operand.

`rn(rm)`

Base Index: the 16 or 24 bit register `rn` is added to the sign extended 16 bit index register, `rm`, to produce the final address in memory of the operand.

`#xx`

Immediate data `xx`.

The Z8000 port of `as` includes these additional assembler directives, for compatibility with other Z8000 assemblers. As shown, these directives do not begin with a period (unlike the ordinary `as` directives).

`segm | z8001`

Generates code for the segmented Z8001.

`unsegm | unseg | z8002`

Generates code for the unsegmented Z8002.

`name`

Synonym for `.file`

`global`

Synonym for `.global`

`wval`

Synonym for `.word`

`lval`

Synonym for `.long`

`bval`

Synonym for `.byte`

`sval`

Assemble a string. `sval` expects one string literal, delimited by single quotes. It assembles each byte of the string into consecutive addresses. You can use the escape sequence, `%xx` (where `xx` represents a two-digit hexadecimal number) to represent the character whose ASCII value is `xx`. Use this feature to describe single quote and other characters that may not appear in string literals as themselves. For example, the C statement `'char *a = "he said \\'it's 50% off\\";"` is represented in Z8000 assembly language (shown with the assembler output in hex at the left) as the following example shows.

```

68652073          sval          'he said %22it%27s 50%25
off%22%00'
61696420
22697427
73203530
25206F66
662200
```

`rsect`

synonym for `.section`

`block`

synonym for `.space`

`even`

special case of `.align`; aligns output to even byte boundary.

Acknowledgments for the GNU Assembler

The following individuals contributed to the development of the GNU assembler. If you have contributed to `as` and you want acknowledgment, contact the maintainer to correct the situation; email: nickc@redhat.com (Nick Clifton).

- Dean Elsner wrote the original GNU assembler for the VAX.
- Jay Fenlason maintained `as` for a while, adding support for GDB-specific debug information and the 68k series machines, most of the pre-processing pass, and extensive changes in `messages.c`, `input-file.c`, `write.c`.
- K. Richard Pixley maintained `as` for a while, adding various enhancements and many bug fixes, including merging support for several processors, breaking `as` up to handle multiple object file format back ends (including heavy rewrite, testing, an integration of the COFF and `b.out` back ends), adding configuration including heavy testing and verification of cross assemblers and file splits and renaming, converted `as` to strictly ANSI C including full prototypes, added support for `m680[34]0` and `cpu32`, did considerable work on `i960` including a COFF port (including considerable amounts of reverse engineering), a SPARC opcode file rewrite, DECstation, RS6000, and HP300/HPUX host ports, updated “`know`” assertions and made them work, much other reorganization, cleanup, and lint.
- Ken Raeburn wrote the high-level BFD interface code to replace most of the code in format-specific I/O modules.
- The original VMS support was contributed by David L. Kashtan. Eric Youngdale

- has done much work with it since.
- The Intel 80386 machine description was written by Eliot Dresselhaus.
 - Minh Tran-Le at IntelliCorp contributed some AIX 386 support.
 - The Motorola 88k machine description was contributed by Devon Bowen of Buffalo University and Torbjorn Granlund of the Swedish Institute of Computer Science.
 - Keith Knowles at the Open Software Foundation wrote the original MIPS back end (`tc-mips.c`, `tc-mips.h`), and contributed Rose format support (which hasn't been merged in yet). Ralph Campbell worked with the MIPS code to support `a.out` format.
 - Support for the Zilog Z8k and Hitachi H8/300 and H8/500 processors (`tc-z8k`, `tc-h8300`, `tc-h8500`), and IEEE 695 object file format (`obj-ieee`), was written by Steve Chamberlain. Steve also modified the COFF back end to use BFD for some low-level operations, for use with the H8/300 and AMD 29k targets.
 - John Gilmore built the AMD 29000 support, added `.include` support, and simplified the configuration of which versions accept which directives. He updated the 68k machine description so that Motorola's opcodes always produced fixed-size instructions (e.g., `jsr`), while synthetic instructions remained shrinkable (`jsr`). John fixed many bugs, including true tested cross-compilation support, and one bug in relaxation that took a week and required the proverbial one-bit fix.
 - Ian Lance Taylor merged the Motorola and MIT syntax for the 68k, completed support for some COFF targets (68k, i386 SVR3, and SCO Unix), added support for MIPS, ECOFF and ELF targets, and made a few other minor patches.
 - Steve Chamberlain made `as` able to generate listings.
 - Hewlett-Packard contributed support for the HP9000/300.
 - Jeff Law wrote `as` and BFD support for the native HPPA object format (SOM) along with a fairly extensive HPPA testsuite (for both SOM and ELF object formats). This work was supported by both the Center for Software Science at the University of Utah.
 - Linas Vepstas added `as` support for the ESA/390 IBM 370 architecture.
 - Support for ELF format files has been worked on by Mark Eichin (original, incomplete implementation for SPARC), Pete Hoogenboom and Jeff Law at the University of Utah (HPPA mainly), Michael Meissner (i386 mainly), and Ken Raeburn (SPARC, and some initial 64-bit support).
 - Richard Henderson rewrote the Alpha assembler.
 - Timothy Wall, Michael Hayes, and Greg Smart contributed to the various `tic*` flavors.
 - Klaus Kaempf wrote `as` and BFD support for open VMS/Alpha.

Using binutils

Copyright © 1991-2000 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information within the documentation.

For licenses and use information, see “General Licenses and Terms for Using GNUPro Toolkit” on page 105; specifically, see “GNU General Public License” on page 106, “GNU Lesser General Public License” on page 111, and “Tcl/Tk Tool Command Language and Windowing Toolkit License” on page 118 in *Getting Started Guide*.

1

Overview of `binutils`, the GNU Binary Utilities

The following documentation contains basic descriptions for the GNU binary utilities:

- `ar`, which creates, modifies, and extracts from archives; see “`ar Utility`” on page 169
- `nm`, which lists symbols from object files; see “`nm Utility`” on page 175
- `objcopy`, which copies and translates object files; see “`objcopy Utility`” on page 179
- `objdump`, which displays information from object files; see “`objdump Utility`” on page 185
- `ranlib`, which generates index to archive contents; see “`ranlib Utility`” on page 190
- `size`, which lists file section sizes and total size; see “`size Utility`” on page 191
- `strings`, which lists printable strings from files; see “`strings Utility`” on page 193
- `strip`, which discards symbols; see “`strip Utility`” on page 194
- `c++filt`, which *demangles* encoded C++ symbols; see “`c++filt Utility`” on page 196
- `addr2line`, which converts addresses into file names and line numbers; see “`addr2line Utility`” on page 198
- `nlmconv`, which converts object code into a Netware Loadable Module (NLM); see “`nlmconv Utility`” on page 200

- `windres`, which manipulates Windows resources; see “`windres Utility`” on page 202
- `dlltool`, which is used to create the files needed to build and use *dynamic link libraries* (DLLs); see “`dlltool Utility`” on page 205
- `readelf`, which displays information about one or more ELF format object files; see “`readelf Utility`” on page 210

ar Utility

```
ar [-]p[mod [relpos]] archive [member ...]
ar -M [ <mri-script ]
```

The GNU `ar` program creates, modifies, and extracts from archives. An *archive* is a single file holding a collection of other files in a structure that makes it possible to retrieve the original individual files (called *members* of the archive).

The original files' contents, mode (permissions), timestamp, owner, and group are preserved in the archive, and can be restored on extraction.

GNU `ar` can maintain archives whose members have names of any length; however, depending on how `ar` is configured on your system, a limit on member-name length may be imposed for compatibility with archive formats maintained with other tools. If it exists, the limit is often 15 characters (typical of formats related to `a.out`) or 16 characters (typical of formats related to `coff`).

`ar` is considered a binary utility because archives of this sort are most often used as libraries holding commonly needed subroutines.

`ar` creates an index to the symbols defined in relocatable object modules in the archive when you specify the modifier, `s`. Once created, this index is updated in the archive whenever `ar` makes a change to its contents (save for the `q` update operation). An archive with such an index speeds up linking to the library, and allows routines in the library to call each other without regard to their placement in the archive.

You may use `nm -s` or `nm --print-armap` to list this index table. If an archive lacks the table, another form of `ar` called `ranlib` can be used to add only the table.

GNU `ar` is designed to be compatible with two different facilities. You can control its activity using command-line options like the different varieties of `ar` on Unix systems; or, if you specify the single command-line option `-M` you can control it with a script supplied via standard input, like the MRI *librarian* program.

Controlling `ar` on the Command Line

```
ar [-]p[mod [relpos]] archive [member ...]
```

When you use `ar` in the Unix style, `ar` insists on at least two arguments to execute: one keyletter (`p`) specifying the *operation* (optionally accompanied by other keyletters specifying *mod*, or *modifiers*, *relpos* `archive [member ...]`, or the name of an existing archive member), and the archive name (*archive*) to which to archive.

Most operations can also accept further member arguments, specifying particular files on which to perform operations. GNU `ar` allows you to mix the operation code `p` and

modifier flags (*mod*) in any order, within the first command-line argument. You may begin the first command-line argument with a dash.

The *p* keyletter specifies what operation to execute; it may be any of the following, but you must specify only one of them:

d

Delete modules from the archive. Specify the names of modules to be deleted as *member...*; the archive is untouched if you specify no files to delete. If you specify the *v* modifier, *ar* lists each module as it is deleted.

m

Move members in an archive.

The ordering of members in an archive can make a difference in how programs are linked using the library, if a symbol is defined in more than one member.

If no modifiers are used with *m*, any members you name in the *member* arguments are moved to the *end* of the archive; you can use the *a*, *b*, or *i* modifiers to move them to a specified place instead.

p

Print the specified members of the archive, to the standard output file. If the *v* modifier is specified, show the member name before copying its contents to standard output.

If you specify no *member* arguments, all the files in the archive are printed.

q

Quick append; add the files *member...* to the end of *archive*, without checking for replacement. The modifiers *a*, *b*, and *i* do *not* affect this operation; new members are always placed at the end of the archive. The modifier *v* makes *ar* list each file as it is appended. Since the point of this operation is speed, the archive's symbol table index is not updated, even if it already existed; you can use *ar s* or *ranlib* explicitly to update the symbol table index.

r

Replacement; inserts the files *member...* into *archive*. This operation differs from *q* in that any previously existing members are deleted if their names match those being added. If one of the files named in *member...* does not exist, *ar* displays an error message, and leaves undisturbed any existing members of the archive matching that name. By default, new members are added at the end of the file; but you may use one of the modifiers *a*, *b*, or *i* to request placement relative to some existing member. The modifier *v* used with this operation elicits a line of output for each file inserted, along with one of the letters, *a* or *r*, to indicate whether the file was appended (no old member deleted) or replaced.

t

Display a *table* listing the contents of *archive*, or those of the files listed in *member...* that are present in the archive. Normally only the member name is

shown; if you also want to see the modes (permissions), timestamp, owner, group, and size, you can request that by also specifying the `v` modifier. If you do not specify a *member*, all files in the archive are listed. If there is more than one file with the same name (for instance, `file`) in an archive (for instance, `b.a`), `ar t b.a file` lists only the first instance; to see them all, you must ask for a complete listing—in the earlier example, `ar t b.a`.

x

Extract members (named *member*) from the archive. You can use the `v` modifier with this operation, to request that `ar` list each name as it extracts it. If you do not specify a *member*, all files in the archive are extracted.

A number of modifiers (*mod*) may immediately follow the `p` keyletter, to specify variations on an operation's behavior:

a

Add new files *after* an existing member of the archive. If you use the modifier, `a`, the name of an existing archive member must be present as the `relpos` argument, before the archive specification.

b

Add new files *before* an existing member of the archive. If you use the modifier, `b`, the name of an existing archive member must be present as the `relpos` argument, before the archive specification. (same as `i`).

c

Create the archive. The specified *archive* is always created if it did not exist, when you request an update. But a warning is issued unless you specify in advance that you expect to create it, by using this modifier.

f

Truncate names in the archive. GNU `ar` will normally permit file names of any length. This will cause it to create archives which are not compatible with the native `ar` program on some systems. If this is a concern, the `f` modifier may be used to truncate file names when putting them in the archive.

i

Insert new files *before* an existing member of the archive. If you use the modifier, `i`, the name of an existing archive member must be present as the `relpos` argument, before the *archive* specification. (same as `b`).

l

This modifier is accepted but not used.

o

Preserve the *original* dates of members when extracting them. If you do not specify this modifier, files extracted from the archive are stamped with the time of extraction.

s

Write an object-file index into the archive, or update an existing one, even if no other change is made to the archive. You may use this modifier flag either with any operation, or alone. Running `ar s` on an archive is equivalent to running `ranlib` on it.

S

Do not generate an archive symbol table. This can speed up building a large library in several steps. The resulting archive can not be used with the linker. In order to build a symbol table, you must omit the `s` modifier on the last execution of `ar`, or you must run `ranlib` on the archive.

u

Normally, `ar r ...` inserts all files listed into the archive. If you would like to insert *only* those of the files you list that are newer than existing members of the same names, use this modifier. The `u` modifier is allowed only for the *replace* operation, using `r`. In particular, the `qu` combination is not allowed, since checking the timestamps would lose any speed advantage from the operation, `q`.

v

This modifier requests the verbose version of an operation. Many operations display additional information, such as file-names processed, when the modifier, `v` is appended.

V

This modifier shows the version number of `ar`.

Controlling `ar` with a Script

```
ar -M [ < script ]
```

If you use the single command-line option, `-M`, with `ar`, you can control its operation with a rudimentary command language. This form of `ar` operates interactively if standard input is coming directly from a terminal. During interactive use, `ar` prompts for input (the prompt is `AR >`), and continues executing even after errors. If you redirect standard input to a script file, no prompts are issued, and `ar` abandons execution (with a nonzero exit code) on any error.

The `ar` command language is *not* designed to be equivalent to the command-line options; in fact, it provides somewhat less control over archives. The only purpose of the command language is to ease the transition to GNU `ar` for developers who already have scripts written for the MRI *librarian* program.

The syntax for the `ar` command language is straightforward as the following discussion details.

- Commands are recognized in upper or lower case; for example, `LIST` is the same

as *list*. In the following descriptions, commands are shown in upper case for clarity.

- A single command may appear on each line; it is the first word on the line.
- Empty lines are allowed, and have no effect.
- Comments are allowed; text is ignored after either of the `*` or `;` characters.
- Whenever you use a list of names as part of the argument to an `ar` command, you can separate the individual names with either commas or blanks. Commas are shown in the following explanations, for clarity.
- `+` is used as a line continuation character; if `+` appears at the end of a line, the text on the following line is considered part of the current command.

The following are the commands you can use in `ar` scripts, or when using `ar` interactively. Three of them have special significance:

`OPEN` or `CREATE` specifies a *current* archive, a temporary file required for most of the other commands.

`SAVE` commits the changes so far specified by the script. Prior to `SAVE`, commands affect only the temporary copy of the current archive.

`ADDLIB` *archive*

`ADDLIB` *archive*(*module*, *module*, ...*module*)

Add all the contents of *archive* (or, if specified, each named *module* from *archive*) to the current archive. Requires prior use of `OPEN` or `CREATE`.

`ADDMOD` *member*, *member*, ...*member*

Add each named *member* as a module in the current archive. Requires prior use of `OPEN` or `CREATE`.

`CLEAR`

Discard the contents of the current archive, canceling the effect of any operations since the last `SAVE`. May be executed (with no effect) even if no current archive is specified.

`CREATE` *archive*

Creates an archive, and makes it the current archive (required for many other commands). The new archive is created with a temporary name; it is not actually saved as *archive* until you use `SAVE`. You can overwrite existing archives; similarly, the contents of any existing file named *archive* will not be destroyed until `SAVE`.

`DELETE` *module*, *module*, ...*module*

Delete each listed *module* from the current archive. Equivalent to a `ar -d` *archive* *module* ...*module* statement. Requires prior use of `OPEN` or `CREATE`.

`DIRECTORY` *archive*(*module*, ...*module*)

`DIRECTORY` *archive*(*module*, ...*module*) *outputfile*

List each named *module* present in *archive*. The separate command `VERBOSE`

specifies the form of the output: when verbose output is off, output is like that of `ar -t archive module ...`. When verbose output is on, the listing is like a `ar -tv archive module ...` statement. Output normally goes to the standard output stream; however, if you specify *outputfile* as a final argument, `ar` directs the output to that file.

END

Exit from `ar`, with a 0 exit code to indicate successful completion. This command does not save the output file; if you have changed the current archive since the last `SAVE` command, those changes are lost.

EXTRACT *module, module, ...module*

Extract each named *module* from the current archive, writing them into the current directory as separate files. Equivalent to `ar -x archive module ...` statement. Requires prior use of `OPEN` or `CREATE`.

LIST

Display full contents of the current archive, in *verbose* style regardless of the state of `VERBOSE`. The effect is like `ar tv archive`. This single command is a GNU `ld` enhancement, rather than present for MRI compatibility. Requires prior use of `OPEN` or `CREATE`.

OPEN *archive*

Opens an existing archive for use as the current archive (required for many other commands). Any changes as the result of subsequent commands will not actually affect *archive* until you next use `SAVE`.

REPLACE *module, module, ...module*

In the current archive, replace each existing *module* (named in the `REPLACE` arguments) from files in the current working directory. To execute this command without errors, both the file, and the module in the current archive, must exist. Requires prior use of `OPEN` or `CREATE`.

VERBOSE

Toggle an internal flag governing the output from `DIRECTORY`. When the flag is on, `DIRECTORY` output matches output from `ar -tv ...`

SAVE

Commit your changes to the current archive, and actually save it as a file with the name specified in the last `CREATE` or `OPEN` command. Requires prior use of `OPEN` or `CREATE`.

nm Utility

```
nm [ -a | --debug-syms ] [ -g | --extern-only ]
  [ -B ] [ -C | --demangle ] [ -D | --dynamic ]
  [ -s | --print-arnmap ] [ -A | -o | --print-file-name ]
  [ -n | -v | --numeric-sort ] [ -p | --no-sort ]
  [ -r | --reverse-sort ] [ --size-sort ] [ -u | --undefined-only ]
  [ -t radix | --radix=radix ] [ -P | --portability ]
  [ --target=bfdname ] [ -f format | --format=format ]
  [ --defined-only ] [ -l | --line-numbers ]
  [ --no-demangle ] [ -V | --version ]
  [ --help ]
  [ object-file ... ]
```

GNU `nm` lists the symbols from object files *object-file* ... If no object files are listed as arguments, `nm` assumes `a.out`. For each symbol, `nm` shows:

- The symbol value, in the radix selected by the following options, or hexadecimal by default.
- The symbol type. At least the following types are used; others are, as well, depending on the object file format. If lowercase, the symbol is local; if uppercase, the symbol is global (external).

A

The symbol's value is absolute, and will not be changed by further linking.

B

The symbol is in the uninitialized data section (known as BSS).

C

The symbol is common. Common symbols are uninitialized data. When linking, multiple common symbols may appear with the same name. If the symbol is defined anywhere, the common symbols are treated as undefined references. For more details on common symbols, see the discussion of `--warn-common` in “Using `ld` Command Line Options” in *Using `ld` in GNUPro Development Tools*.

D

The symbol is in the initialized data section.

G

The symbol is in an initialized data section for small objects. Some object file formats permit more efficient access to small data objects, such as a global `int` variable as opposed to a large global array.

I

The symbol is an indirect reference to another symbol. This is a GNU extension to the `a.out` object file format which is rarely used.

- N The symbol is a debugging symbol.
- R The symbol is in a read only data section.
- S The symbol is in an uninitialized data section for small objects.
- T The symbol is in the text (code) section.
- U The symbol is undefined.
- W The symbol is weak. When a weak defined symbol is linked with a normal defined symbol, the normal defined symbol is used with no error. When a weak undefined symbol is linked and the symbol is not defined, the value of the weak symbol becomes zero with no error.
- The symbol is a `stabs` symbol in an `a.out` object file. In this case, the next values printed are the `stabs other` field, the `stabs desc` field, and the `stab type`. `stabs` symbols are used to hold debugging information; for more information, see “.`stabd`, `.stabn`, and `.stabs Directives`” on page 66 in *Using as*.
- ? The symbol type is unknown, or object file format specific.

- The symbol name.

The long and short forms of options, shown here as alternatives, are equivalent.

`-A`

`-o`

`--print-file-name`

Precede each symbol by the name of the input file (or archive element) in which it was found, rather than identifying the input file once only, before all of its symbols.

`-a`

`--debug-syms`

Display all symbols, even debugger-only symbols; normally these are not listed.

`-B`

The same as `--format=bsd` (for compatibility with the MIPS `nm`).

`-C`

`--demangle`

Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++

function names readable. See “c++filt Utility” on page 196 for more information on demangling.

--no-demangle

Do not *demangle* low-level symbol names. This is the default.

-D

--dynamic

Display the dynamic symbols rather than the normal symbols. This is only meaningful for dynamic objects, such as certain types of shared libraries.

-f *format*

--format=*format*

Use the output format, *format*, which can be `bsd`, `sysv`, or `posix`. The default is `bsd`. Only the first character of *format* is significant; it can be either upper or lower case.

-g

--extern-only

Display only external symbols.

-l

--line-numbers

For each symbol, use debugging information to try to find a filename and line number. For a defined symbol, look for the line number of the address of the symbol. For an undefined symbol, look for the line number of a relocation entry that refers to the symbol. If line number information can be found, print it after the other symbol information.

-n

-v

--numeric-sort

Sort symbols numerically by their addresses, rather than alphabetically by their names.

-p

--no-sort

Do not bother to sort the symbols in any order; print them in the order encountered.

-P

--portability

Use the POSIX.2 standard output format instead of the de-fault format. Equivalent to `-f posix`.

-s

--print-arnmap

When listing symbols from archive members, include the index: a mapping (stored in the archive by `ar` or `ranlib`) of which modules contain definitions for which names.

`-r`
`--reverse-sort`
Reverse the order of the sort (whether numeric or alphabetic); let the last come first.

`--size-sort`
Sort symbols by `size`. `size` is computed as the difference between the value of the symbol and the value of the symbol with the next higher value. The size of the symbol is printed, rather than the value.

`-t radix`
`--radix=radix`
Use *radix* as the radix for printing the symbol values. It must be `d` for decimal, `o` for octal, or `x` for hexadecimal.

`--target=bfdname`
Specify an object code format other than your system's default format. See "Target Selection" on page 214, for more information.

`-u`
`--undefined-only`
Display only undefined symbols (those external to each object file).

`--defined-only`
Display only defined symbols for each object file.

`-v`
`--version`
Show the version number of `nm` and exit.

`--help`
Show a summary of the options to `nm` and exit.

objcopy Utility

```
objcopy [ -F bfdname | --target=bfdname ]
[ -I bfdname | --input-target=bfdname ]
[ -O bfdname | --output-target=bfdname ]
[ -S | --strip-all ] [ -g | --strip-debug ]
[ -K symbolname | --keep-symbol=symbolname ]
[ -N symbolname | --strip-symbol=symbolname ]
[ -L symbolname | --localize-symbol=symbolname ]
[ -W symbolname | --weaken-symbol=symbolname ]
[ -x | --discard-all ] [ -X | --discard-locals ]
[ -b byte | --byte=byte ]
[ -i interleave | --interleave=interleave ]
[ -R sectionname | --remove-section=sectionname ]
[ -p | --preserve-dates ] [ --debugging ]
[ --gap-fill=val ] [ --pad-to=address ]
[ --set-start=val ] [ --adjust-start=incr ]
[ --change-address=incr ]
[ --change-section-address=section{=,+,-}val ]
[ --change-warnings ] [ --no-change-warnings ]
[ --set-section-flags=section=flags ]
[ --add-section=sectionname=filename ]
[ --change-leading-char ] [ --remove-leading-char ]
[ --weaken ]
[ -v | --verbose ] [ -V | --version ] [ --help ]
input-file [outfile]
```

The GNU `objcopy` utility copies the contents of an object file to another. `objcopy` uses the GNU BFD Library to read and write the object files. It can write the destination object file in a format different from that of the source object file. The exact behavior of `objcopy` is controlled by command-line options.

`objcopy` creates temporary files to do its translations and deletes them afterward. `objcopy` uses BFD to do all its translation work; it has access to all the formats described in BFD and is able to recognize most formats without being told explicitly. See “BFD Library” and “The BFD Canonical Object File Format” in *Using 1d in GNUPro Development Tools*.

`objcopy` can be used to generate S-records by using an output target of `srec` (use `-O srec`).

`objcopy` can be used to generate a raw binary file by using an output target of `binary` (meaning `-O binary`). When `objcopy` generates a raw binary file, it will essentially produce a memory dump of the contents of the input object file. All symbols and relocation information will be discarded. The memory dump will start at the virtual address of the lowest section copied into the output file.

When generating an S-record or a raw binary file, it may be helpful to use `-s` to remove sections containing debugging information. In some cases `-R` will be useful to remove sections containing information that is not needed by the binary file.

input-file

outfile

The source and output files, respectively. If you do not specify *outfile*, `objcopy` creates a temporary file and destructively renames the result with the name of *input-file*.

`-I bfdname`

`--input-target=bfdname`

Consider the source file's object format to be *bfdname*, rather than attempting to deduce it. See “Target Selection” on page 214 for more information.

`-O bfdname`

`--output-target=bfdname`

Write the output file using the object format, *bfdname*. See “Target Selection” on page 214 for more information.

`-F bfdname`

`--target=bfdname`

Use *bfdname* as the object format for both the input and the output file; i.e., simply transfer data from source to destination with no translation. See “Target Selection” on page 214 for more information.

`-R sectionname`

`--remove-section=sectionname`

Remove any section named *sectionname* from the output file. This option may be given more than once.

NOTE: Using this option inappropriately may make the output file unusable.

`-S`

`--strip-all`

Do not copy relocation and symbol information from the source file.

`-g`

`--strip-debug`

Do not copy debugging symbols from the source file.

`--strip-unneeded`

Strip all symbols that are not needed for relocation processing.

`-K symbolname`

`--keep-symbol=symbolname`

Copy only symbol *symbolname* from the source file. This option may be given more than once.

`-N symbolname`
`--strip-symbol=symbolname`
Do not copy symbol *symbolname* from the source file. This option may be given more than once, and may be combined with strip options other than `-k`.

`-L symbolname`
`--localize-symbol=symbolname`
Make symbol *symbolname* local to the file, so that it is not visible externally. This option may be given more than once.

`-W symbolname`
`--weaken-symbol=symbolname`
Make symbol *symbolname* weak. This option may be given more than once.

`-x`
`--discard-all`
Do not copy non-global symbols from the source file.

`-X`
`--discard-locals`
Do not copy compiler-generated local symbols (these usually start with `L` or `.`).

`-b byte`
`--byte=byte`
Keep only every *byte* of the input file (header data is not affected). *byte* can be in the range from 0 to *interleave*-1, where *interleave* is given by the `-i` or `--interleave` option, or the default of 4. This option is useful for creating files to program ROM. It is typically used with an `srec` output target.

`-i interleave`
`--interleave=interleave`
Only copy one out of every *interleave* bytes. Select which byte to copy with the `-b` or `--byte` option. The default is 4. `objcopy` ignores this option if you do not specify either `-b` or `--byte`.

`-p`
`--preserve-dates`
Set the access and modification dates of the output file to be the same as those of the input file.

`--debugging`
Convert debugging information, if possible. This is not the default because only certain debugging formats are supported, and the conversion process can be time consuming.

`--gap-fill val`
Fill gaps between sections with *val*. This is done by increasing the size of the section with the lower address, and filling in the extra space created with *val*.

`--pad-to address`
Pad the output file up to the virtual address *address*. This is done by increasing

the size of the last section. The extra space is filled in with the value specified by `--gap-fill` (default zero).

`--set-start val`

Set the address of the new file to *val*. Not all object file formats support setting the start address.

`--change-start incr`

`--adjust-start incr`

Change the start address by adding *incr*. Not all object file formats support setting the start address.

`--change-addresses incr`

`--adjust-vma incr`

Change the VMA and LMA addresses of all sections, *section*., as well as the start address, by adding *incr*. Some object file formats do not permit section addresses to be changed arbitrarily.

WARNING! This does not relocate the sections; if the program expects sections to be loaded at a certain address, and this option is used to change the sections such that they are loaded at a different address, the program may fail.

`--change-section-address section{=,+,-}val`

`--adjust-section-vma section{=,+,-}val`

Set or change both the VMA address and the LMA address of the named section.

If `=` is used, the section address is set to *val*. Otherwise, *val* is added to or subtracted from the section address. See the comments for `--change-addresses`, the previous option. If *section* does not exist in the input file, a warning will be issued, unless `--no-change-warnings` is used.

`--change-section-lma section{=,+,-}val`

Set or change the LMA address of the named *section*. The LMA address is the address where the section will be loaded into memory at program load time.

Normally this is the same as the VMA address, which is the address of the section at program run time, but on some systems, especially those where a program is held in ROM, the two can be different. If `=` is used, the section address is set to *val*. Otherwise, *val* is added to or subtracted from the section address. See the comments for `--change-addresses`. If *section* does not exist in the input file, a warning will be issued, unless `--no-change-warnings` is used.

`--change-section-vma section{=,+,-}val`

Set or change the VMA address of the named *section*. The VMA address is the address where the section will be located once the program has started executing. Normally this is the same as the LMA address, which is the address where the section will be loaded into memory, but on some systems, especially those where a program is held in ROM, the two can be different. If `=` is used, the section address is set to *val*. Otherwise, *val* is added to or subtracted from the section

address. See the comments for `--change-addresses`. If *section* does not exist in the input file, a warning will be issued, unless `--no-change-warnings` is used.

`--change-warnings`

`--adjust-warnings`

If `--change-section-address` or `--change-section-lma` or

`--change-section-vma` is used, and the named section does not exist, issue a warning.

This is the default.

`--no-change-warnings`

`--no-adjust-warnings`

Do not issue a warning if `--change-section-address` or

`--adjust-section-lma` or `--adjust-section-vma` is used, even if the named section does not exist.

`--set-section-flags section=flags`

Set the flags for the named section. The *flags* argument is a comma separated string of flag names. The recognized names are `alloc`, `load`, `readonly`, `code`, `data`, and `rom`. You can set the `contents` flag for a section which does not have contents, but it is not meaningful to clear the `contents` flag of a section which does have contents; just remove the section instead. Not all flags are meaningful for all object file formats.

`--add-section sectionname=filename`

Add a new section named *sectionname* while copying the file. The contents of the new section are taken from the file *filename*. The size of the section will be the size of the file. This option only works on file formats which can support sections with arbitrary names.

`--change-leading-char`

Some object file formats use special characters at the start of symbols. The most common such character is underscore, which compilers often add before every symbol. This option tells `objcopy` to change the leading character of every symbol when it converts between object file formats. If the object file formats use the same leading character, this option has no effect. Otherwise, it will add a character, or remove a character, or change a character, as appropriate.

`--remove-leading-char`

If the first character of a global symbol is a special symbol leading character used by the object file format, remove the character. The most common symbol leading character is underscore. This option will remove a leading underscore from all global symbols. This can be useful if you want to link together objects of different file formats with different conventions for symbol names.

`--weaken`

Change all global symbols in the file to be weak. This can be useful when building an object that will be linked against other objects using the `-R` option to the linker.

This option is only effective when using an object file format that supports weak symbols.

`-V`

`--version`

Show the version number of `objcopy`.

`-v`

`--verbose`

Verbose output: list all object files modified. In the case of archives, `objcopy -V` lists all members of the archive.

`--help`

Show a summary of the options to `objcopy`.

objdump Utility

```
objdump [ -a | --archive-headers ]
[ -b bfdname | --target=bfdname ] [ --debugging ]
[ -C | --demangle ] [ -d | --disassemble ]
[ -D | --disassemble-all ] [ --disassemble-zeroes ]
[ -EB | -EL | --endian={big | little} ]
[ -f | --file-headers ]
[ -h | --section-headers | --headers ] [ -i | --info ]
[ -j section | --section=section ]
[ -l | --line-numbers ] [ -S | --source ]
[ -m machine | --architecture=machine ]
[ -p | --private-headers ]
[ -r | --reloc ] [ -R | --dynamic-reloc ]
[ -s | --full-contents ] [ --stabs ]
[ -t | --syms ] [ -T | --dynamic-syms ] [ -x | --all-headers ]
[ -w | --wide ] [ --start-address=address ]
[ --stop-address=address ]
[ --prefix-addresses ] [ --[no]show-raw-insn ]
[ --adjust-vma=offset ]
[ --version ] [ --help ]
[ object-file... ]
```

`objdump` displays information about one or more object files. The options control what particular information to display. This information is mostly useful to programmers who are working on the compilation tools, as opposed to programmers who just want their program to compile and work.

object-file... are the object files to be examined. When you specify archives, `objdump` shows information on each of the member object files.

The long and short forms of options, shown here as alternatives, are equivalent. At least one option besides `-l` must be given.

`-a`

`--archive-header`

If any of the object-file files are archives, display the archive header information (in a format similar to `ls -l`). Besides the information you could list with `ar tv`, `objdump -a` shows the object file format of each archive member.

`--adjust-vma=offset`

When dumping information, first add *offset* to all the section addresses. This is useful if the section addresses do not correspond to the symbol table, which can happen when putting sections at particular addresses when using a format that can not represent section addresses, such as `a.out`.

`-b bfdname`

`--target=bfdname`

Specify that the object-code format for the object files is BFD-name. This option may not be necessary; `objdump` can automatically recognize many formats.

`objdump -b oasys -m vax -h fu.o`

The previous example displays summary information from the section headers (`-h`) of `fu.o`, which is explicitly identified (`-m`) as a Vax object file in the format produced by Oasys compilers. You can list the formats available with the `-i` option. See “Target Selection” on page 214 for more information.

`-C`

`--demangle`

Decode (demangle) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See “c++filt Utility” on page 196 for more information on demangling.

`--debugging`

Display debugging information. This attempts to parse debugging information stored in the file and print it out using a C like syntax. Only certain types of debugging information have been implemented.

`-d`

`--disassemble`

Display the assembler mnemonics for the machine instructions from *object-file*. This option only disassembles those sections which are expected to contain instructions.

`-D`

`--disassemble-all`

Like `-d`, but disassembles the contents of all sections, not just those expected to contain instructions.

`--prefix-addresses`

When disassembling, print the complete address on each line. This is the older disassembly format.

`--disassemble-zeroes`

Normally the disassembly output will skip blocks of zeroes.

This option directs the disassembler to disassemble those blocks, just like any other data.

`-EB`

`-EL`

`--endian={big | little}`

Specify the endianness of the object files. This only affects disassembly. This can be useful when disassembling a file format that does not describe endianness information, such as S-records.

`-f`
`--file-header`
Display summary information from the overall header of each of the *object-file* files.

`-h`
`--section-header`
`--header`
Display summary information from the section headers of the object file.
File segments may be relocated to nonstandard addresses, for example by using the `-Ttext`, `-Tdata`, or `-Tbss` options to `ld`. However, some object file formats, such as `a.out`, do not store the starting address of the file segments. In those situations, although `ld` relocates the sections correctly, using `objdump -h` to list the file section headers cannot show the correct addresses. Instead, it shows the usual addresses, which are implicit for the target.

`--help`
Print a summary of the options to `objdump` and exit.

`-i`
`--info`
Display a list showing all architectures and object formats available for specification with `-b` or `-m`.

`-j name`
`--section=name`
Display information only for section *name*.

`-l`
`--line-numbers`
Label the display (using debugging information) with the filename and source line numbers corresponding to the object code or relocs shown. Only useful with `-d` or `-D` or `-r`.

`-m machine`
`--architecture=machine`
Specify the architecture to use when disassembling object files, *object-file*. This can be useful when disassembling a file format that does not describe architecture information, such as S-records. You can list available architectures using the `-i` option.

`-p`
`--private-headers`
Print information that is specific to the object file format. The exact information printed depends upon the object file format. For some object file formats, no additional information is printed.

- `-r`
 - `--reloc`

Print the relocation entries of the file. If used with `-d` or `-D` options, the relocations are printed interspersed with the disassembly.
- `-R`
 - `--dynamic-reloc`

Print the dynamic relocation entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries.
- `-s`
 - `--full-contents`

Display the full contents of any sections requested.
- `-S`
 - `--source`

Display source code intermixed with disassembly, if possible.
Implies `-d`.
 - `--show-raw-insn`

When disassembling instructions, print the instruction in hex as well as in symbolic form. Not all targets handle this correctly yet.
 - `--no-show-raw-insn`

When disassembling instructions, do not print the instruction bytes. This is the default when using `--prefix-addresses`.
 - `--stabs`

Display the full contents of any sections requested. Display the contents of the `.stab` and `.stab.index` and `.stab.excl` sections from an ELF file. This is only useful on systems (such as Solaris 2.0) in which `.stab` debugging symbol-table entries are carried in an ELF section. In most other file formats, debugging symbol-table entries are interleaved with linkage symbols, and are visible in the `--syms` output. For more information on `stabs` symbols, see “`.stabd`, `.stabn`, and `.stabs` Directives” on page 66 in *Using AS*.
 - `--start-address=address`

Start displaying data at the specified address. This affects the output of the `-d`, `-r` and `-s` options.
 - `--stop-address=address`

Stop displaying data at the specified address. This affects the output of the `-d`, `-r` and `-s` options.
 - `-t`
 - `--syms`

Print the symbol table entries of the file. This is similar to the information provided by the `nm` program.

- T
- dynamic-syms
Print the dynamic symbol table entries of the file. This is only meaningful for dynamic objects, such as certain types of shared libraries. This is similar to the information provided by the `nm` program when given the `-D` (`--dynamic`) option.
- version
Print the version number of `objdump` and exit.
- x
- all-header
Display all available header information, including the symbol table and relocation entries.
Using `-x` is equivalent to specifying all of `-a -f -h -r -t`.
- w
- wide
Format some lines for output devices having more than 80 columns.

`ranlib` Utility

```
ranlib [-vV] archive
```

`ranlib` generates an index to the contents of an archive and stores it in the archive. The index lists each symbol defined by a member of an archive that is a relocatable object file.

You may use `nm -s` or `nm --print-arnmap` to list this index.

An archive with such an index speeds up linking to the library and allows routines in the library to call each other without regard to their placement in the archive.

The GNU `ranlib` program is another form of GNU `ar`; running `ranlib` is completely equivalent to executing `ar -s`. See “`ar` Utility” on page 169.

`-v`

`-V`

Show the version number of `ranlib`.

size Utility

```
size [ -A | -B | --format=compatibility ]
      [ --help ] [ -d | -o | -x | --radix=number ]
      [ --target=bfdname ] [ -V | --version ]
      [ object-file... ]
```

The GNU `size` utility lists the section sizes—and the total size—for each of the object or archive files *object-file* in its argument list. By default, one line of output is generated for each object file or each module in an archive.

object-file ... are the object files to be examined.

The command line options have the following meanings.

-A

-B

--format=*compatibility*

Using one of these options, you can choose whether the output from GNU `size` resembles output from System V `size` (using -A, or --format=sysv), or Berkeley `size` (using -B, or --format=berkeley). The default is the one-line format similar to Berkeley's.

The following is an example of the Berkeley (default) format of output from `size`.

```
size --format=Berkeley ranlib size
text    data    bss    dec    hex    filename
294880  81920    11592  388392  5ed28  ranlib
294880  81920    11888  388688  5ee50  size
```

The following example shows the same data, but displayed closer to System V conventions.

```
size --format=SysV ranlib size
ranlib :
section      size      addr
.text        294880    8192
.data        81920    303104
.bss         11592    385024
Total        388392

size :
section      size      addr
.text        294880    8192
.data        81920    303104
.bss         11888    385024
Total        388688
```

--help

Shows a summary of acceptable arguments and options.

`-d`

`-o`

`-x`

`--radix=number`

Using one of these options, you can control whether the size of each section is given: in decimal (`-d`, or `--radix=10`); in octal (`-o`, or `--radix=8`); or, in hexadecimal (`-x`, or `--radix=16`).

In `--radix=number`, only the three values (8, 10, 16) are supported. The total size is always given in two radices; decimal and hexadecimal for `-d` or `-x` output; or octal and hexadecimal if you're using `-o`.

`--target=bfdname`

Specify that the object-code format for *object-file* is *bfdname*. This option may not be necessary; *size* can automatically recognize many formats. See “Target Selection” on page 214 for more information.

`-V`

`--version`

Display the version number of *size*.

strings Utility

```
strings [-afov] [-min-len] [-n min-len] [-t radix] [-]
        [--all] [--print-file-name] [--bytes=min-len]
        [--radix=radix] [--target=bfdname]
        [--help] [--version] file ...
```

For each *file* given, GNU *strings* prints the printable character sequences that are at least 4 characters long (or the number given with the following options) and are followed by an unprintable character. By default, it only prints the strings from the initialized and loaded sections of object files; for other types of files, it prints the strings from the whole file.

strings is mainly useful for determining the contents of non-text files.

```
-a
--all
-
```

Do not scan only the initialized and loaded sections of object files; scan the whole files.

```
-f
--print-file-name
```

Print the name of the file before each string.

```
--help
```

Print a summary of the program usage on the standard output and exit.

```
-min-len
```

```
-n min-len
```

```
--bytes=min-len
```

Print sequences of characters that are at least *min-len* characters long, instead of the default 4.

```
-o
```

Like `-t o`. Some other versions of *strings* have `-o` act like `-t d`. Since we can not be compatible with both ways, we simply chose one.

```
-t radix
```

```
--radix=radix
```

Print the offset within the file before each string. The single character argument specifies the radix of the offset—`o` for octal, `x` for hexadecimal, or `d` for decimal.

```
--target=bfdname
```

Specify an object code format other than your system's default format. See “Target Selection” on page 214 for more information.

```
-v
```

```
--version
```

Print the program version number on the standard output and exit.

strip Utility

```
strip  [ -F bfdname | --target=bfdname ]  
      [ -I bfdname | --input-target=bfdname ]  
      [ -O bfdname | --output-target=bfdname ]  
      [ -s | --strip-all ] [ -S | -g | --strip-debug ]  
      [ -K symbolname | --keep-symbol=symbolname ]  
      [ -N symbolname | --strip-symbol=symbolname ]  
      [ -x | --discard-all ] [ -X | --discard-locals ]  
      [ -R sectionname | --remove-section=sectionname ]  
      [ -o file ] [ -p | --preserve-dates ]  
      [ -v | --verbose ] [ -V | --version ] [ --help ]  
      [ object-file... ]
```

GNU `strip` discards all symbols from object files *object-file*. The list of object files may include archives. At least one object file must be given. `strip` modifies the files named in its argument, rather than writing modified copies under different names.

`-F bfdname`

`--target=bfdname`

Treat the original *object-file* as a file with the object code format *bfdname*, and rewrite it in the same format. See “Target Selection” on page 214 for more information.

`--help`

Show a summary of the options to `strip` and exit.

`-I bfdname`

`--input-target=bfdname`

Treat the original *object-file* as a file with the object code format *bfdname*. See “Target Selection” on page 214 for more information.

`-O bfdname`

`--output-target=bfdname`

Replace *object-file* with a file in the output format, *bfdname*. See “Target Selection” on page 214 for more information.

`-R sectionname`

`--remove-section=sectionname`

Remove any section named *sectionname* from the output file. This option may be given more than once.

IMPORTANT! Using this option inappropriately may make the output file unusable.

`-s`

`--strip-all`

Remove all symbols.

-g
-S
--strip-debug
 Remove debugging symbols only.
--strip-unnneeded
 Remove all symbols that are not needed for relocation processing.
-K *symbolname*
--keep-symbol=*symbolname*
 Keep only symbol *symbolname* from the source file. This option may be given more than once.
-N *symbolname*
--strip-symbol=*symbolname*
 Remove symbol *symbolname* from the source file. This option may be given more than once, and may be combined with `strip` options other than `-k`.
-o *file*
 Put the stripped output in *file*, rather than replacing the existing file. When this argument is used, only one *object-file* argument may be specified.
-p
--preserve-dates
 Preserve the access and modification dates of the file.
-x
--discard-all
 Remove non-global symbols.
-X
--discard-locals
 Remove compiler-generated local symbols. (these usually start with `_L` or `._`).
-V
--version
 Show the version number for `strip`.
-v
--verbose
 Verbose output: list all object files modified. In the case of archives, `strip -v` lists all members of the archive.

`c++filt` Utility

```
c++filt [ -_ | --strip-underscores ]
        [ -n | --no-strip-underscores ]
        [ -sformat | --format=format ]
        [ --help ] [ --version ] [ symbol... ]
```

The C++ language provides function overloading, which means that you can write many functions with the same name (providing each takes parameters of different types). All C++ function names are encoded into a low-level assembly label (this process is known as *mangling*). The `c++filt` program does the inverse mapping: it decodes (*demangles*) low-level names into user-level names so that the linker can keep these overloaded functions from clashing.

Every alphanumeric word (consisting of letters, digits, underscores, dollars, or periods) seen in the input is a potential label. If the label decodes into a C++ name, the C++ name replaces the low-level name in the output.

You can use `c++filt` to decipher individual symbols, using a declaration like the following example.

```
c++filt symbol
```

If no *symbol* arguments are given, `c++filt` reads symbol names from the standard input and writes the demangled names to the standard output. All results are printed on the standard output.

`-_`

`--strip-underscores`

On some systems, both the C and C++ compilers put an underscore in front of every name. This option removes the initial underscore. Whether `c++filt` removes the underscore by default is target dependent.

`-n`

`--no-strip-underscores`

Do not remove the initial underscore.

`-s format`

`--format=format`

GNU `nm` can decode three different methods of mangling, used by different C++ compilers. The argument to this option selects which method it uses:

`gnu`

The one used by the GNU compiler (the default method).

`lucid`

The one used by the Lucid compiler.

`arm`

The one specified by the *C++ Annotated Reference Manual*.

--help

Print a summary of the options to `c++filt` and exit.

--version

Print the version number of `c++filt` and exit.

WARNING! `c++filt` is a developing utility, meaning that the details of its user interface are subject to change as C++ changes. In particular, a command-line option may be required in the future to decode a name passed as an argument on the command line; for example, `c++filt symbol` may in a future release become `c++filt option symbol`.

addr2line Utility

```
addr2line [ -b bfdname | --target=bfdname ]
          [ -C | --demangle ]
          [ -e filename | --exe=filename ]
          [ -f | --functions ] [ -s | --basename ]
          [ -H | --help ] [ -V | --version ]
          [ addr addr ... ]
```

`addr2line` translates program addresses into file names and line numbers. Given an address and an executable, it uses the debugging information in the executable to figure out which file name and line number are associated with a given address.

The executable to use is specified with the `-e` option. The default is `a.out`.

`addr2line` has two modes of operation.

In the first, hexadecimal addresses are specified on the command line, and `addr2line` displays the file name and line number for each address.

In the second, `addr2line` reads hexadecimal addresses from standard input, and prints the file name and line number for each address on standard output. In this mode, `addr2line` may be used in a pipe to convert dynamically chosen addresses.

The format of the output is `FILENAME:LINENO`. The file name and line number for each address is printed on a separate line. If the `-f` option is used, then each `FILENAME:LINENO` line is preceded by a `FUNCTIONNAME` line which is the name of the function containing the address.

If the file name or function name can not be determined, `addr2line` will print two question marks in their place. If the line number can not be determined, `addr2line` will print 0.

The long and short forms of options, shown here as alternatives, are equivalent.

`-b bfdname`

`--target=bfdname`

Specify that the object-code format for the object files is *bfdname*.

`-C`

`--demangle`

Decode (*demangle*) low-level symbol names into user-level names. Besides removing any initial underscore prepended by the system, this makes C++ function names readable. See “`c++filt` Utility” on page 196 for more information on demangling.

`-e filename`

`--exe=filename`

Specify the name of the executable for which addresses should be translated. The default file is `a.out`.

- f
- functions
Display function names as well as file and line number information.
- s
- basenames
Display only the base of each file name.

nlmconv Utility

`nlmconv` converts a relocatable object file into a NetWare Loadable Module.

WARNING! `nlmconv` is not always built as part of the binary utilities, since it is only useful for NLM targets.

```
nlmconv [ -I bfdname | --input-target=bfdname ]
        [ -O bfdname | --output-target=bfdname ]
        [ -T headerfile | --header-file=headerfile ]
        [ -d | --debug ] [ -l linker | --linker=linker ]
        [ -h | --help ] [ -V | --version ]
        input-file outfile
```

`nlmconv` converts the relocatable `i386` object file *input-file* into the NetWare Loadable Module, *outfile*, optionally reading *headerfile* for NLM header information.

For instructions on writing the NLM command file language used in header files, see “linkers” or “NLMLINK” in particular, in the *NLM Development and Tools Overview*, which is part of the *NLM Software Developer’s Kit* (“*NLM SDK*”), available from Novell, Inc.

`nlmconv` uses the GNU Binary File Descriptor library (BFD) to read *input-file*; see the documentation for “BFD Library” in *Using ld* in *GNU Pro Development Tools* for more information.

`nlmconv` can perform a link step. In other words, you can list more than one object file for input if you list them in the definitions file (rather than simply specifying one input file on the command line). In this case, `nlmconv` calls the linker for you.

`-I bfdname`

`--input-target=bfdname`

Object format of the input file. `nlmconv` can usually determine the format of a given file (so no default is necessary). See “Target Selection” on page 214 for more information.

`-O bfdname`

`--output-target=bfdname`

Object format of the output file. `nlmconv` infers the output format based on the input format, (for example, for a `i386` input file, the output format is `nlm32-i386`). See “Target Selection” on page 214 for more information.

`-T headerfile`

`--header-file=headerfile`

Reads *headerfile* for NLM header information. For instructions on writing the NLM command file language used in header files, see “linkers” or “NLMLINK”

in particular, of the *NLM Development and Tools Overview*, which is part of the *NLM Software Developer's Kit* (“*NLM SDK*”), available from Novell, Inc.

-d

--debug

Displays (on standard error) the linker command line used by `nlmconv`.

-l *linker*

--linker=*linker*

Use *linker* for any linking. *linker* can be an absolute or a relative pathname.

-h

--help

Prints a usage summary.

-v

--version

Prints the version number for `nlmconv`.

windres Utility

`windres` may be used to manipulate Windows resources.

WARNING! `windres` is not always built as part of the binary utilities, since it is only useful for Windows targets.

```
windres [ -i filename | --input filename ]
        [ -o filename | --output filename ]
        [ -I format | --input-format format ]
        [ -O format | --output-format format ]
        [ -F target | --target target ]
        [ --preprocessor program | --include-dir directory ]
        [ -define sym[=val] | --language val ]
        [ --help ] [ --version ] [--yydebug ]
        input-file output-file
```

`windres` reads resources from an input file and copies them into an output file. Either file may be in one of three formats:

`rc`

A text format read by the Resource Compiler.

`res`

A binary format generated by the Resource Compiler.

`coff`

A COFF object or executable.

The exact description of these different formats is available in documentation from Microsoft.

When `windres` converts from the `rc` format to the `res` format, it is acting like the Windows Resource Compiler. When `windres` converts from the `res` format to the `coff` format, it is acting like the Windows `CVTRES` program.

When `windres` generates an `rc` file, the output is similar but not identical to the format expected for the input. When an input `rc` file refers to an external filename, an output `rc` file will instead include the file contents.

If the input or output format is not specified, `windres` will guess based on the file name, or, for the input file, the file contents. A file with an extension of `.rc` will be treated as an `rc` file, a file with an extension of `.res` will be treated as a `res` file, and a file with an extension of `.o` or `.exe` will be treated as a `coff` format file.

If no output file is specified, `windres` will print the resources in `rc` format to standard output.

The normal use is for you to write an `rc` file, use `windres` to convert it to a COFF file, and then link the COFF file into your application.

This will make the resources described in the `rc` file available to Windows.

`-i filename`

`--input filename`

The name of the input file. If this option is not used, then `windres` will use the first non-option argument as the input file name. If there are no non-option arguments, then `windres` will read from standard input. `windres` can not read a COFF file from standard input.

`-o filename`

`--output filename`

The name of the output file. If this option is not used, then `windres` will use the first non-option argument, after any used for the input file name, as the output file name. If there is no non-option argument, then `windres` will write to standard output. `windres` can not write a COFF file to standard output.

`-I format`

`--input-format format`

The input format to read. *format* may be `res`, `rc`, or `coff`. If no input format is specified, `windres` will guess.

`-O format`

`--output-format format`

The output format to generate. *format* may be `res`, `rc`, or `coff`. If no output format is specified, `windres` will guess.

`-F target`

`--target target`

Specify the BFD format to use for a COFF file as input or output. This is a BFD target name; you can use the `--help` option to see a list of supported targets. Normally `windres` will use the default format, which is the first one listed by the `--help` option.

`--preprocessor program`

When `windres` reads an `rc` file, it runs it through the C preprocessor first. This option may be used to specify the preprocessor to use, including any leading arguments. The default preprocessor argument uses the following commands and arguments:

```
gcc -E -xc-header -DRC_INVOKED.
```

`--include-dir directory`

Specify an include directory to use when reading an `rc` file.

`windres` will pass this to the preprocessor as an `-I` option.

`windres` will also search this directory when looking for files named in the `rc` file.

`--define sym[=val]`

Specify a `-D` option to pass to the preprocessor when reading an `rc` file.

`--language val`

Specify the default language to use when reading an `rc` file.

`val` should be a hexadecimal language code. The low eight bits are the language, and the high eight bits are the sub-language.

`--help`

Prints a usage summary.

`--version`

Prints the version number for `windres`.

`--yydebug`

If `windres` is compiled with `YYDEBUG` defined as 1, this will turn on parser debugging.

dlltool Utility

`dlltool` may be used to create the files needed to build and use *dynamic link libraries* (DLLs).

WARNING! `dlltool` is not always built as part of the binary utilities, since it is only useful for those targets which support DLLs.

```
dlltool [-d|--input-def def-file-name]
        [-b|--base-file base-file-name]
        [-e|--output-exp exports-file-name]
        [-z|--output-def def-file-name]
        [-l|--output-lib library-file-name]
        [--export-all-symbols] [--no-export-all-symbols]
        [--exclude-symbols list]
        [--no-default-excludes]
        [-S|--as path-to-assembler] [-f|--as-flags options]
        [-D|--dllname name] [-m|--machine machine]
        [-a|--add-indirect] [-U|--add-underscore] [-k|--kill-at]
        [-A|--add-stdcall-alias]
        [-x|--no-idata4] [-c|--no-idata5] [-i|--interwork]
        [-n|--nodelete] [-v|--verbose] [-h|--help] [-V|--version]
        object-file...
```

`dlltool` reads its inputs, which can come from the `-d` and `-b` options as well as object files specified on the command line. It then processes these inputs and if the `-e` option has been specified it creates a exports file. If the `-l` option has been specified it creates a library file and if the `-z` option has been specified it creates a definition file (a `.def` file). Any or all of the `-e`, `-l` and `-z` options can be present in one invocation of `dlltool`.

When creating a DLL, along with the source for the DLL, it is necessary to have three other files. `dlltool` can help with the creation of these files.

The first file is a `.def` file which specifies the functions that are exported from the DLL, which functions the DLL imports, and so on. This is a text file and can be created by hand, or `dlltool` can be used to create it using the `-z` option. In this case, `dlltool` will scan the object files specified on its command line, looking for those functions which have been specially marked as being exported, putting entries for them in the `.def` file it creates.

In order to mark a function as being exported from a DLL, it needs to have an `-export:<name_of_function>` entry in the `.directive` section of the object file. This can be done in C by using the `asm()` operator, as in the following example.

```
asm (".section .directive");
asm (".ascii \"-export:my_func\"");
```

```
int my_func (void) { ... }
```

The second file needed for DLL creation is an exports file. This file is linked with the object files that make up the body of the DLL and it handles the interface between the DLL and the outside world. This is a binary file and it can be created by giving the `-e` option to `dlltool` when it is creating or reading in a `.def` file.

The third file needed for DLL creation is the library file that programs will link with in order to access the functions in the DLL. This file can be created by giving the `-l` option to `dlltool` when it is creating or reading in a `.def` file.

`dlltool` builds the library file by hand, but it builds the exports file by creating temporary files containing assembler statements and then assembling these. The `-s` command line option can be used to specify the path to the assembler that the `dlltool` utility will use, and the `-f` option can be used to pass specific flags to that assembler. The `-n` can be used to prevent `dlltool` from deleting these temporary assembler files when it is done, and if `-n` is specified twice then this will prevent `dlltool` from deleting the temporary object files it used to build the library.

The following example shows how to create a DLL from a source file `dll.c` and also creating a program (from an object file called `program.o`) that uses that DLL.

```
gcc -c dll.c
dlltool -e exports.o -l dll.lib dll.o
gcc dll.o exports.o -o dll.dll
gcc program.o dll.lib -o program
```

The command line options have the following meanings.

`-d filename`

`--input-def filename`

Specifies the name of a `.def` file to be read in and processed.

`-b filename`

`--base-file filename`

Specifies the name of a base file to be read in and processed. The contents of this file will be added to the relocation section in the exports file generated by `dlltool`.

`-e filename`

`--output-exp filename`

Specifies the name of the export file to be created by `dlltool`.

`-z filename`

`--output-def filename`

Specifies the name of the `.def` file to be created by `dlltool`.

`-l filename`

`--output-lib filename`

Specifies the name of the library file to be created by `dlltool`.

`--export-all-symbols`

Treat all global and weak defined symbols found in the input object files as

symbols to be exported. There is a small list of symbols which are not exported by default; see the `--no-default-excludes` option. You may add to the list of symbols to not export by using the `--exclude-symbols` option.

`--no-export-all-symbols`

Only export symbols explicitly listed in an input `.def` file or in `.directive` sections in the input object files. This is the default behaviour. The `.directive` sections are created by `dllexport` attributes in the source code.

`--exclude-symbols list`

Do not export the symbols in `list`. This is a list of symbol names separated by comma or colon characters. The symbol names should not contain a leading underscore. This is only meaningful when `--export-all-symbols` is used.

`--no-default-excludes`

When `--export-all-symbols` is used, it will by default avoid exporting certain special symbols. The current list of symbols to avoid exporting is `DllMain@12`, `DllEntryPoint@0`, `impure_ptr`. You may use the `--no-default-excludes` option to go ahead and export these special symbols. This is only meaningful when `--export-all-symbols` is used.

`-S path`

`--as path`

Specifies the path (`path`), including the filename, of the assembler to be used to create the exports file.

`-f switches`

`--as-flags switches`

Specifies any specific command line switches (`switches`) to be passed to the assembler when building the exports file. This option will work even if the `-s` option is not used. This option only takes one argument, and if it occurs more than once on the command line, then later occurrences will override earlier occurrences. So, if it is necessary to pass multiple switches to the assembler, they should be enclosed in double quotes.

`-D name`

`--dll-name name`

Specifies the name to be stored in the `.def` file as the name of the DLL when the `-e` option is used. If this option is not present, then the filename given to the `-e` option will be used as the name of the DLL.

`-m machine`

`-machine machine`

Specifies the type of machine for which the library file should be built. `dlltool` has a built in default type, depending upon how it was created, but this option can be used to override that initialization. This is normally only useful when creating DLLs for an ARM processor, when the contents of the DLL actually encode using THUMB instructions.

`-a`
`--add-indirect`
Specifies that, when `dlltool` is creating the exports file, `dlltool` should add a section which allows the exported functions to be referenced without using the import library.

`-U`
`--add-underscore`
Specifies that, when `dlltool` is creating the exports file, `dlltool` should prepend an underscore to the names of the exported functions.

`-k`
`--kill-at`
Specifies that when `dlltool` is creating the exports file it should not append the string, `@ <number>`. These numbers are called ordinal numbers and they represent another way of accessing the function in a DLL, other than by name.

`-A`
`--add-stdcall-alias`
Specifies that when `dlltool` is creating the exports file it should add aliases for stdcall symbols without `@ <number>` in addition to the symbols with `@ <number>`.

`-x`
`--no-idata4`
Specifies that when `dlltool` is creating the exports and library files it should omit the `.idata4` section. This is for compatibility with certain operating systems.

`-c`
`--no-idata5`
Specifies that when `dlltool` is creating the exports and library files it should omit the `.idata5` section. This is for compatibility with certain operating systems.

`-i`
`--interwork`
Specifies that `dlltool` should mark the objects in the library file and the exports file that it produces as supporting interworking between ARM and THUMB code.

`-n`
`--nodelete`
Makes `dlltool` preserve the temporary assembler files it used to create the exports file. If this option is repeated, the `dlltool` will also preserve the temporary object files it uses to create the library file.

`-v`
`--verbose`
Make `dlltool` describe what it is doing.

`-h`
`--help`
Displays a list of command line options and then exits.

-v

--version

Displays dlltool's version number and then exits.

readelf Utility

```
readelf [ -a | --all ]
        [ -b | --file-header ]
        [ -l | --program-headers | --segments ]
        [ -S | --section-headers | --sections ]
        [ -e | --headers ]
        [ -s | --syms | --symbols ]
        [ -r | --relocs ]
        [ -d | --dynamic ]
        [ -V | --version-info ]
        [ -D | --use-dynamic ]
        [ -x <number> | --hex-dump=<number> ]
        [ -w[liapr] | --debug-dump
          [=info,=line,=abbrev,=pubnames,=ranges ] [ --histogram ]
        [ -V | --version ]
        [ -H | --help ]
        elffile...
```

`readelf` displays information about one or more ELF format object files. The options control what particular information to display.

`elffile...` signifies the object files to be examined. `readelf` does not support examining archives, nor does it support examining 64 bit ELF files.

The long and short forms of options, shown here as alternatives, are equivalent (for instance, `-a` is the same as `--all`).

At least one option besides `-v` or `-H` must be given.

`-a`

`--all`

Equivalent to specifying `--file-header`, `--program-headers`, `--sections`, `--symbols`, `--relocs`, `--dynamic` and `--version-info`.

`-h`

`--file-header`

Displays the information contained in the ELF header at the start of the file.

`-l`

`--program-headers`

`--segments`

Displays the information contained in the file's segment headers, if it has any.

`-S`

`--sections`

`--section-headers`

Displays the information contained in the file's section headers, if it has any.

`-s`
`--symbols`
`--syms`
Displays the entries in symbol table section of the file, if it has one.

`-e`
`--headers`
Display all the headers in the file. Equivalent to `-h -l -s` input.

`-r`
`--relocs`
Displays the contents of the file's relocation section, if it has one.

`-d`
`--dynamic`
Displays the contents of the file's dynamic section, if it has one.

`-V`
`--version-info`
Displays the contents of the version sections in the file, if they exist.

`-D`
`--use-dynamic`
When displaying symbols, this option makes `readelf` use the symbol table in the file's dynamic section, rather than the one in the symbols section.

`-x <number>`
`--hex-dump=<number>`
Displays the contents of the indicated section (designated `<number>`) as a hexadecimal dump.

`-w[liapr]`
`--debug-dump[=line,=info,=abbrev,=pubnames,=ranges]`
Displays the contents of the debug sections in the file, if any are present. If one of the optional letters or words follows the switch then only data found in those specific sections will be dumped.

`--histogram`
Display a histogram of bucket list lengths when displaying the contents of the symbol tables.

`-v`
`--version`
Display the version number of `readelf`.

`-H`
`--help`
Display the command line options understood by `readelf`.

2

Selecting the Target System

You can specify three aspects of the target system to the GNU binary file utilities, selecting each in several ways:

- as a *target*; see “Target Selection” on page 214 for discussions of the lists of ways to specify values for each specification
- as an *architecture*; see “Architecture Selection” on page 216 for the ways to manage the binary utilities on different processors
- as a *linker emulation*, a personality of the linker, giving the linker default values applying only to the linker; see “Linker Emulation Selection” on page 217

In the following summaries, the lists of ways to specify values are in order of decreasing precedence; the ways listed first override those listed in precedence.

The commands to list valid values only list the values for which the programs you are running were configured. If they were configured with `--enable-targets=all`, the commands list most of the available values, although a few are missing; not all targets can be configured in at once because some of them can only be configured native (on hosts with the same type as the target system).

See “Overview of binutils, the GNU Binary Utilities” on page 167 for locating more documentation on the individual utilities.

Target Selection

A *target* is an object file format. A given target may be supported for multiple architectures (see “Architecture Selection” on page 216). A target selection may also have variations for different operating systems or architectures. The commands to list valid values only list the values for which the programs you are running were configured. If they were configured with `--enable-targets=all`, the commands list most of the available values, but a few are left out; not all targets can be configured in at once because some of them can only be configured *native* (on hosts with the same type as the target system). The command to list valid target values is `objdump -i` (the first column of output contains the relevant information). Some sample values are: `a.out-hp300bsd`, `ecoff-littlemips`, `a.out-sunos-big`. You can also specify a target using a *configuration triplet*. This is the same sort of name that is passed to `configure` to specify a target. When you use a configuration triplet as an argument, it must be fully *canonicalized*. You can see the canonical version of a triplet by running the shell script, `config.sub`, which is included with the sources. Some sample configuration triplets are `m68k-hp-bsd`, `mips-dec-ultrix`, and `sparc-sun-sunos`.

objdump Target

Ways to specify:

- command line option: `-b` or `--target`
- environment variable `GNUTARGET`
- deduced from the input file

objcopy and strip Input Target

Ways to specify:

- command line options, `-I`, `--input-target`, `-F`, or `--target`
- environment variable, `GNUTARGET`
- deduced from the input file

objcopy and strip Output Target

Ways to specify:

- command line options, `-O`, `--output-target`, `-F`, or `--target`
- the input target (see “objcopy and strip Input Target” on page 214)
- environment variable, `GNUTARGET`
- deduced from the input file

nm, size, and strings Target

Ways to specify:

- command line option, `--target`
- environment variable `GNUTARGET`
- deduced from the input file

Linker Input Target

Ways to specify:

- command line option, `-b` or `--format` (see “Using ld Command Line Options” in *Using ld* in *GNUPro Development Tools*)
- script command, `TARGET` (see “Commands Dealing with Object File Formats” in *Using ld* in *GNUPro Development Tools*)
- environment variable, `GNUTARGET` (see “ld Environment Variables” on page 26 in *Using ld* in *GNUPro Development Tools*)
- the default target of the selected linker emulation (see “Linker Emulation Selection” on page 217).

Linker Output Target

Ways to specify:

- command line option, `-oformat` (see “Using ld Command Line Options” in *Using ld* in *GNUPro Development Tools*)
- script command, `OUTPUT_FORMAT` (see “Commands Dealing with Object File Formats” in *Using ld* in *GNUPro Development Tools*)
- the linker input target; see “Linker Input Target” (above)

Architecture Selection

An *architecture* is a type of CPU on which an object file is to run. Its name may contain a colon, separating the name of the processor family from the name of the particular processor. The command to list valid architecture values is `objdump -i` (the second column contains the relevant information).

objdump Architecture

Ways to specify: `objdump`

- command line option: `-m` or `--architecture`
- deduced from the input file

objcopy, nm, size, strings Architecture

Its specification is deduced from the input file.

Linker input Architecture

Its specification is deduced from the input file.

Linker output Architecture

Ways to specify:

- script command, `OUTPUT_ARCH` (see “Other Linker Script Commands” in *Using ld* in *GNUPro Development Tools*)
- the default architecture from the linker output target (see “Target Selection” on page 214)

Linker Emulation Selection

A *linker emulation* is a personality of the linker, giving the linker default values for the other aspects of the target system. In particular, it consists of the linker script, the target and several *hook* functions (which are run at certain stages of the linking process to do special things that some targets require). The command to list valid linker emulation values is `ld -v`. Sample values: `hp300bsd`, `mips1it`, `sun4`.

Ways to specify:

- command line option, `-m` (see “Using ld Command Line Options” in *Using ld in GNUPro Development Tools*)
- environment variable, `LDEMULATION`
- compiled-in `DEFAULT_EMULATION` from `Makefile`, which comes from `EMUL` in `config/target.mt`

Using Cygwin

1

Windows Development with Cygwin: a Win32 Porting Layer

Cygwin™, a full-featured Win32 porting layer for UNIX applications, is compatible with all Win32 hosts (currently, these are Microsoft’s Windows NT, Windows 95, or Windows 98 systems). The following documentation discusses porting the GNU development tools to the Win32 host while exploring the development and architecture of the Cygwin library.

- “Porting UNIX Tools to Win32” on page 222
- “Goals of Cygwin” on page 223
- “Compatibility Issues with Cygwin” on page 233

See also “Setting up Cygwin” on page 237, “Using GCC with Cygwin” on page 255, “Debugging Cygwin Programs” on page 256, “Building and Using DLLs with Cygwin” on page 257, “Defining Microsoft Windows Resources for Cygwin” on page 258, “Cygwin Utilities” on page 262, and “Cygwin Functions” on page 276.

Cygwin was invented in 1995 as part of the answer to the question of how to port the GNU development tools to a Win32 host. The Win32-hosted GNUPro compiler tools that use the Cygwin library are available for a variety of embedded processors as well as a native version for writing Win32 applications. By basing this technology on the GNU tools, Cygwin provides you with a high-performance, feature-rich 32-bit code development environment, including a graphical source-level debugger, Insight™ (see “Insight, the GNUPro Debugger GUI” on page 169 in *GNUPro Debugger Tools*).

Cygwin is a dynamically-linked library (DLL) that provides a large subset of the system calls in common UNIX implementations. The current release includes all POSIX.1/90 calls except for `setuid` and `mkfifo`, all ANSI C standard calls, and many common BSD and SVR4 services (including Berkeley sockets). See also “Compatibility Issues with Cygwin” on page 233.

When the Free Software Foundation (FSF) first wrote the GNU tools in the mid-1980s, portability among existing and future UNIX operating systems was an important goal. By mid-1995, the tools had been ported to 16-bit DOS using the GO32 32-bit extender by DJ Delorie*. However, no one had completed a native 32-bit port for Windows NT, Windows 95, or Windows 98. It seemed likely that the demand for Win32-hosted native and cross-development tools would soon be large enough to justify the development costs involved.

This project’s fulfillment and its ongoing challenges are testaments to the growth that Cygwin provides; for the individuals who have been responsible for creating the Cygwin porting layer, see <http://cygwin.com/who.html>.

Porting UNIX Tools to Win32

The first step in porting compiler tools to Win32 was to enhance them so that they could generate and interpret Win32 native object files, using Microsoft’s Portable Executable (PE) format. This proved to be relatively straightforward because of similarities to the Common Object File Format (COFF), which the GNU tools already supported. Most of these changes were confined to the Binary File Descriptor (BFD) library and to the linker.

In order to support the Win32 Application Programming Interface (API), there is an extension of the capabilities of the binary utilities to handle Dynamic-Linked Libraries (DLLs). After creating export lists for the specific Win32 API DLLs that are shipped with Win32 hosts, the tools were able to generate static libraries that executables could use to gain access to Win32 API functions. Because of redistribution restrictions on Microsoft’s Win32 API header files, there are Win32 header files written from scratch (on an as-needed basis). Once this work was completed, it was possible to build UNIX-hosted cross-compilers capable of generating valid PE executables that ran on Win32 systems. See also “Using GCC with Cygwin” on page 255.

The next task was to port the compiler tools themselves to Win32. Previous experiences using Microsoft Visual C++ to port GDB convinced us to find another means for bootstrapping the full set of tools. In addition to wanting to use the GNU

* DJ Delorie, maintainer of the DJGPP Project (see <http://www.delorie.com/djgpp/>).

compiler technology, there is a desire for a portable build system. The GNU development tools' configuration and build procedures require a large number of additional UNIX utilities not available on Win32 hosts. So a decision was made to use UNIX-hosted cross-compilers to build Win32-hosted native and cross-development tools. It made perfect sense to do this since there were successes using a nearly identical technique to build DOS-hosted products.

The next obstacle to overcome was the many dependencies on UNIX system calls in the sources, especially in the GNU debugger GDB. While sizable portions could have been rewritten for the source code to work within the context of the Win32 API (as was done for the DOS-hosted tools), this would have been prohibitively time-consuming. Worse, it would have introduced conditionalized code that would have been expensive to maintain in the long run. Instead, developers took a substantially different approach by writing Cygwin. See also “Debugging Cygwin Programs” on page 256.

Goals of Cygwin

The following documentation discusses the work in developing the Cygwin tools.

- “Harnessing the Power of the Web for Cygwin” on page 224
- “The Cygwin Architecture” on page 225
- “Files and Filetypes for Cygwin” on page 226
- “Text Mode and Binary Mode Interoperability with Cygwin” on page 227
- “ANSI C Library for Cygwin” on page 228
- “Process Creation for Cygwin” on page 228
- “Signals with Cygwin” on page 229
- “Sockets with Cygwin” on page 230
- “The select Function with Cygwin” on page 230
- “Performance Issues with Cygwin” on page 230
- “Ported Software with Cygwin” on page 231
- “Future Work for Cygwin” on page 232

The original goal of Cygwin was simply to get the development tools working. Completeness with respect to POSIX.12 and other relevant UNIX standards was not a priority. Part of a definition of “working native tools” is having a build environment similar enough to UNIX to support rebuilding the tools themselves on the host system, a process called *self-hosting*. The typical configuration procedure for a GNU tool involves running `configure`, a complex Bourne shell script that determines information about the host system. The script then uses that information to generate

the Makefiles used to build the tool on the host in question. This configuration mechanism is needed under UNIX because of the large number of varying versions of UNIX. If Microsoft continues to produce new variants of the Win32 API as it releases new versions of its operating systems, it may prove to be quite valuable on the Win32 host as well. The need to support this configuration procedure added the requirement of supporting user tools such as `sh`, `make`, file utilities (such as `ls` and `rm`), text utilities (such as `cat`, `tr`), and shell utilities (such as `echo`, `date`, `uname`, `sed`, `awk`, `find`, `xargs`, `tar`, and `gzip`, among many others). Previously, most of these user tools had only been built natively (on the host on which they would run). As a result, `configure` scripts had to be modified to be compatible with cross-compilation.

Other than making the necessary configuration changes, it became necessary to avoid Win32-specific changes since the UNIX compatibility was to be provided by Cygwin as much as possible. While this would be a sizable amount of work, there was more to gain than just achieving self-hosting of the tools. Supporting the configuration of the development tools would also provide an excellent method of testing the Cygwin library.

Although it was possible to build working Win32-hosted toolchains with cross-compilers relatively soon after the birth of Cygwin, it took much longer than before the tools could reliably rebuild themselves on the Win32 host because of the many complexities involved.

Harnessing the Power of the Web for Cygwin

Instead of keeping the Cygwin technology proprietary and developing it in-house, it is publicly available under the terms of the GNU General Public License (GPL), the traditional license for the GNU tools. Since its inception, there is a new Cygwin release available using `ftp` over the Internet every three or four months. Each release includes binaries of Cygwin and the development tools, coupled with the source code needed to build them. These free releases come without any assurances of quality or support, although there is a mailing list that is used for discussion and feedback.

In retrospect, making the technology freely available was a good decision because of the high demand for quality 32-bit native tools in the Win32 arena, as well as significant additional interest in a UNIX portability layer like Cygwin. While far from perfect, the beta releases are good enough for many people. They provide us with tens of thousands of interested developers who are willing to use and test the tools. A few of them are even willing to contribute code fixes and new functionality to the library. As of the last public release, developers on the Net had written or improved a significant portion of the library, including important aspects such as support for UNIX signals and the TTY/PTY calls.

In order to spur as much Net participation as possible, the Cygwin project features an

open development model. There are source snapshots available to the general public using CVS, in addition to the periodic full Cygwin releases. A mailing list for developers facilitates discussion of proposed changes to the library.

In addition to the GPL version of Cygwin, there is a commercial license for supported customers of the native Win32 GNUPro tools.

The Cygwin Architecture

The following documentation discusses the architecture underlying the Cygwin tools.

- “Files and Filetypes for Cygwin” on page 226
- “Text Mode and Binary Mode Interoperability with Cygwin” on page 227
- “ANSI C Library for Cygwin” on page 228

When a binary linked against the library is executed, the Cygwin DLL is loaded into the application’s text segment. Because Cygwin is trying to emulate a UNIX kernel that needs access to all processes running under it, the first Cygwin DLL to run creates shared memory areas that other processes using separate instances of the DLL can access. This is used to keep track of open file descriptors and assist `fork` and `exec`, among other purposes. In addition to the shared memory regions, every process also has a per-process structure that contains information such as process ID, user ID, signal masks, and other similar process-specific information.

The DLL is implemented using the Win32 API, allowing it to run on all Win32 hosts. Because processes run under the standard Win32 subsystem, they can access both the UNIX compatibility calls provided by Cygwin as well as any of the Win32 API calls. This gives the programmer complete flexibility in designing the structure of their program in terms of the APIs used. For example, a project might require a Win32-specific GUI using Win32 API calls on top of a UNIX back-end that uses Cygwin.

Early on in the development process, an important design decision was made to overcome the necessity to strictly adhere to existing UNIX standards like POSIX[†], if it was not possible or if it would significantly diminish the usability of the tools on the Win32 platform. In many cases, an environment variable can be set to override the default behavior and force standards compliance.

While Windows 95 and Windows 98 are similar enough to each other that developers can safely ignore the distinction when implementing Cygwin, Windows NT is an extremely different operating system. For this reason, whenever the DLL is loaded, the library checks which operating system is active so that it can act accordingly. In

[†] ISO/IEC 9945-1:1996 (ANSI/IEEE Std 1003.1, 1996 Edition); *POSIX Part 1: System Application Program Interface (API)* [C Language].

some cases, the Win32 API is only different for historical reasons. In this situation, the same basic functionality is available under Windows 95, Windows 98, and Windows NT, although the method used to gain this functionality differs. A trivial example is in the implementation of `uname`, the library examines the `sysinfo.wProcessorLevel` structure member to determine the processor type used for Windows 95, Windows 98, and Windows NT. This field is not supported in Windows NT, which has its own operating system-specific structure member called `sysinfo.wProcessorLevel`.

Other differences between Windows 95, Windows 98, and Windows NT are much more fundamental in nature. The best example is that only Windows NT provides a security model. Windows NT includes a sophisticated security model based on Access Control Lists (ACLs). Although some modern UNIX operating systems include support for ACLs, Cygwin maps Win32 file ownership and permissions to the more standard, older UNIX model. The `chmod` call maps UNIX-style permissions back to the Win32 equivalents. Because many programs expect to be able to find the `/etc/passwd` and `/etc/group` files, there are utilities that can be used to construct them from the user and group information provided by the operating system.

With Windows NT, the administrator is permitted to `chown` files. There is currently no mechanism to support the `setuid` concept or API call. In practice, the programs that have ported have not needed it.

With Windows 95 and Windows 98, the situation is considerably different. Since a security model is not provided, Cygwin fakes file ownership by making all files look like they are owned by a default user and group ID. As with Windows NT, file permissions can still be determined by examining their read/write/execute status. Rather than return an unimplemented error, with Windows 95 and Windows 98, the `chown` call succeeds immediately without actually performing any action. This is appropriate since essentially all users jointly own the files when no concept of file ownership exists.

It is important to discuss the implications of the Cygwin *kernel*, using shared memory areas to store information about Cygwin processes. Because these areas are not yet protected in any way, a malicious user could perhaps modify them to cause unexpected behavior in Cygwin processes. While this is not a new problem under Windows 95 and Windows 98 (because of the lack of operating system security), it does constitute a security hole under Windows NT. This is because one user could affect the Cygwin programs run by another user by changing the shared memory information in ways that they could not in a more typical Windows NT program. For this reason, it is not appropriate to use Cygwin in high-security applications. In practice, this will not be a major problem for most uses of the library.

Files and Filetypes for Cygwin

Cygwin supports both Win32- and POSIX-style paths, using either forward or back

slashes as the directory delimiter. Paths coming into the DLL are translated from Win32 to POSIX as needed. As a result, the library believes that the file system is a POSIX-compliant one, translating paths back to Win32 paths whenever it calls a Win32 API function. UNC pathnames (*Universal Naming Conventions*, which are paths that start with two slashes) are supported. See also “Cygwin’s Compatibility with POSIX.1 Standards” on page 234.

The layout of this POSIX view of the Windows file system space is stored in the Windows registry. While the slash (‘/’) directory points to the system partition by default, this is easy to change with the Cygwin mount utility. In addition to selecting the slash partition, it allows mounting arbitrary Win32 paths into the POSIX file system space. Many people use the utility to mount each drive letter under the slash partition (that is, **C:**\ to /c, **D:**\ to /d, and so forth).

The library exports several Cygwin-specific functions that can be used by external programs to convert a path or path list from Win32 to POSIX or vice versa. Shell scripts and Makefiles cannot call these functions directly. Instead, they can do the same path translations by executing the “`cygpath`” utility.

Win32 file systems are case preserving but case insensitive. Cygwin does not currently support case distinction because, in practice, few UNIX programs actually rely on it. While it could be possible to mangle file names to support case distinction, this would add unnecessary overhead to the library and make it more difficult for non-Cygwin applications to access those files.

Symbolic links are emulated by files containing a magic cookie followed by the path to which the link points. They are marked with the **System** attribute so that only files with that attribute have to be read to determine whether or not the file is a symbolic link. Hard links are fully supported under Windows NT on NTFS file systems. On a FAT file system, the call falls back to copying the file, a strategy that works in many cases.

The `inode` number for a file is calculated by hashing its full Win32 path. The `inode` number generated by the `stat` call always matches the one returned in `d_ino` of the `dirent` structure. It is worth noting that the number produced by this method is not guaranteed to be unique. However, this is not a significant problem because of the low probability of generating a duplicate `inode` number.

Text Mode and Binary Mode Interoperability with Cygwin

Interoperability with other Win32 programs such as text editors was critical to the success of the port of the development tools. Most customers upgrading from the older DOS-hosted toolchains expected the new Win32-hosted ones to continue to work with their old development sources.

Since UNIX and Win32 use different end-of-line terminators in text files,

consequently, carriage-return newlines have to be translated by Cygwin into a single newline when reading in text mode. The **Ctrl+z** character is interpreted as a valid end-of-file character for a similar reason.

This solution addresses the compatibility requirement at the expense of violating the POSIX standard that states that text and binary mode will be identical. Consequently, processes that attempt to lseek through text files can no longer rely on the number of bytes read as an accurate indicator of position in the file. For this reason, an environment variable can be set to override this behavior. See also “Cygwin’s Compatibility with POSIX.1 Standards” on page 234, “Environment Variables for Cygwin” on page 247 and “Text and Binary Modes” on page 250.

ANSI C Library for Cygwin

The package for Cygwin includes an existing ANSI C[‡] library, `newlib`, as part of the library, rather than write all of the GNU C libraries and math calls from scratch. `newlib` is a BSD-derived ANSI C library, previously only used by cross-compilers for embedded systems development. The reuse of existing free implementations of such things as the `glob`, `regex`, and `getopt` libraries saved considerable effort. In addition, Cygwin uses Doug Lea’s free `malloc` implementation that successfully balances speed and compactness. The library accesses the `malloc` calls, using an exported function pointer. This makes it possible for a Cygwin process to provide its own `malloc` if required. For more information, see “Cygwin’s Compatibility with ANSI Standards” on page 233.

Process Creation for Cygwin

The following documentation discusses the process in the API with Cygwin tools.

- “Signals with Cygwin” on page 229
- “Sockets with Cygwin” on page 230
- “The select Function with Cygwin” on page 230
- “Performance Issues with Cygwin” on page 230
- “Ported Software with Cygwin” on page 231

The `fork` call in Cygwin is particularly interesting because it does not map well on top of the Win32 API. This makes it very difficult to implement. Currently, the Cygwin `fork` is a non-copy-on-write implementation similar to what was present in early versions of UNIX.

The first thing that happens when a parent process forks a child process is that the parent initializes a space in the Cygwin process table for the child. It then creates a

[‡] ISO/IEC 9899:1990, *Programming Language (C)*.

suspended child process using the Win32 **CreateProcess** call. Next, the parent process calls `setjmp` to save its own context and sets a pointer to this in a Cygwin shared memory area (shared among all Cygwin tasks). It then fills in the child's `.data` and `.bss` sections by copying from its own address space into the suspended child's address space. After the child's address space is initialized, the child is run while the parent waits on a `mutex`. The child discovers it has been forked and longjumps using the saved jump buffer. The child then sets the `mutex` the parent is waiting on and blocks on another `mutex`. This is the signal for the parent to copy its stack and heap into the child, after which it releases the `mutex` the child is waiting on and returns from the `fork` call. Finally, the child wakes from blocking on the last `mutex`, recreates any memory-mapped areas passed to it from the shared area, and returns from **fork** itself. While there are ideas as to how to speed up `fork` implementation by reducing the number of context switches between the parent and child process, **fork** will almost certainly always be inefficient under Win32. Fortunately, in most circumstances, the `spawn` family of calls provided by Cygwin can be substituted for a `fork/exec` pair with only a little effort. These calls map cleanly on top of the Win32 API. As a result, they are much more efficient. Changing the compiler's driver program to call `spawn` instead of `fork` was a trivial change and increased compilation speeds by 20-30% in tests.

However, `spawn` and `exec` present their own set of difficulties. Because there is no way to do an actual `exec` under Win32, Cygwin has to invent its own Process IDs (PIDs). As a result, when a process performs multiple `exec` calls, there will be multiple Windows PIDs associated with a single Cygwin PID. In some cases, stubs of each of these Win32 processes may linger, waiting for their Cygwin process to exit.

Signals with Cygwin

When a Cygwin process starts, the library starts a secondary thread for use in signal handling. This thread waits for Windows events used to pass signals to the process. When a process notices it has a signal, it scans its signal bitmask and handles the signal in the appropriate fashion.

Several complications in the implementation arise from the fact that the signal handler operates in the same address space as the executing program. The immediate consequence is that Cygwin system functions are interruptible unless special care is taken to avoid them. Measures have been used to prevent the `sig_send` function that sends signals from being interrupted. In the case of a process sending a signal to another process, there is a `mutex` around `sig_send` such that `sig_send` will not be interrupted until it has completely finished sending the signal.

In the case of a process sending itself a signal, there is a separate `semaphore/event` pair instead of the `mutex`. `sig_send` starts by resetting the event and incrementing the semaphore that flags the signal handler to process the signal. After the signal is

processed, the signal handler signals the event that it is done. This process keeps intraprocess signals synchronous, as required by POSIX. Most standard UNIX signals are provided. Job control works as expected in shells that support it.

Sockets with Cygwin

Socket-related calls in Cygwin simply call the functions by the same name in Winsock, Microsoft's implementation of Berkeley sockets. Only a few changes were needed to match the expected UNIX semantics; one of the most troublesome differences was that Winsock must be initialized before the first socket function is called. As a result, Cygwin has to perform this initialization when appropriate. In order to support sockets across fork calls, child processes initialize Winsock if any inherited file descriptor is a socket.

Unfortunately, implicitly loading DLLs at process startup is usually a slow affair. Because many processes do not use sockets, Cygwin explicitly loads the Winsock DLL the first time it calls the Winsock initialization routine. This single change sped up GNU `configure` times by 30%.

The `select` Function with Cygwin

The UNIX `select` function is another call that does not map cleanly on top of the Win32 API. The Win32 `select` in Winsock only worked on socket handles. A new implementation allows `select` to function normally when given different types of file descriptors (such as sockets, pipes, handles, and a custom `/dev/windows` windows messages pseudo-device).

Upon entry into the `select` function, the first operation is to sort the file descriptors into the different types. There are then two cases to consider.

- The simple case is when at least one file descriptor is a type that is always known to be ready (such as a disk file). In that case, `select` returns immediately as soon as it has polled each of the other types to see if they are ready.
- The more complex case involves waiting for socket or pipe file descriptors to be ready. This is accomplished by the main thread suspending itself, after starting one thread for each type of file descriptor present. Each thread polls the file descriptors of its respective type with the appropriate Win32 API call. As soon as a thread identifies a ready descriptor, that thread signals the main thread to wake up. This case is now the same as the first one since there is at least one descriptor ready. So `select` returns, after polling all of the file descriptors one last time.

Performance Issues with Cygwin

Early on in the development process, correctness was almost the entire emphasis and, as Cygwin became more complete, performance became a much important issue. It

was known that the tools ran much more slowly under Win32 than under Linux on the same machine, but it was not clear at all whether to attribute this to differences in the operating systems or to inefficiencies in Cygwin.

The lack of a working profiler has made analyzing Cygwin's performance particularly difficult. Although the latest version of the library includes real **itimer** support, there is no current way to implement a virtual **itimer**. This is the most reliable way of obtaining profiling data since concurrently running processes aren't likely to skew the results. There will soon be available a combination of the GCC compiler and the GNU profile analysis tool, `gprof`, working with real **itimer** support which will help a great deal in optimizing Cygwin.

Even without a profiler, there are several areas inside Cygwin that definitely need a fresh approach. While those sections of code were rewritten, there is the speed of configuring the tools under Win32 as the primary performance measurement. This choice made sense because there is process creation speed, which was especially poor, something that the GNU `configure` process stresses.

These performance adjustments made it possible to configure completely the development tools under NT with Cygwin in only 10 minutes and complete the build in just under an hour on a dual Pentium Pro 200 system with 128 MB of RAM. This is reasonably competitive with the time taken to complete this task under a typical UNIX operating system running on an identical machine.

Ported Software with Cygwin

In addition to being able to configure and build most GNU software, several other significant packages have been successfully ported to the Win32 host using the Cygwin library. Following is a list of some of the more interesting ones (most are not included in the distributions):

- X11R6 client libraries, enabling porting many X programs to the existing free Win32 X servers (examples of successfully ported X applications include `xterm`, `ghostview`, `xfig`, and `xconq`)
- `xemacs` and `vim` editors
- GNU `inetutils` in order to run the `inetd` daemon as a Windows NT service to enable UNIX-style networking, using a custom NT login binary to allow remote logins with full user authentication; one can achieve similar results under Windows 95 and Windows 98 by running `inetd` out of the `autoexec.bat` file, providing a custom tailored login binary for Windows 95 and Windows 98
- KerbNet, the implementation of the Kerberos security system
- CVS (Concurrent Versions System), a popular version control program based on RCS; there is also a Kerberos-enabled version of CVS to grant secure access to GNU source code for local and remote engineers

- `ncurses`, a library that can be used to build a functioning version of the pager
- `ssh` (secure shell) client and server
- PERL 5 scripting language
- `bash`, `tcsh`, `ash`, and `zsh` shells; full job control is available in supported shells
- Apache web server (some source-level changes were necessary)
- Tcl/Tk 8; also `tix`, `itcl`, and `expect` (Tcl/Tk needed non-trivial configuration changes)

Typically, the only necessary source code modification involves specifying binary mode to open calls as appropriate. Because the Win32 compiler always generates executables that end in the standard `.exe` suffix, it is also often necessary to make minor modifications to makefiles so that `make` will expect the newly built executables to end with the suffix.

Future Work for Cygwin

Standards conformance is becoming a more important focus. Previous work includes getting all POSIX.1/90 calls implemented; except for `mkfifo` and `setuid`, they have been. X/Open Release 4** conformance may be a desirable goal, but it is not yet implemented. While the current version of the library passes most of the NIST POSIX test suite††, it performs poorly with respect to mimicking the UNIX security model, so there is still room for improvement. When considering how to implement the `setuid` functionality, there must be a secure alternative to the library's usage of the shared memory areas.

Cygwin does not yet support applications that use multiple Windows threads, even though the library itself is multi-threaded. Overcoming this shortcoming through the use of locks at strategic points in the DLL is desired, as well as creating support for POSIX threads.

Although Cygwin allows the GNU development tools that depend heavily on UNIX semantics to run successfully on Win32 hosts, it is not always desirable to use it. A program using a perfect implementation of the library would still incur a noticeable amount of overhead. As a result, an important future direction involves modifying the compiler so that it can optionally link against the Microsoft DLLs that ship with both Win32 operating systems, instead of Cygwin. This will give developers the ability to choose whether or not to use Cygwin on a per-program basis.

The lack of source code, coupled with the licensing fees associated with each of these

** The X/Open Release 4 CAE Specification, System Interfaces and Headers, Issue 4, Vol. 2, X/Open Co, Ltd., 1994.

†† NIST POSIX test suite (see http://www.itl.nist.gov/div897/ctg/posix_form.htm).

commercial offerings, might still have required writing a library if there was the same challenge of porting.

Compatibility Issues with Cygwin

The following documentation discusses the compatibility issues with Cygwin porting layer tools and the Cygwin library and its functionality.

- Cygwin’s Compatibility with ANSI Standards (below)
- “Cygwin’s Compatibility with POSIX.1 Standards” on page 234
- “Cygwin’s Compatibility with Other Miscellaneous Standards” on page 235

Cygwin’s Compatibility with ANSI Standards

The following functions are compatible with ANSI standards.

- **stdio functions**
clearerr, fclose, feof, ferror, fflush, fgetc, fgetpos, fgets, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, fsetpos, ftell, fwrite,getc, getchar, gets, perror, printf, putc, putchar, puts, remove, rename, rewind, scanf, setbuf, setvbuf, sprintf, sscanf, tmpfile, tmpnam, vfprintf, ungetc, vprintf, vsprintf.
- **string functions**
memchr, memcmp, memcpy, memmove, memset, strcat, strchr, strcmp, strcoll, strcpy, strcspn, strerror, strlen, strncat, strncmp, strncpy, strpbrk, strrchr, strspn, strstr, strtok, strxfrm.
- **stdlib functions**
abort, abs, assert, atexit, atof, atoi, atol, bsearch, calloc, div, exit, free, getenv, labs, ldiv, longjmp, malloc, mblen, mbstowcs, mbtowc, qsort, rand, realloc, setjmp, srand, strtod, strtol, strtoul, system, wctombs, wctomb.
- **time functions**
asctime, gmtime, localtime, time, clock, ctime, difftime, mktime, strftime.
- **signals functions**
raise, signal.
- **ctype functions**
isalnum, isalpha, iscntrl, isdigit, isgraph, islower, isprint, ispunct, isspace, isupper, isxdigit, tolower, toupper.

- **math functions**
acos, asin, atan, atan2, ceil, cos, cosh, exp, fabs, floor, fmod, frexp, ldexp, log, log10, modf, pow, sin, sinh, sqrt, tan, tanh.
- **Miscellaneous functions**
localeconv, setlocale, va_arg, va_end, va_start.

Cygwin's Compatibility with POSIX.1 Standards

The following functions are compatible with POSIX.1.

- **Process primitives**
fork, execl, execlp, execlpe, execv, execve, execvp, wait, waitpid, _exit, kill, sigemptyset, sigfillset, sigaddset, sigdelset, sigismember, sigaction, pthread_sigmask, sigprocmask, sigpending, sigsuspend, alarm, pause, sleep.
- **Process environment**
getpid, getppid, getuid, geteuid, getgid, getegid, setuid, setgid, getgroups, getlogin, getpgrp, setsid, setpgid, uname, time, times, getenv, ctermid, ttyname, isatty, sysconf.
- **Files and directories**
opendir, readdir, rewinddir, closedir, chdir, getcwd, open, creat, umask, link, mkdir, unlink, rmdir, rename, stat, fstat, access, chmod, fchmod, chown, utime, ftruncate, pathconf, fpathconf.
- **Input and output primitives**
pipe, dup, dup2, close, read, write, fcntl, lseek, fsync.
- **Device-specific and class-specific functions**
cfgetispeed, cfgetospeed, cfsetispeed, cfsetospeed, tcdrain, tcflow, tcflush, tcgetattr, tcgetpgrp, tcsendbreak, tcsetattr, tcsetpgrp.
- **Language-specific services for the C programming language**
abort, exit, fclose, fdopen, fflush, fgetc, fgets, fileno, fopen, fprintf, fputc, fputs, fread, freopen, fscanf, fseek, ftell, fwrite, getc, getchar, gets, perror, printf, putc, putchar, puts, remove, rewind, scanf, setlocale, siglongjmp, sigsetjmp, tmpfile, tmpnam, tzset.
- **Synchronization functions**
pthread_mutex_destroy, pthread_mutex_init, pthread_mutex_lock, pthread_mutex_trylock, pthread_mutex_unlock, sem_destroy, sem_init, sem_post, sem_trywait, sem_wait
- **System databases**
getgrgid, getgrnam, getpwnam, getpwuid.

- **Memory management**
mmap, mprotect, msync, munmap.
- **Thread management calls**
pthread_attr_getstacksize, pthread_attr_init,
pthread_attr_setstacksize, pthread_create, pthread_equal,
pthread_exit, pthread_self
- **Thread-specific data functions**
pthread_getspecific, pthread_key_create, pthread_key_delete,
pthread_setspecific

setuid and setgid are stubs that set ENOSYS and return 0.

link will copy the file if it can't implement a true symbolic linkcopy file in Win 95, and when link fails in Windows NT.

chown is a stub in Win 95, always returning 0.

fcntl doesn't support F_GETLK; it returns -1 and sets errno to ENOSYS.

lseek only works properly on binary files.

Cygwin's Compatibility with Other Miscellaneous Standards

The following functions are compatible with other miscellaneous standards.

- **Networking functions**
(standardized by POSIX 1.g, still in draft)
accept, bind, connect, getdomainname, gethostbyaddr, gethostbyname,
getpeername, getprotobyname, getprotoynumber, getservbyname,
getservbyport, getsockname, getsockopt, perror, htonl, htons, inet_addr,
inet_makeaddr, inet_netof, inet_ntoa, listen, ntohl, ntohs, rcmd, recv,
recvfrom, rexec, rresvport, send, sendto, setsockopt, shutdown, socket,
socketpair.

Of these networking calls, rexec, rcmd and rresvport are implemented in MS IP stack but may not be implemented in other vendor stacks.

- **Other functions**
chroot, closelog, cwait, cygwin_conv_to_full_posix_path,
cygwin_conv_to_full_win32_path, cygwin_conv_to_posix_path,
cygwin_conv_to_win32_path, cygwin_posix_path_list_p,
cygwin_posix_to_win32_path_list,
cygwin_posix_to_win32_path_list_buf_size, cygwin_split_path,
cygwin_win32_to_posix_path_list,
cygwin_win32_to_posix_path_list_buf_size, cygwin_winpid_to_pid,

dlclose, dlerror, dlfork, dlopen, dlsym, endgrent, ffs, fstatfs, ftime, get_osfhandle, getdtablesize, getgrent, gethostname, getitimer, getmntent, getpagesize, getpgid, getpwent, gettimeofday, grantpt, initgroups, ioctl, killpg, login, logout, lstat, mknod, memccpy, nice, openlog, pclose, popen, ptsname, putenv, random, readv, realpath, regfree, rexec, select, setegi, setenv, seterrno, seteuid, setitimer, setmntent, setmode, setpassent, setpgrp, setpwent, settimeofday, sexecl, sexecle, sexeclp, sexecclpe, sexecclpe, sexecp, sexecv, sexecve, sexecvpe, sigpause, spawnl, spawnle, spawnlp, spawnlpe, spawnv, spawnve, spawnvp, spawnvpe, srandom, statfs, strsignal, strtosigno, swab, syslog, timezone, truncate, ttyslot, unlockpt, unsetenv, usleep, utimes, vfork, vhangup, wait3, wait4, wcscmp, wcslen, wprintf, writev

`initgroups` does nothing.

`chroot`, `mknod`, `settimeofday`, and `vhangup` always return -1 and sets `errno` to `ENOSYS`.

`seteuid`, `setegid`, and `settimeofday` always return 0 and sets `errno` to `ENOSYS`.

`vfork` just calls `fork`.

2

Setting up Cygwin

The following documentation discusses setting up the Cygwin tools.

- “Directory Structure for Cygwin” on page 238
- “Microsoft Windows NT security and ntsec usage” on page 238
- “Environment Variables for Cygwin” on page 247
- “Mount Table” on page 250
- “Text and Binary Modes” on page 250
- “Using GCC with Cygwin” on page 255
- “Debugging Cygwin Programs” on page 256
- “Building and Using DLLs with Cygwin” on page 257
- “Defining Microsoft Windows Resources for Cygwin” on page 258
- “Cygwin Utilities” on page 262
- “Cygwin Functions” on page 276

The following packages are included in the native Win32 release of GNUPro.

- GNUPro development tools: `binutils`, `bison`, `byacc`, `dejag`, `diff`, `expect`, `flex`, `gas`, `gcc`, `gdb`, `itcl`, `ld`, `libstdc++`, `make`, `patch`, `tcl`, `tix`, `tk`.
- GNUPro unsupported tools: `ash`, `bash`, `bzip2`, `diff`, `fileutils`, `findutils`, `gawk`, `grep`, `gzip`, `m4`, `sed`, `shellutils`, `tar`, `textutils`, `time`.

Directory Structure for Cygwin

Cygwin knows how to emulate a standard UNIX directory structure, to some extent. You should make sure that you always have `/tmp` both with and without the `mount` table translations, just in case. If you want to emulate the `/etc` directory (so that the UNIX declaration, `ls -l`, works), use the following example's declarations as a guide.

```
mkdir /etc/  
cd /etc  
mkpasswd > /etc/passwd  
mkgroup > /etc/group
```

IMPORTANT! This input only works fully under Microsoft Windows NT. With Microsoft Windows 95 and Microsoft Windows 98, you may need to edit these files with a text editor.

Further changes to your Microsoft Windows NT registry will *not* be reflected in `/etc/passwd` or `/etc/group` after this implementation, so you may want to regenerate these files periodically. You should also set your home directories to something other than `/` to prevent unexplained delays in various programs.

Cygwin comes with two shells: `bash.exe` and `sh.exe`. `sh.exe` is based on `ash`. The system is faster when `ash` is used as the non-interactive shell. In case of trouble with `ash`, make `sh.exe` point to `bash.exe`.

Microsoft Windows NT security and `ntsec` usage

The setting of UNIX like object permissions is controlled by the `CYGWIN` variable setting, `ntsec`, as well as its opposite, `nontsec`. Microsoft Windows NT security allows a process to allow or deny access of different kind to objects (files, processes, threads, semaphores, and other UNIX entities). The main data structure of Microsoft Windows NT security is the *security descriptor* (SD) structure, which explains the permissions granted (or denied) an object, and which contains information related to so called *security identifiers* (SID, a unique identifier for *users*, *groups*, and *domains*). SIDs are comparable to UNIX *user identifiers* (UIDs) and *group identifiers* (GIDs), but SIDs are more complicated because they are unique across networks. The following example shows the SID of a system, `foo`.

```
S-1-5-21-165875785-1005667432-441284377
```

The following example shows the SID of a user, `johnndoe`, of the system, `foo`.

```
S-1-5-21-165875785-1005667432-441284377-1023
```

The previous example shows the convention for printing SIDs. `s` shows that it is a SID. The next number is a version number which is always 1. The next number is the *top-level authority* that identifies the source that issued the SID.

While each system in a Microsoft Windows NT network has its own SID, the situation is modified in Microsoft Windows NT domains. The SID of the domain controller is the base SID for each domain user. If an Microsoft Windows NT user has one account as domain user and another account on a local machine, these accounts are, under any circumstance, *different*, regardless of the usage of the same user name and password.

The following example shows the SID of a domain, `bar`.

```
S-1-5-21-186985262-1144665072-740312968
```

Compare the previous example with the following example, showing a user, `johnndoe`, in the domain, `bar`.

```
S-1-5-21-186985262-1144665072-740312968-1207
```

The last part of the SID, the *relative identifier* (RID), is by default used as a UID and/or a GID under Cygwin. As the name in the previous example implies, this identifier is unique only relative to one system or domain.

IMPORTANT! It's possible that a user has the same RID on two different systems. The resulting SIDs are nevertheless different, so the SIDs are representing different users in an Microsoft Windows NT network.

There is a big difference between UNIX IDs and Microsoft Windows NT SIDs, the existence of *well known groups*. For example, UNIX has no GID for the group of *all users*. Microsoft Windows NT has an SID for them (in the English versions), `Everyone`. The SIDs of well-known groups are not unique across an Microsoft Windows NT network but their meanings are unmistakable. The following example shows well-known groups.

```
everyone                S-1-1-0
creator/owner           S-1-3-0
batch process (via 'at') S-1-5-3
authenticated users    S-1-5-11
system                  S-1-5-18
```

The last important group of SIDs are the *predefined groups*, used mainly on systems outside of domains to simplify the administration of user permissions. The corresponding SIDs are not unique across the network so they are interpreted only locally, as the following example shows.

```
administrators          S-1-5-32-544
users                   S-1-5-32-545
guests                  S-1-5-32-546
```

Permissions are also given to objects, a process assigning an SD to the object, which consists of the following three parts.

- the SID of the owner
- the SID of the group
- a list of SIDs with their permissions, an *access control list* (ACL)

UNIX is able to create three different permissions, the permissions for the owner, for the group and for the world. In contrast, the ACL has a potentially infinite number of members. Every member is an *access control element* (ACE). An ACE contains the following three parts:

- the type of the ACE
- permissions, described with a `DWORD`
- the SID, for which the previous permissions are set

The two important types of ACEs are the *access allowed* ACE and the *access denied* ACE. `ntsec` only uses access allowed ACEs.

The possible permissions on objects are more complicated than in UNIX. For example, the permission to delete an object is different from the write permission.

With the aforementioned method, Microsoft Windows NT is able to grant or revoke permissions to objects in a far more specific way. With Cygwin, in a POSIX environment, it would be fine to have the security behavior of a POSIX system.

Because there's a leak in the Microsoft Windows NT model, the Microsoft Windows NT security model is only mostly able to reproduce the POSIX model, which `ntsec` alleviates; for more information, see "Mapping leak" on page 243.

The creation of explicit object security is a bit complicated, so typically only two simple variations are used:

- default permissions, computed by the operating system
- each permission to everyone

For parameters to functions that create or open securable objects another data structure is used, the *security attributes* (SA). This structure contains an SD and an option that specifies whether the returned handle to the created or opened object is inherited to child processes or not. This property is not important in describing `ntsec`, so assume SDs and SAs are more or less identical.

Any process started under control of Cygwin has a semaphore attached to it, used for signaling purposes. The creation of this semaphore can be found in `sigproc.cc`, with the function, `getsem`. The first parameter to the `CreateSemaphore` function call is an SA. Without `ntsec`, this SA assigns default security to the semaphore. There is a simple disadvantage in that only the owner of the process may send signals to it. In

other words, if the owner of the process is not a member of the administrators' group, no administrator may kill the process. This is especially annoying, if processes are started using the service manager.

`ntsec` now assigns an SA to the process control semaphore, which has each permission set for the user of the process, for the administrators' group, and for `system`, which is a synonym for the operating system itself. The creation of this SA is done by the `sec_user` function, which can be found in `shared.cc`. Each member of the administrators' group is now allowed to send signals to any process created in Cygwin, regardless of the process owner. Moreover, each process now has the appropriate security settings, when it is started using `CreateProcess`. You will find this in the `spawn_guts` function in the `spawn.cc` module. The security settings for starting a process in another user context have to add the SID of the new user. In the case of the `CreateProcessAsUser` call, `sec_user` creates an SA with an additional entry for the sid of the new user.

If `ntsec` is turned on, file permissions are set as in UNIX. An SD is assigned to the file containing the owner and group and ACEs for the owner, the group and `Everyone`.

The complete settings of UNIX like permissions can be found in the `security.cc` file. The two functions, `get_nt_attribute` and `set_nt_attribute`, are the main code. The reading and writing of the SDs is done by the `read_sd` and `write_sd` functions. `write_sd` uses the `BackupRead` function instead of the simpler function, `SetFileSecurity`, because the latter is unable to set owners different from the caller.

If you are creating a `foo` file outside of Cygwin, you will see something like the following examples show for output, if you use the `ls -ln` command.

- If your login is associated with the administrators' group:


```
rw-rw-rw- 1 544 513 ... foo
```
- If your login is not associated with the administrators' group:


```
rw-rw-rw- 1 1000 513 ... foo
```

544 is the UID of the administrators' group. This is a feature of Microsoft Windows NT. If you are a member of the administrators' group, every file that you created is owned by the administrators' group, instead of by you. The second example shows the UID of the first user that you created with Microsoft Windows NT's user administration tool. The users and groups are sequentially numbered, starting with 1000. Users and groups have the same numbering scheme, so a user and a group don't share the same ID. In both examples, the 513 GID is of special interest. This GID is a well known group with different naming in local systems and domains. Outside of domains, the group is named `None` (`Kein` in German, `Aucun` in French, among others); in domains, it is named `Domain Users`. Unfortunately, the `None` group is never shown in the user admin tool outside of domains, which is very confusing, although, it seems, it has no negative influences. To work correctly, `ntsec` depends on reasoned files,

`/etc/passwd/` and `/etc/group`. The names and the IDs must correspond to the appropriate Microsoft Windows NT IDs. The IDs used in Cygwin are the RID of the Microsoft Windows NT SID.

Unfortunately, workstations and servers outside of domains are not able to set primary groups. In these cases, where there is no correlation of users to primary groups, Microsoft Windows NT returns 513 (*None*) as primary group, regardless of the membership to existing local groups. When using the `mkpasswd -l -g` command on such systems, you have to change the primary group by hand if *None* as primary group is not what you want.

Groups may be mentioned in the `passwd` file, which has two advantages, because Microsoft Windows NT assigns them to files as owners, an `ls -l` command is often more readable, and because it's possible to assign files to owners with Cygwin's `chown` command.

The `system` group is the aforementioned synonym for the operating system itself and is normally the owner of processes started through a service manager; the same is true for files that are created by processes started through a service manager.

There is a new technique of using `/etc/passwd` and `/etc/group`. Both files may now contain SIDs of users and groups. They are saved in the last field of `pw_gecos` in `/etc/passwd` and in the `gr_passwd` field in `/etc/group`. This has the following advantages:

- `ntsec` works better in domain environments.
- Accounts (users and groups) may get another name in Cygwin that their Microsoft Windows NT account name. The name in `/etc/passwd` or `/etc/group` is transparently used by Cygwin applications (for example, `chown`, `chmod`, and `ls`); use `root::500:513::/home/root:/bin/sh` instead of `administrator::500:513::/home/root:/bin/sh`.

WARNING! If you like to use the account as login account using something like the `telnet` command, you have to let the login name remain unchanged; a future release will provide a special version of the `login` command.

- Cygwin UIDs and GIDs are now not necessarily the RID part of the Microsoft Windows NT SID; use `root::0:513:S-1-5-21-54355234-56236534-345635656-500:/home/root:/bin/sh` instead of `root::500:513::/home/root:/bin/sh`.
- As in UNIX systems, UIDs and GIDs have a numbering scheme now that doesn't influence each identifier. So it's possible to have the same identifiers for a user and a group:
 - `/etc/passwd:`
`root::0:0:S-1-5-21-54355234-56236534-345635656-500:/home/root:/bin/sh`

- `/etc/group:`

```
root:S-1-5-32-544:0:
```

The `mkgpasswd` and `mkgroup` tools create the needed entries by default. If you don't want that you can use the `-s` or `--no-sids` options.

IMPORTANT! The `pw_gecos` field in `/etc/passwd` is defined as a comma separated list. The SID has to be the last field.

You are able to use Cygwin account names different from the Microsoft Windows NT account names. If you want to login using `telnet` or something else, you have to use a special `login` command. You may then add another field to `pw_gecos`, containing the Microsoft Windows NT user name including its domain in order to login as each domain user; just add an entry of the form, `U-ntdomain\ntusername`, to the `pw_gecos` field. The SID must still remain the last field in `pw_gecos`, as the following example shows.

```
loginname::1:1:name,U-STILLHERE\name,S-1-5-21-1234-5678-9012-1000:/bin/sh
```

For a local user, drop the domain, as the following example shows.

```
loginname::1:1:name,U-name,S-1-5-21-1234-5678-9012-1000:/bin/sh
```

In each case, the password of the user is taken from the Microsoft Windows NT user database, not from the `passwd` file.

Mapping leak

There is a leak in Microsoft Windows NT permissions. The official documentation has the following explanation.

- Access allows ACEs to be accumulated according to the group membership of the caller.
- The order of ACEs is important; the system reads them in sequence until either any needed right is denied or all needed rights are granted, and ACEs are later then not taken into account.
- All access denied ACEs should precede any access allowed ACE.

IMPORTANT! The last rule is a preference, not a law; Microsoft Windows NT will correctly deal with the ACL regardless of the sequence order.

The second rule is not modified to get the ACEs in the preferred order.

Unfortunately, the **Security** tab of the Microsoft Windows NT 4 Explorer is completely unable to deal with access denied ACEs while the Explorer of Microsoft Windows 2000 rearranges the order of the ACEs before you can read them. The sort order remains unchanged if one presses the **Cancel** button.

Microsoft Windows NT ACLs are unable to reflect each possible combination of POSIX permissions. For example, use the following command.

```
rw-r-xrw-
```

On the first try, you use the following input.

```
UserAllow: 110 GroupAllow: 101 OthersAllow: 110
```

Because of the accumulation of allow rights, the user may execute because the group may execute.

On the second try, you use the following alternative input.

```
UserDeny: 001 GroupAllow: 101 OthersAllow: 110
```

Now you may read and write but not execute. Unfortunately, the group may write now because others may write.

For a third try, you use the following alternative input.

```
UserDeny: 001 GroupDeny: 010 GroupAllow: 001 \
OthersAllow: 110
```

Now, the group may not write as intended but unfortunately the user may not write anymore. To solve the problem, according to the official rules, a `UserAllow` has to follow the `GroupDeny` but this can never be solved that way. The only chance would entail using the following example's input.

```
UserDeny: 001 UserAllow: 010 GroupDeny: 010 \
GroupAllow: 001 OthersAllow: 110
```

This solution works for both Microsoft Windows NT 4 and Microsoft Windows 2000. Only the GUIs aren't able to deal with that order.

Cygwin API Calls

There are the following API calls.

For dealing with ACLs, Cygwin now has the `acl` API as it's implemented in newer versions of Solaris. The new data structure for a single ACL entry (ACE in Microsoft Windows NT terminology) is defined in `sys/acl.h` as the following example shows.

```
typedef struct acl
{
    int      a_type;
    /* entry type */
    uid_t    a_id;
    /* UID | GID */
    mode_t   a_perm;
    /* permissions */
}
aclent_t;
```

The `a_perm` member of the `aclent_t` type contains only the bits for read, write and execute as in the file mode. If read permission is granted, all read bits (`S_IRUSR`, `S_IRGRP`, and `S_IROTH`) are set. `CLASS_OBJ` or `MASK` ACL entries are not fully implemented yet.

The following calls are the `acl` calls: `acl(2)`, `facl(2)` `aclcheck(3)`, `aclsort(3)`, `acltomode(3)`, `aclfrommode(3)`, `acltopbits(3)`, `aclfrompbits(3)`, `acltotext(3)`, and `aclfromtext(3)`.

Like the Sun Solaris operating system, Cygwin has two commands for working with ACLs on the command line, `getfacl` and `setfacl`. For documentation, see <http://docs.sun.com>.

The `setuid` Concept

UNIX applications that have to switch the user context have the `setuid` and `seteuid` calls, which are not part of the Microsoft Windows API. Nevertheless these calls are supported under Windows Microsoft Windows NT and Microsoft Windows 2000 with Cygwin. Because of the nature of Microsoft Windows NT security an application which needs the ability has to be patched, though.

Microsoft Windows NT uses *access tokens* to identify a user and its permissions. To switch the user context, the application has to request such an access token. This is typically done by calling the Microsoft Windows NT API function, `LogonUser`. The access token is returned and either used in `ImpersonateLoggedOnUser` to change user context of the current process or in `CreateProcessAsUser` to change user context of a spawned child process. An important restriction is that the application using `LogonUser` must have special permissions:

```
"Act as part of the operating system"
"Replace process level token"
"Increase quotas"
```

Administrators do not have all that user rights set by default.

Two Cygwin calls support porting `setuid` applications with a minimum of effort. You have to give Cygwin the right access token and then you can call `seteuid` or `setuid` as in POSIX applications. The call to `sexec` is not a requirement. Porting a

```
/* First include all needed cygwin stuff. */
#ifdef __CYGWIN__
#include windows.h
#include sys/cygwin.h
/* Use the following define to determine the Windows version */
#define is_winnt      (GetVersion() < 0x80000000)
#endif
```

[...]

```
    struct passwd *user_pwd_entry = getpwnam (username);
    char *cleartext_password = getpass ("Password:");
```

[...]

```
#ifdef __CYGWIN__
    /* Patch the typical password test. */
    if (is_winnt)
    {
        HANDLE token;

        /* Try to get the access token from NT. */
        token = cygwin_logon_user (user_pwd_entry,
cleartext_password);
        if (token == INVALID_HANDLE_VALUE)
            error_exit;
        /* Inform Cygwin about the new impersonation token.
        Cygwin is able now, to switch to that user context by
        setuid or seteuid calls. */
        cygwin_set_impersonation_token (token);
    }
    else
#endif /* CYGWIN */
    /* Use standard method for W9X as well. */
    hashed_password = crypt (cleartext_password, salt);
    if (!user_pwd_entry ||
        strcmp (hashed_password, user_pwd_entry-pw_password))
        error_exit;

[...]
```

/* Everything else remains the same! */

```
setegid (user_pwd_entry-pw_gid);
seteuid (user_pwd_entry-pw_uid);
execl ("/bin/sh", ...);
```

The Cygwin call to retrieve an access token is defined as follows:

```
#include <windows.h>;
#include <sys/cygwin.h>
HANDLE cygwin_logon_user          \
    (struct passwd *pw, const char *cleartext_password)
```

You can call that function as often as you want for different user logons and remember the access tokens for further calls to the second function, as the following example shows.

```
#include <windows.h>
#include <sys/cygwin.h>
void cygwin_set_impersonation_token (HANDLE hToken);
```

The previous example shows how to inform Cygwin about the user context to which further calls to `setuid` and `seteuid` commands switch. While you always need the correct access token to do a `setuid` or `seteuid` command to another user's context, you are always able to use the `setuid` or `seteuid` commands to return to your own user context by giving your own UID as parameter.

If you have remembered several access tokens from calls to `cygwin_logon_user`, you can switch to different user contexts by using the following example's order.

```
cygwin_set_impersonation_token (user1_token);
seteuid (user1_uid);
```

[...]

```
seteuid (own_uid);
cygwin_set_impersonation_token (user2_token);
seteuid (user2_uid);
```

[...]

```
seteuid (own_uid);
cygwin_set_impersonation_token (user1_token);
seteuid (user1_uid);
```

Environment Variables for Cygwin

Before starting `bash`, you must set some environment variables, some of which can also be set or modified inside `bash`. You have a `.bat` file where the most important ones are set before initially invoking `bash`. The fully editable `.bat` file installs by default in `..\cygwin/cygnus.bat` and the **Start** menu points to it.

The most important environment variable is the `CYGWIN` variable. The `CYGWIN` variable is used to configure many global settings for the Cygwin runtime system. Initially you can leave `CYGWIN` unset or set it to `tty` using input like the following example's syntax in a DOS shell, before launching `bash`.

```
C:\..\> set CYGWIN=tty notitle
```

The `PATH` environment variable is used by Cygwin applications as a list of directories to search for executable files to run. Convert this environment variable, when a

Cygwin process first starts, from a Microsoft Windows format (C:\WinNT\system32;C:\WinNT) to UNIX format (/WinNT/system32:/WinNT).

Set the `PATH` environment variable so that, before launching `bash`, it contains at least a `bin` directory: C:\..\cygwin\H-i586-cygwin32\bin.

`make` uses an environment variable, `MAKE_MODE`, to decide if it uses **Command.com** or `/bin/sh` to run command lines. If you're getting strange errors from `make` with the `/c not found` message, set `MAKE_MODE` to UNIX with a declaration like the following example's form.

```
C:\> set MAKE_MODE=UNIX
$ export MAKE_MODE=UNIX
```

The `HOME` environment variable is used by UNIX shells to determine the location of your home directory. This environment variable is converted from the Microsoft Windows format (that is, C:\home\bob) to UNIX format (that is, /home/bob) when a Cygwin process first starts. To prevent confusion, ensure that `HOME` and `/etc/passwd` agree on your home directory.

The `TERM` environment variable specifies your terminal type. It is set it to `cygwin`.

The `LD_LIBRARY_PATH` environment variable is used by the Cygwin function, `dlopen()`, as a list of directories to search for `.dll` files to load. This environment variable is converted from the Microsoft Windows format (that is,

C:\WinNT\system32;C:\WinNT) to UNIX format (that is, /WinNT/system32:/WinNT) when a Cygwin process first starts.

The `CYGWIN` environment variable is used to configure many global settings for the Cygwin runtime system, using the following options.

IMPORTANT! Each option is separated by others with a space. Many options can be turned off by prefixing with `no` (such as `nobar` or `bar` options).

- `(no)binmode`
If set, unspecified file opens by default to binary mode (no `CR/LF` or **Ctrl+Z** translations) instead of text mode. This option must be set before starting a Cygwin shell to have an effect on redirection. On by default.
- `(no)envcache`
If set, environment variable conversions (between Win32 and POSIX) are cached. Note that this is may cause problems if the mount table changes, as the cache is not invalidated and may contain values that depend on the previous mount table contents. Defaults to set.
- `(no)export`
If set, the final values of these settings are re-exported to the environment as `$CYGWIN` again.

- `(no)title`
If set, the title bar reflects the currently running program's name. Off by default.
- `(no)glob`
If set, command line arguments containing UNIX-style file wildcard characters (brackets, question mark, asterisk) are expanded into lists of files that match those wildcards. This is applicable only to programs running from a windows command line prompt. Set by default.
- `(no)tty`
If set, Cygwin enables extra support (such as `termios`) for UNIX-like `tty` calls. Off by default.
- `(no)ntea`
If set, use the full Microsoft Windows NT **Extended Attributes** to store UNIX-like inode information.

WARNING! `ntea` may create additional large files on non-NTFS partitions. `ntea` only operates under Microsoft Windows NT. Off by default.

- `(no)smbntsec`
If `smbntsec` is set, use `ntsec` on remote drives as well (this is the default). If you encounter problems with Microsoft Windows NT shares or Samba drives, setting this to `nosmbntsec` could help. In that case the permission and owner/group information is faked as on FAT partitions. A reason for a non working `ntsec` on remote drives could be insufficient permissions of the users. Since the needed user rights are somewhat dangerous, it's not always an option to grant those rights to users. However, this shouldn't be a problem in Microsoft Windows NT domain environments. Default is `smbntsec`.
- `(no)reset_com`
If `reset_com` is set, serial ports are reset to 9600-8-N-1 with no flow control when used. This is done at open time and when handles are inherited. Default is `reset_com`.
- `(no)strip_title`
If `strip_title` is set, strips the directory part off the window title, if any. Defaults to `strip_title`.
- `(no)title`
If `title` is set, the title bar reflects the name of the program currently running. Under Microsoft Windows 95 and Microsoft Windows 98, the title bar is always enabled and it is stripped by default, but this is because of the way Microsoft Windows 95 and Microsoft Windows 98 work. In order not to strip, specify `title` or `title nostrip_title`. Defaults to `nostrip_title`.

- (no)ntsec
 If `ntsec` is set, use the Microsoft Windows NT security model to set UNIX-like permissions on files and processes. The file permissions can only be set on NTFS partitions. FAT doesn't support the Microsoft Windows NT file security. For more information, see "Microsoft Windows NT security and ntsec usage" on page 238. Defaults to be not set.

Mount Table

The `mount` utility controls a mount table for emulating a POSIX view of the Microsoft Windows file system space. Use it to change the Microsoft Windows path that uses `/` and mount arbitrary Win32 paths into the POSIX file system space. Many people use the utility to mount each drive letter under the slash partition (such as `C:\` to `/c` or `D:\` to `/d`, and so forth).

Executing `mount` without any arguments prints the current mount table to the screen. Otherwise, provide the Win32 path you would like to mount as the first argument and the POSIX path as the second argument. The following example demonstrates using the `mount` utility to mount the `C:/../..../H-i586-cygwin/bin` directory to the `/bin` folder, and the network directory, `\\pollux\home\joe\data` to `/data`. This makes `/bin/sh` a valid shell, to satisfy `make`. `/bin` is assumed to already exist.

```
c:\..\> ls /bin /data
ls: /data: No such file or directory
c:\..\> mount C:\..\cygwin\H-i586-cygwin\bin /bin
c:\..\> mount \\pollux\home\joe\data /data
Warning: /data does not exist!
c:\..\> mount
```

Device	Directory	Type	Flags
\\pollux\home\joe\data	/data	native	text!=binary
C:\..\cygwin\H-i586-cygwin\bin	/bin	native	text!=binary
D:	/d	native	text!=binary
\\.\tape1:	/dev/st1	native	text!=binary
\\.\tape0:	/dev/st0	native	text!=binary
\\.\b:	/dev/fd1	native	text!=binary
\\.\a:	/dev/fd0	native	text!=binary
C:	/	native	text!=binary

```
c:\..\> ls /bin/sh
/bin/sh
```

The `mount` table is stored in the Microsoft Windows registry (`HKEY_CURRENT_USER/Software/Cygnus Solutions/Cygwin/mounts v2`).

Text and Binary Modes

The following documentation discusses some of the main distinction with text and binary modes with UNIX and Microsoft Windows interoperability, and how Cygwin

solves the problems. See also “Programming” on page 253, “File Permissions” on page 253 and “Special File Names” on page 254.

On a UNIX system, when an application reads from a file it gets exactly what’s in the file on disk and the same is true for writing to the file. The situation is different in the DOS and Microsoft Windows world where a file can be opened in one of two modes, either binary or text. In the binary mode, the system behaves exactly as in UNIX.

However in text mode there are major differences:

- On writing in text mode, a new line, NL ($\backslash n$, $\wedge J$), is transformed into a carriage return/new line sequence, or CR ($\backslash r$, $\wedge M$) NL.
- On reading in text mode, a carriage return followed by a new line is deleted and a $\wedge z$ character signals the end of file.

The mode can be specified explicitly; see “Programming” on page 253. In an ideal DOS and Microsoft Windows world, all programs using lines as records (such as `bash`, `make`, or `sed`) would open files (changing the mode of their standard input and output) as text. All other programs (such as `cat`, `cmp`, or `tr`) would use binary mode. In practice with Cygwin, programs that deal explicitly with object files specify binary mode (as is the case of `od`, which is helpful to diagnose CR problems). Most other programs (such as `cat`, `cmp`, `tr`) use the default mode. The Cygwin system gives us some flexibility in deciding how files are to open when the mode is not specified explicitly:

- If the file appears to reside on a file system that is mounted (that is, if its pathname starts with a directory displayed by `mount`), then the default is specified by the `mount` flag.

The `CYGWIN` environment variable will affect a disk file when you are using `stdio` redirection from the Microsoft Windows prompt. Otherwise, the mode of the file is based on the mode specified by a `--change-cygdrive-prefix` call, as the following example shows (which makes the default mode be binary).

```
mount -b --change-cygdrive-prefix /cygdrive
```

If the file is a symbolic link, the mode of the target file system applies.

- Pipes and non-file devices are always opened in binary mode.
- When a Cygwin program is launched by a shell, its standard input, output and error are in binary mode.

When redirecting, the Cygwin shells uses the first three rules. For these shells, the relevant value of `CYGWIN` is that at the time the shell was launched and not that at the time the program is executed.

Non-Cygwin shells always pipe and redirect with binary mode. With non-Cygwin shells, the `cat filename | program` and `program < filename` commands are not equivalent when `filename` is on a text-mounted partition. To illustrate the

various rules, the following example's script deletes CRs from files by using the `tr` program, which can only write to standard output.

```
#!/bin/sh
# Remove \r from the files given as arguments
for file in "$@"
do
    CYGWIN=binmode sh -c "tr -d \\\"\\r\\\" < '$file' >
c:tmpfile.tmp"
    if [ "$?" = "0" ]
    then
        rm "$file"
        mv c:tmpfile.tmp "$file"
    fi
done
```

The script works irrespective of the mount because the second rule applies for the path, `c:tmpfile.tmp`. According to the fourth rule, `CYGWIN` must be set before invoking the shell. These precautions are necessary because `tr` does not set its standard output to binary mode. It would thus reintroduce `\r` when writing to a file on a text mounted partition. The desired behavior can also be obtained by using `tr -d \r` in a `.bat` file.

UNIX programs that have been written for maximum portability will know the difference between text and binary files and act appropriately under Cygwin. For those programs, the text mode default is a good choice. Programs included in official distributions should work well in the default mode.

Text mode makes it much easier to mix files between Cygwin and Microsoft Windows programs, since Microsoft Windows programs will usually use the carriage return/line feed (CR/LF) format. Unfortunately you may still have some problems with text mode. First, some of the utilities included with Cygwin do not yet specify binary mode when they should; `cat` will not work, for instance, with binary files (input will stop at `^z`, CRs will be introduced in the output). Second, you will introduce CRs in text files you write, causing problems when moving them back to a UNIX system.

If you are mounting a remote file system from a UNIX machine, or moving files back and forth to a UNIX machine, you can access them in binary mode since text files found there will normally be NL format anyway, and you would want any files put there by Cygwin programs to be stored in a format that the UNIX machine will understand. Remove CRs from all Makefiles and shell scripts and make sure that you only edit the files with DOS/Windows editors that can cope with binary mode files.

IMPORTANT! You can decide this on a disk by disk basis (for example, mounting local disks in text mode and network disks in binary mode). You can also partition a disk, for example by mounting `c:` in text mode, and `c:\home` in binary mode.

Programming

In the `open()` function call, binary mode can be specified with the flag, `O_BINARY`, and text mode with `O_TEXT`. These symbols are defined in `fcntl.h`.

In the `fopen()` function call, binary mode can be specified by adding a `b` to the `mode` string. There is no direct way to specify text mode.

The mode of a file can be changed by the call, `setmode(fd,mode)` where `fd` is a file descriptor (an integer) and `mode` is `O_BINARY` or `O_TEXT`. The function returns, `O_BINARY` or `O_TEXT`, depending on the mode before the call, and `EOF` on error.

File Permissions

On Microsoft Windows 95 and Microsoft Windows 98 systems, files are always readable, and Cygwin uses the native read-only mode to determine if they are writable. Files are considered to be executable if the filename ends with `.bat`, `.com` or `.exe`, or if its content starts with `#!`. Consequently `chmod` can only affect the `w` mode, whereas it silently ignores actions involving the other modes. With Microsoft Windows NT, file permissions default to the same behavior as Microsoft Windows 95 and Microsoft Windows 98. However, there is optional functionality in Cygwin that can make file systems behave more like on UNIX systems. This is turned on by adding the `ntea` and `ntsec` options to the `CYGWIN` environment variable; for more information, see “Microsoft Windows NT security and `ntsec` usage” on page 238 and “Environment Variables for Cygwin” on page 247.

When the `ntea` feature is activated, Cygwin will start with basic permissions, while storing POSIX file permissions in Microsoft Windows NT *Extended Attributes*. This feature works quite well on NTFS partitions because the attributes can be stored sensibly inside the normal NTFS filesystem structure. However, on a FAT partition, Microsoft Windows NT stores extended attributes in a flat file at the root of the `EA DATA. SF` partition. This file can grow to extremely large sizes if you have a large number of files on the partition in question, slowing the system to a crawl. In addition, the `EA DATA. F` file can only be deleted outside of Microsoft Windows because of its *in use* status. For these reasons, the use of Microsoft Windows NT *Extended Attributes* is off by default in Cygwin. Finally, specifying `ntea` with Cygwin has no effect with Microsoft Windows 95 and Microsoft Windows 98. With Microsoft Windows NT, the `[-w filename]` test is only true if `filename` is writable across the board, such as with a `chmod +w filename` call.

Special File Names

The following documentation discusses some special file naming usage by Cygwin.

- **DOS devices**
Microsoft Windows filenames invalid with Microsoft Windows are also invalid under Cygwin; this means that base filenames such as `AUX`, `COM1`, `LPT1` or `PRN` cannot be used with Cygwin for a Microsoft Windows or POSIX path, even with an extension (`prn.txt`). Special names can be used as filename extensions (`file.aux`). You can use the special names as you would under DOS; for example you can print on your default printer with the command, `cat filename > PRN` (making sure to end with a Form Feed).

- **POSIX devices**
There is no need to create a POSIX `/dev` directory as it is simulated within Cygwin automatically. It supports the following devices: `/dev/null`, `/dev/tty` and `/dev/comX` (the serial ports). These devices cannot be seen with the command, `ls /dev`, although commands such as `ls /dev/tty` work fine.

- **The `.exe` extension**
Executable program filenames end with `.exe` but the `.exe` extension is not necessary in the command, so that traditional UNIX names can be used. To the contrary the `.bat` and `.com` extensions cannot be omitted.

As a side effect, the `'ls filename'` gives information about `filename.exe` if `filename.exe` exists and `filename` does not. In the same situation, the function, call `stat("filename" ...)`, gives information about `filename.exe`. The two files can be distinguished by examining their inodes, as the following example's script demonstrates.

```
C:\Cygwin directory\> ls *
a          a.exe  b.exe
C:\Cygwin directory\> ls -i a a.exe
445885548 a          435996602 a.exe
C:\Cygwin directory\> ls -i b b.exe
432961010 b          432961010 b.exe
```

The GCC compiler produces an executable named `filename.exe` when asked to produce `filename`. This allows many makefiles written for UNIX systems to work well under Cygwin. Unfortunately the `install` and `strip` commands do distinguish between `filename` and `filename.exe`. They fail when working on a non-existing filename even if `filename.exe` exists, thus breaking some makefiles. This problem can be solved by writing `install` and `strip` shell scripts to provide the `.exe` extension when needed.

- **@pathname**
To circumvent the limitations on shell line length in the native Microsoft

Windows command shells, Cygwin programs expand their arguments starting with `@` in a special way. If a file pathname exists, the argument, `@pathname` expands recursively to the content of `pathname`.

Double quotes can be used inside the file to delimit strings containing blank space. In the following example, compare the behaviors of the `bash` built-in `echo` and of the `/bin/echo` program.

```

/..$ echo 'This is "a long" line' > mylist
/..$ echo @mylist
@mylist
/..$ /bin/echo @mylist
This is a      long line
/..$ rm mylist
/..$ /bin/echo @mylist
@mylist

```

Using GCC with Cygwin

The following documentation discusses using the GNUPro compiler, GCC, with Cygwin.

- “Console Mode Applications” (below)
- “GUI Mode Applications” (below)

Console Mode Applications

Use GCC to compile, just like under UNIX. See *Using GNU CC* in the *GNUPro Compiler Tools* documentation for information on standard usage and options. The following example shows the usage practice for the shell’s console.

```

C:\..\> gcc hello.c -o hello.exe
C:\..\> hello.exe
Hello, World

C:\..\>

```

GUI Mode Applications

Cygwin allows you to build programs with full access to the standard Microsoft Windows 32-bit API, including the GUI functions as defined in any Microsoft or off-the-shelf publication. However, the process of building those applications is slightly different, as you’ll be using the GNU tools instead of the Microsoft tools.

For the most part, your sources won’t need to change at all. However, you should remove all `__export` attributes from functions and replace them. The following example’s script shows such implementation.

```
int foo (int) __attribute__ ((__dllexport__));

int
foo (int i)
```

For most cases, you can just remove the `__export` attributes. For convenience sake, you might want to work around a misfeature in Cygwin’s libraries by including the following code fragment; otherwise, you’ll have to add a `-e _mainCRTStartup` declaration to your link line in your Makefile, as the following example shows.

```
#ifdef __CYGWIN__
WinMainCRTStartup() { mainCRTStartup(); }
#endif
```

The Makefile is similar to any other UNIX-like Makefile, and any other Cygwin Makefile. The only difference is that you use a “`gcc -mwindows`” declaration to link your program into a GUI application instead of a command line application. The following example’s script shows an implementation.

```
myapp.exe : myapp.o myapp.res
            gcc -mwindows myapp.o myapp.res -o $@

myapp.res : myapp.rc resource.h
            windres $< -O coff -o $@
```

IMPORTANT! The use of `windres` is for compiling the Microsoft Windows resources into a COFF-format `.res` file. That will include all the bitmaps, icons, and other resources you need, into one handy object file. Normally, if you omitted the `-O coff` declaration, it would create a Microsoft Windows `.res` format file, with the capability to link only COFF objects. So, `windres` now produces a COFF object, for compatibility with the many examples that assume your linker can handle Microsoft Windows resource files directly, using the `.res` naming convention. For more information on `windres`, see *Using binutils* in the *GNUPro Utilities* documentation.

Debugging Cygwin Programs

When your programs don’t work properly, they usually have bugs (meaning there’s something wrong with the program itself that is causing unexpected results or crashes). Diagnosing these bugs and fixing them is made easy by special tools called debuggers. In the case of Cygwin, the debugger is GDB, accessed with Insight, the GUI debugger tool that lets you run your program in a controlled environment so that you can investigate the state of your program while it is running or after it crashes.

Before you can debug your program, you need to prepare your program for debugging. Add a `-g` declaration to all the other flags you use when compiling your

sources to objects. Consider the following example's declarations.

```
gcc -g -O2 -c myapp.c
gcc -g myapp.c -o myapp
```

What this does is add extra information to *myapp* objects (making them much bigger), telling the debugger about line numbers, variable names, and other useful things. These extra symbols and debugging data give your program enough information about the original sources so that the debugger can make debugging much easier for you.

To invoke it, use the `gdb myapp.exe` declaration (substituting the executable file's name for *myapp*) at the command prompt. See "Insight, GDB's Alternative Interface" on page 171 in the *GNUPro Debugging Tools* for information about Insight and how to use it. See also *Debugging with GDB* in the *GNUPro Debugging Tools* for information about GDB and how to use it; for GDB, you may have to type `thread 1` to switch to the main program thread.

If your program crashes and you're trying to determine why it crashed, the best thing to do is type `run` and let your program run. After it crashes, you can use the `where` command to determine where it crashed, or `info locals` to see the values of all the local variables. There's also the `print` declaration that lets you examine individual variables or what pointers point to. If your program is doing something unexpected, you can use the `break` command to tell GDB to stop your program when it gets to a specific function or line number, as the following example shows.

```
(gdb) break my_function
(gdb) break 47
```

Now, using the `run` command, your program will stop at that breakpoint, and you can use the other GDB commands to look at the state of your program at that point, to modify variables, and to step through your program's statements one at a time.

IMPORTANT! Specify additional arguments to the `run` command to provide command-line arguments to your program. These previous example's cases and the next example's case are the same as far as your program is concerned:

```
myprog -t foo --queue 47

gdb myprog
(gdb) run -t foo --queue 47
```

Building and Using DLLs with Cygwin

The following documentation discusses building and using dynamically linked libraries (DLLs) with Cygwin.

DLLs are linked into your program at run time instead of build time.

The following documentation describes the three parts to a DL: *exports*, *code and data* and the *import library*.

- *exports*
A list of functions and variables that the `.dll` file makes available to other programs, as a list of *global* symbols, with the rest of the contents hidden. Normally, you'd create this list by hand with a text editor; however, it's possible to do it automatically from the list of functions in your code. The `dlltool` program creates the exports section of the `.dll` file from your text file of exported symbols.
- *code and data*
The parts you write: functions, variables, etc. All these are merged together, for instance, for building one big object file and linking it to a `.dll` file. They are not put into your `.exe` at all.
- *import library*
The import library is a regular UNIX-like `.a` library, but it only contains the tiny bit of information needed to tell the operating system how your program interacts with (or *imports*) the `.dll` as data. This information is linked into your `.exe` file. This is also generated by the `dlltool` utility.

Building DLLs

The following documentation provides an example of how to build a `.dll` file, using a single file, `myprog.c`, for the program, `myprog.exe`, and a single file, `mydll.c`, for the contents of the `.dll` file, `mydll.dll`, then compiling everything as objects.

```
gcc -shared myprog.c -o mydll.dll -e _mydll_init@12
```

Now, when you build your program, you link against the import library, with declaration's like the following example's commands.

```
gcc myprog.o mydll.dll -o myprog.exe
```

Defining Microsoft Windows Resources for Cygwin

`windres` reads a Microsoft Windows resource file (`*.rc`) and converts it to a `res` or `coff` file. The syntax and semantics of the input file are the same as for any other resource compiler; see any publication describing the Microsoft Windows resource format for details. Also, see the `windres` documentation in *Using binutils* in *GNUPro Utilities*. The following example shows the usage of `windres` in a project.

```
myapp.exe : myapp.o myapp.res  
          gcc -mwindows myapp.o myapp.res -o $@
```

```
myapp.res : myapp.rc resource.h
           windres $< -O coff -o $@
```

What follows is a quick-reference to the syntax that `windres` supports.

```
id ACCELERATORS suboptions
BEG
"^C" 12
"Q" 12
65 12
65 12 , VIRTKEY ASCII NOINVERT SHIFT CONTROL ALT
65 12 , VIRTKEY, ASCII, NOINVERT, SHIFT, CONTROL, ALT
(12 is an acc_id)
END
```

SHIFT, CONTROL, ALT require VIRTKEY

```
id BITMAP memflags "filename"
memflags defaults to MOVEABLE
```

```
id CURSOR memflags "filename"
memflags defaults to MOVEABLE,DISCARDABLE
```

```
id DIALOG memflags exstyle x,y,width,height styles BEG controls END
id DIALOGEX memflags exstyle x,y,width,height styles BEG controls END
id DIALOGEX memflags exstyle x,y,width,height,helpid styles BEG
controls END
```

```
memflags defaults to MOVEABLE
exstyle may be EXSTYLE=number
styles: CAPTION "string"
        CLASS id
        STYLE FOO | NOT FOO | (12)
        EXSTYLE number
        FONT number, "name"
        FONT number, "name",weight,italic
        MENU id
        CHARACTERISTICS number
        LANGUAGE number,number
        VERSIONK number
controls:
        AUTO3STATE params
        AUTOCHECKBOX params
        AUTORADIOBUTTON params
        BEDIT params
```

```

CHECKBOX params
COMBOBOX params
CONTROL ["name",] id, class, style, x,y,w,h [,exstyle] [data]
CONTROL ["name",] id, class, style, x,y,w,h, exstyle, helpid
[data]
    CTEXT params
    DEFPUSHBUTTON params
    EDITTEXT params
    GROUPBOX params
    HEDIT params
    ICON ["name",] id, x,y [data]
    ICON ["name",] id, x,y,w,h, style, exstyle [data]
    ICON ["name",] id, x,y,w,h, style, exstyle, helpid [data]
    IEDIT params
    LISTBOX params
    LTEXT params
    PUSHBOX params
    PUSHBUTTON params
    RADIOBUTTON params
    RTEXT params
    SCROLLBAR params
    STATE3 params
    USERBUTTON "string", id, x,y,w,h, style, exstyle
params:
    ["name",] id, x, y, w, h, [data]
    ["name",] id, x, y, w, h, style [,exstyle] [data]
    ["name",] id, x, y, w, h, style, exstyle, helpid [data]

[data] is optional BEG (string|number) [,(string|number)] (etc) END

id FONT memflags "filename"
memflags defaults to MOVEABLE|DISCARDABLE

id ICON memflags "filename"
memflags defaults to MOVEABLE|DISCARDABLE

LANGUAGE num,num

id MENU options BEG items END
items:
    "string", id, flags:
    SEPARATOR:
    POPUP "string" flags BEG menuitems END
flags::
    CHECKED:
    GRAYED:
    HELP:
    INACTIVE:

```

```

        MENUBARBREAK:
        MENUBREAK

id MENUEX suboptions BEG items END
items::
    MENUITEM "string":
    MENUITEM "string", id:
    MENUITEM "string", id, type [,state]:
    POPUP "string" BEG items END:
    POPUP "string", id BEG items END:
    POPUP "string", id, type BEG items END:
    POPUP "string", id, type, state [,helpid] BEG items END

memflags defaults to MOVEABLE

id RCDATA suboptions BEG (string|number) [,(string|number)] (etc) END

STRINGTABLE suboptions BEG strings END
strings::
    id "string":
    id, "string"

(User data)
id id suboptions BEG (string|number) [,(string|number)] (etc) END

id VERSIONINFO stuffs BEG verblocks END
stuffs: FILEVERSION num,num,num,num:
        PRODUCTVERSION num,num,num,num:
        FILEFLAGSMASK num:
        FILEOS num:
        FILETYPE num:
        FILESUBTYPE num:
verblocks::
    BLOCK "StringFileInfo" BEG BLOCK BEG vervals END END:
    BLOCK "VarFileInfo" BEG BLOCK BEG vertrans END END
vervals: VALUE "foo","bar"
vertrans: VALUE num,num

suboptions::
    memflags:
    CHARACTERISTICS num:
    LANGUAGE num,num:
    VERSIONK num

memflags are MOVEABLE/FIXED PURE/IMPURE PRELOAD/LOADONCALL
DISCARDABLE
    
```

Cygwin Utilities

Cygwin comes with a number of command line utilities for managing the UNIX emulation portion of the Cygwin environment. While many of these reflect their UNIX counterparts, each was written specifically for Cygwin. See the corresponding documentation to the following Cygwin utilities.

- “cygcheck” on page 263
- “cygpath” on page 264
- “kill” on page 266
- “mkgroup” on page 268
- “mkpasswd” on page 269
- “mount” on page 270
- “ps” on page 274
- “umount” on page 275

cygcheck

USAGE `cygcheck [-s] [-v] [-r] [-h] [program ...]`

DESCRIPTION `cygcheck` is a diagnostic utility that examines your system and reports the information that is significant to the proper operation of cygwin programs. It can give information about a specific program (or program) you are trying to run, general system information, or both. If you list one or more programs on the command line, it will diagnose the runtime environment of that program or programs.

The `cygcheck` program should be used to send information about your system to support for troubleshooting (if your support representative requests it).

```
cygcheck -s -v -r -h > tocygnus.txt
```

You must at least give either an `-s` option or a program name, signified in the usage as *program*.

Use the following options with the `cygcheck` utility.

- The `-s` option will give general system information. If you specify `-s` and list one or more programs on the command line, `cygcheck` reports on both specified programs.
- The `-v` option causes the output to be more verbose. What this means is that `cygcheck` will report additional information which is usually not interesting, such as the internal version numbers of DLLs, additional information about recursive DLL usage, and if a file in one directory in the `PATH` also occurs in other directories on the `PATH`.
- The `-r` option causes `cygcheck` to search your registry for information that is relevant to some programs. These registry entries are the ones that have “Cygnus” in the name. If you are concerned about privacy, you may remove information from this report, keeping in mind that doing so makes it harder for support staff to diagnose problems.
- The `-h` option prints additional helpful messages in the report, at the beginning of each section. It also adds table column headings. While this is useful information, this functionality also adds significantly to the size of the report; if you want a compact report or if you know what everything is already, don't use this option.

cygpath

USAGE `cygpath [-p|--path] (-u|--unix)|(-w|--windows) filename
[-v|--version] [-a|--absolute] [-c|--close handle]
[-f|--file file] [-u|--unix] [-w|--windows]
[-W|--windir] [-S|--sysdir]`

DESCRIPTION `cygpath` is a utility that converts Microsoft Windows native filenames to Cygwin POSIX-style pathnames and reverse. Use it when a Cygwin program needs to pass a *filename* to a native Microsoft Windows program, or when Cygwin expects to get a *filename* from a native Microsoft Windows program. Use the long or short option names interchangeably.

Use the following options with the `cygpath` utility.

- The `-p` option means that you want to convert a path-style string rather than a single filename. For example, the `PATH` environment variable is semicolon-delimited in Microsoft Windows, but colon-delimited in UNIX. By giving `-p` you are instructing `cygpath` to convert between these formats. Consider the following example's usage.

```
#!/bin/sh
for i in `echo *.exe | sed 's/\.\exe/cc/'`
do
notepad `cygpath -w $i`
done
```

- The `-u` and `-w` options indicate whether you want a conversion from Microsoft Windows to UNIX (POSIX) format (with `-u`) or conversion from UNIX (POSIX) to Microsoft Windows format (with `-w`). Give exactly one of these options. To give neither or both is an error.
- The `-v` option causes the output to be more verbose.
- The `--version` option prints the version of the utility.
- The `-a` and `--absolute` options provide output to an absolute path.
- The `-c` and `--close handle` options are for use in a captured process, enabling closing of a *handle*.
- The `-f` and `--file file` options allow reading of a *file* for its path information.
- The `-u` and `--unix` options provide the UNIX form of a filename.
- The `-w` and `--windows` options provide the Microsoft Windows form of a filename.
- The `-W` and `--windir` options provide a full Microsoft Windows path

for a program or tool.

- The `-s` and `--sysdir` options print the system directory.
- The `-p` and `--path` options provide a filename argument a path.

kill

USAGE kill [-sigN] pid1 [pid2 ...]

DESCRIPTION kill allows sending arbitrary signals to other Cygwin programs. The usual purpose is to end a running program from some other window when the keystroke combination, **Ctrl+C**, won't work, but you can also send program-specified signals such as `SIGUSR1` to trigger actions within the program, such as when enabling debugging or when re-opening log files.

Each program defines the signals they understand.

“pid” values are the Cygwin *process ID* values, not the Microsoft Windows process ID values. To get a list of running programs and their Cygwin PIDs, use the Cygwin `ps` program (see “ps” on page 274).

To send a specific signal, use the `-sig[n,M]` option, either with a signal number `[n]`, or with a signal name `[M]` (minus the “SIG” part), like the following example's input specifies, where 123 replaces the `n` option as the signal number.

```
kill 123
kill -1 123
kill -HUP 123
```

The following list provides the available signals, their numbers, and some commentary on them; the file, `<sys/signal.h>`, is the official source of this information.

SIGHUP	1	hangup
SIGINT	2	interrupt
SIGQUIT	3	quit
SIGILL	4	illegal instruction (not reset when caught)
SIGTRAP	5	trace trap (not reset when caught)
SIGABRT	6	used by abort
SIGEMT	7	EMT instruction
SIGFPE	8	floating point exception
SIGKILL	9	kill (cannot be caught or ignored)
SIGBUS	10	bus error
SIGSEGV	11	segmentation violation
SIGSYS	12	bad argument to system call
SIGPIPE	13	write on a pipe with no one to read it
SIGALRM	14	alarm clock
SIGTER	15	software termination signal from kill
SIGURG	16	urgent condition on IO channel
SIGSTOP	17	sendable stop signal not from tty
SIGTSTP	18	stop signal from tty
SIGCONT	19	continue a stopped process
SIGCHLD	20	to parent on child stop or exit

SIGCLD	20	System V name for SIGCHLD
SIGTTIN	21	to readers pgrp upon background tty read
SIGTTOU	22	like TTIN for output if (tp->t_local<OSTOP)
SIGIO	23	input/output possible signal
SIGPOLL	23	System V name for SIGIO
SIGXCPU	24	exceeded CPU time limit
SIGXFSZ	25	exceeded file size limit
SIGVTALRM	26	virtual time alarm
SIGPROF	27	profiling time alarm
SIGWINCH	28	window changed
SIGLOST	29	resource lost (eg, record-lock lost)
SIGUSR1	30	user defined signal 1
SIGUSR2	31	user defined signal 2

mkgroup

USAGE `mkgroup <options> [domain]`

DESCRIPTION For Microsoft Windows NT only, `mkgroup` prints group information to `stdout`.

Use `mkgroup` to help configure your Microsoft Windows NT system to be more UNIX-like, creating an initial `/etc/group` substitute (some commands need this file) from your system information. To initially set up your machine, use the following example's declarations as a guide.

```
mkdir /etc
mkgroup > /etc/group
```

This information is static. If you change the group information in your system, regenerate the `group` file for it to have the new information.

`mkgroup` can use the following options.

`-l`

`--local`

Prints pseudo group information if there is no domain.

`-d`

`--domain`

Prints global group information from the domain specified (or from the current domain if there is no domain specified).

`-?`

`--help`

Prints the following message description.

This program does only work on Windows NT

The `-d` and `-l` options allow you to specify where the information derives, either the default (or given) domain (with `-d`), or the local machine (with `-l`).

mkpasswd

USAGE `mkpasswd <options> [domain]`

DESCRIPTION For Microsoft Windows NT only, `mkpasswd` prints a `/etc/passwd` file to `stdout`.

`mkpasswd` helps to configure your Microsoft Windows NT system to be more UNIX-like by creating an initial `/etc/passwd` substitute (some commands need this file) from your system information. To initially set up your machine, use the following example's declarations as a guide.

```
mkdir /etc
mkpasswd > /etc/passwd
```

This information is static. If you change the user information in your system, regenerate the `passwd` file for it to have the new information.

The following options are useful with `mkpasswd`.

```
-l
--local
    Print local accounts.
-d
--domain
    Print domain accounts (from current domain if no domain specified).
-g
--local-groups
    Print local group information too.
-?
--help
    Displays the following message:
    This program does only work on Windows NT
```

The `-d` and `-l` options allow you to specify where the information derives, either the default (or given) domain (with `-d`), or the local machine (with `-l`).

mount

USAGE `mount [-bfstux] <dospath> <unixpath>`

DESCRIPTION Use `mount` to map your drives and share the simulated POSIX directory tree, much like the POSIX `mount` command or the DOS `join` command, making your drive letters appear as subdirectories somewhere else. In POSIX operating systems (like Linux™), there is no concept of drives, nor drive letters. All absolute paths begin with a slash instead of `c:`, and all file systems appear as subdirectories (for example, you might buy a new disk and make it be the `/disk2` directory). This practice is simulated by Cygwin to assist in porting POSIX programs to Microsoft Windows. Just give the DOS or Microsoft Windows equivalent path and where you want it to show up in the simulated POSIX tree, like the following example's declarations (in which *release* is the version that you acquired).

```
C:\..> mount c:\ /
C:\..> mount c:\release\bin /bin
C:\..> mount d:\ /usr/data
C:\..> mount e:\mystuff /mystuff
```

```
bash$ mount 'c:\' /
```

Since native paths use backslashes, and backslashes are special in most POSIX-like shells (like `bash`), you need to properly quote them if you are using such a shell. There are many opinions on what the proper set of mounts is, and the appropriate one for you depends on how closely you want to simulate a POSIX environment, whether you mix Microsoft Windows and Cygwin programs, and how many drive letters you are using. If you want to be very POSIX-like, you may want to use declarations like the following example shows (in which *release* is the version that you acquired).

```
C:\> mount c:\ release/
C:\> mount c:\ /c
C:\> mount d:\ /d
C:\> mount e:\ /cdrom
```

To share Microsoft Windows and Cygwin programs, create an *identity* mapping to eliminate problems of conversions between the two (see “`cygpath`” on page 264); for instance, use declarations like the following example shows.

```
C:\> mount c:\ \
C:\> mount d:\foo /foo
C:\> mount d:\bar /bar
C:\> mount e:\grill /grill
```

Repeat this process for all top-level subdirectories on all drives, in order to have the top-level directories available as the same names in both systems.

The `-b` and `-t` options change the default text file type for files found in that *mount point*. The default is text, which means that Cygwin will automatically convert files between the POSIX text style (each line ends with the `NL` new line character) and the Microsoft Windows text style (each line ends with a `CR` character and an `LF` character, or `CRLF`) as needed. The program can, and should, explicitly specify text or binary file access as needed, but not all do.

If your programs are properly written with the differentiation between text and binary files, the default (`-t`) is a good choice. You must use `-t` if you are going to mix files between Cygwin and Microsoft Windows programs, since Microsoft Windows programs will always use the `CRLF` format. Text files get `\\r\\n` line endings by default.

If you are mounting a remote filesystem from a UNIX machine, use `-b`, as the text files found there will normally be `NL` format anyway, and you would want any files put there by Cygwin programs to be stored in a format that the UNIX machine will understand.

You do not need to set up mounts for most devices in the POSIX `/dev` directory (like `/dev/null`) as these are simulated automatically within Cygwin.

The `-f` option forces `mount`, suppressing warnings about missing mount point directories.

The `-s` option adds a mount point to system-wide registry location.

The `-u` option adds a mount point `.to` a user registry location by default.

The `-x` option treats all files under a mount point as executables.

With the `-b` and `-s` options, the following options are available:

- `--change-cygdrive-prefix posixpath`
Changes the `/cygdrive` path prefix to *posixpath*.
- `--show-cygdrive-prefixes`
Shows user and/or system `/cygdrive` path prefixes.
- `--import-old-mounts`
Copies old registry mount table mounts into the current mount areas.

passwd

USAGE `passwd [name]`
`passwd [-x max] [-n min] [-i inact] [-L len]`
`passwd {-l|-u|-S} name`

DESCRIPTION `passwd` changes passwords for user accounts. A normal user may only change the password for their own account, and the administrators may change the password for any account. `passwd` also changes account information, such as password expiration dates and intervals.

- ***Password changes***

The user is first prompted for their old password, if one is present. This password is then encrypted and compared against the stored password. The user has only one chance to enter the correct password. The administrators are permitted to bypass this step so that forgotten passwords may be changed. The user is then prompted for a replacement password. `passwd` will prompt again and compare the second entry against the first. Both entries must match in order for the password to be changed. After the password has been entered, password aging information is checked to see if the user is permitted to change their password at this time. If not, `passwd` refuses to change the password and exits.

- ***Password expiry and length***

The password aging information may be changed by the administrators with the `-x`, `-n` and `-i` options. The `-x` option is used to set the maximum number of days a password remains valid. After `max` days, the password is required to be changed. The `-n` option is used to set the minimum number of days before a password may be changed. The user will not be permitted to change the password until `min` days have elapsed. The `-i` option is used to disable an account after the password has been expired for a number of days. After a user account has had an expired password for `inact` days, the user may no longer sign on to the account. Allowed values for the above options are 0 to 999. The `-L` option sets the minimum length of allowed passwords for users, which doesn't belong to the administrators' group, to `len` characters. Allowed values for the minimum password length are 0 to 14. A value of 0 means no restrictions in any of the previous cases.

- ***Account maintenance***

User accounts may be locked and unlocked with the `-l` and `-u` flags. The `-l` option disables an account. `-u` re-enables an account and the

account status may be given with the `-s` option. The status information is self explanatory.

- ***Limitations***
Users may not be able to change their password on some systems.

ps

USAGE `ps [-aefl] [-u uid]`

DESCRIPTION `ps` gives the status of all the Cygwin processes running on the system (`ps` stands for *process status*). Due to the limitations of simulating a POSIX environment with Microsoft Windows, there is little information to give. The `PID` column is the *process ID* you need to give to the `kill` command (see “kill” on page 266). The `WINPID` column is the process ID that displays for Microsoft Windows NT’s **Task Manager** program.

When using `ps`, the following options are available.

- a
- e
Shows processes of all users
- f
Shows process uids, ppids
- l
Shows process uids, ppids, pgids, winpids
- u uid
Lists processes owned by uid

umount

USAGE `umount <path>`

DESCRIPTION `umount` removes a mount from the system. You may specify either the Microsoft Windows path or the POSIX path. See “mount” on page 270 for information about the mount table.

Cygwin Functions

The following documentation discusses the Cygwin functions.

- “`cygwin_attach_handle_to_fd`” on page 276
- “`cygwin_conv_to_full_posix_path`” on page 276
- “`cygwin_conv_to_full_win32_path`” on page 276
- “`cygwin_conv_to_posix_path`” on page 277
- “`cygwin_conv_to_win32_path`” on page 277
- “`cygwin_detach_dll`” on page 277
- “`cygwin_getshared`” on page 277
- “`cygwin_internal`” on page 277
- “`cygwin_posix_path_list_p`” on page 277
- “`cygwin_posix_to_win32_path_list`” on page 278
- “`cygwin_posix_to_win32_path_list_buf_size`” on page 278
- “`cygwin_split_path`” on page 278
- “`cygwin_win32_to_posix_path_list`” on page 278
- “`cygwin_win32_to_posix_path_list_buf_size`” on page 278
- “`cygwin_winpid_to_pid`” on page 279

These functions are specific to Cygwin itself, and probably will not have relations to any other library or standards.

`cygwin_attach_handle_to_fd`

```
extern "C" int cygwin_attach_handle_to_fd(char *name, int fd, HANDLE handle, int bin, int access);
```

Converts a Win32 *handle* into a POSIX-style file handle. *fd* may be -1 to make Cygwin allocate a handle; the actual handle is returned in all cases.

`cygwin_conv_to_full_posix_path`

```
extern "C" void cygwin_conv_to_full_posix_path(const char *path, char *posix_path);
```

Converts a Win32 path to a POSIX path. If *path* is already a POSIX path, leaves it alone. If *path* is relative, then *posix_path* will be converted to an absolute path.

posix_path must point to a buffer of sufficient size; use `MAX_PATH` if needed.

`cygwin_conv_to_full_win32_path`

```
extern "C" void cygwin_conv_to_full_win32_path(const char *path, char *win32_path);
```

Converts a POSIX path to a Win32 path. If *path* is already a Win32 path, there is no

change. If *path* is relative, then *win32_path* will be converted to an absolute path. *win32_path* must point to a buffer of sufficient size; use `MAX_PATH` if needed.

cygwin_conv_to_posix_path

```
extern "C" void cygwin_conv_to_posix_path(const char *path, char *posix_path);
```

Converts a Win32 path to a POSIX path. If *path* is already a POSIX path, there is no change. If *path* is relative, then *posix_path* will also be relative.

posix_path must point to a buffer of sufficient size; use `MAX_PATH` if needed.

cygwin_conv_to_win32_path

```
extern "C" void cygwin_conv_to_win32_path(const char *path, char *win32_path);
```

Converts a POSIX path to a Win32 path. If *path* is already a Win32 path, there is no change. If *path* is relative, then *win32_path* will also be relative.

win32_path must point to a buffer of sufficient size; use `MAX_PATH` if needed.

cygwin_detach_dll

```
extern "C" void cygwin_detach_dll(int dll_index);
```

This function has unsupported functionality.

cygwin_getshared

```
shared_info * cygwin_getshared(void);
```

Returns a pointer to an internal Cygwin memory structure containing shared information used by cooperating Cygwin processes. This function is intended for use only by system programs like `mount` and `ps`.

cygwin_internal

```
extern "C" DWORD cygwin_internal(cygwin_getinfo_types t, ...);
```

Provides access to various internal data and functions.

WARNING! Use care with this function; its results are unpredictable.

cygwin_posix_path_list_p

```
extern "C" int posix_path_list_p(const char *path);
```

Provides information if the supplied *path* is a POSIX-style path (such as POSIX names, forward slashes, or colon delimiters) or a Win32-style path (such as drive letters, reverse slashes, or semicolon delimiters). The return value is true if the path is a POSIX path. “_p” means *predicate*, a lisp term meaning that the function tells you something about the parameter. Rather than use a mode to say what the *proper* path list format is, there are many, giving applications the tools they need to convert between the two. If a `;` is present in the *path* list, it’s a Win32 path list. Otherwise, if the first path begins with a drive letter and colon (in which case it can be the only element, since, if it wasn’t, a `;` would be present), it’s a Win32 path list. Otherwise,

it's a POSIX path list.

cygwin_posix_to_win32_path_list

extern "C" void cygwin_posix_to_win32_path_list(const char *posix, char *win32);
Given a POSIX path-style string (that is, /foo:/bar), converts to the equivalent Win32 path-style string (that is, d:\;e:\bar). Win32 must point to a sufficiently large buffer.

```
char *_epath;  
char *_win32epath;  
_epath = _win32epath = getenv (NAME);  
/* If have a POSIX path list, convert to win32 path list */  
if (_epath != NULL && *_epath != 0  
    && cygwin_posix_path_list_p (_epath))  
{  
    _win32epath = (char *) xmalloc  
        (cygwin_posix_to_win32_path_list_buf_size (_epath));  
    cygwin_posix_to_win32_path_list (_epath, _win32epath);  
}
```

See also “cygwin_posix_to_win32_path_list_buf_size” on page 278.

cygwin_posix_to_win32_path_list_buf_size

extern "C" int cygwin_posix_to_win32_path_list_buf_size(const char *path_list);
Returns the number of bytes needed to hold the result of calling cygwin_posix_to_win32_path_list.

cygwin_split_path

extern "C" void cygwin_split_path(const char *path, char *dir, char *file);
Splits a path into portions: the directory, *dir*, and the file, *file*. Both *dir* and *file* must point to buffers of sufficient size.

```
char dir[200], file[100];  
cygwin_split_path("c:/foo/bar.c", dir, file);  
printf("dir=%s, file=%s\n", dir, file);
```

cygwin_win32_to_posix_path_list

extern "C" void cygwin_win32_to_posix_path_list(const char *win32, char *posix);
Given a Win32 path-style string (that is, d:\;e:\bar), converts it to the equivalent POSIX path-style string (that is, /foo:/bar). POSIX must point to a sufficiently large buffer. See also “cygwin_win32_to_posix_path_list_buf_size” on page 278.

cygwin_win32_to_posix_path_list_buf_size

extern "C" int cygwin_win32_to_posix_path_list_buf_size(const char *path_list);
Informs how many bytes are needed for the results of cygwin_win32_to_posix_path_list.

cygwin_winpid_to_pid

```
extern "C" pid_t cygwin_winpid_to_pid (int winpid);
```

Given a Microsoft Windows process ID, *winpid*, converts to the corresponding Cygwin process ID, if any. Returns -1 if Microsoft Windows process ID does not correspond to a Cygwin process ID.

```
extern "C" cygwin_winpid_to_pid (int winpid);  
pid_t mypid;  
mypid = cygwin_winpid_to_pid (windows_pid);
```


Using info

Copyright © 1988-2000 Free Software Foundation

Permission is granted to make and distribute verbatim copies of this documentation, provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this documentation under the conditions for verbatim copying, provided also that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this documentation into another language, under the above conditions for modified versions.

While every precaution has been taken in the preparation of this documentation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information within the documentation.

For licenses and use information, see “General Licenses and Terms for Using GNUPro Toolkit” on page 105; specifically, see “GNU General Public License” on page 106, “GNU Lesser General Public License” on page 111, and “Tcl/Tk Tool Command Language and Windowing Toolkit License” on page 118 in *Getting Started Guide*.

1

Overview of `info`, the GNU Online Documentation

The GNU `info` program is online documentation used to view help files on an ASCII terminal. `info` files are the result of processing Texinfo files with the program, `Makeinfo`, using the Emacs editing command: **M-x** `texinfo-format-buffer`.

Texinfo is a documentation language allowing printed and online documentation (an `info` file) to be produced from a single source file. The following documentation discusses `info` in more detail.

- “Using the `info` program” on page 284
- “Reading `info` Files” on page 285
- “Making `info` files from Texinfo files” on page 303

Using the `info` program

`info` displays a summary of all its commands when you type with the `?` key. `info` is organized into *nodes*, corresponding to the chapters and sections of printed books. You can follow them in sequence just like in the printed book or, using menus, go quickly to the node having the information you need.

`info` has *hot* references; if one section refers to another, you can tell `info` to take you immediately to that other section. You can get back again easily to take up your reading where you left off. Naturally, you can also search for particular words and phrases.

The best way to get started with the online documentation system is to use a programmed tutorial by running `info` itself. You run `info` by just typing its name at your shell prompt (shown in the following example as `#`). No options or arguments are necessary.

```
# info
```

`info` displays its first screen, a menu of available documentation, and waits. To request a tutorial for learning `info`, use the `h` key. Or, from Emacs document browsing mode, in the window's command buffer, use the `Ctrl-h + i` key combinations.

Exit `info` any time by using the `q` key.

2

Reading `info` Files

You can read the documentation for GNU software either on paper, as with any other instruction manual, or as online `info` files, using an ordinary ASCII terminal.

You can browse through the online documentation with GNU Emacs. The program, `info`, is a small program intended just for the purpose of viewing help files.

`info` files are generated by the program, `makeinfo`, from a Texinfo source file.

Texinfo is a documentation markup language designed to allow the same source file to generate either printed or online documentation. The Texinfo language is described in *Texinfo (The GNU Documentation Format)**.

GNU `info` Command Line Options

GNU `info` accepts several options to control the initial node being viewed, and to specify which directories to search for `info` files. The following is a template showing an invocation of GNU `info` from the shell.

```
info [-- option-name option-value] menu-item ...
```

The following *option-names* are available when invoking `info` from the shell.

* *Texinfo (The GNU Documentation Format)* is published by the Free Software Foundation (ISBN 1-882114-12-4).

`--directory directory-path`

`-d directory-path`

Adds *directory-path* to the list of directory paths searched when Info needs to find a file. You may issue `--directory` multiple times; once for each directory which contains info files.

Alternatively, you may specify a value for the environment variable, `INFOPATH`; if `--directory` is not given, the value of `INFOPATH` is used. The value of `INFOPATH` is a colon separated list of directory names. If you do not supply `INFOPATH` or `--directory-path`, a default path is used.

`--file filename`

`-f filename`

Specifies a particular info file to visit. Instead of visiting the file, `dir, info` will start with (*filename*) Top as the first file and node.

`--node nodename`

`-n nodename`

Specifies a particular node to visit in the initial file loaded. This is especially useful in conjunction with `--file`[†].

You can specify `--node` multiple times. For an interactive Info session, each *nodename* is visited in its own window. For a non-interactive Info (such as when `--output` is given), each *node-name* is processed sequentially.

`--output filename`

`-o filename`

Specify *filename* as the name of a file to output to. Each node that Info visits will be output to *filename* instead of interactively viewed. A value of `-` for *filename* specifies the standard output.

`--subnodes`

This option only has meaning given in conjunction with `--output`. It means to recursively output the nodes appearing in the menus of each node being output. Menu items which resolve to external info files are not output, and neither are menu items which are members of an index. Each node is only output once.

`--help`

`-h`

Produces a relatively brief description of the available Info options.

`--version`

Prints the version information of `info` and exits.

`menu-item`

Remaining arguments to `info` are treated as the names of menu items. The first argument would be a menu item in the initial node visited, while the second

[†] You can specify both the file and node in a `--node` command; but don't forget to escape the open and close parentheses from the shell as in `info --node '(emacs)Buffers'`

argument would be a menu item in the first argument's node. You can easily move to the node of your choice by specifying the menu names which describe the path to that node, as in the following example.

```
info emacs buffers
```

This input example selects the menu item `Emacs` in the node `(dir)Top`, and then selects the menu item `Buffers` in the node `(emacs)Top`.

Moving the Cursor

Many people find that reading screens of text page by page is made easier when one is able to indicate particular pieces of text with some kind of pointing device. Since this is the case, GNU `info` (both the Emacs and standalone versions) have several commands which let you move the cursor about the screen. The notation in this documentation to describe keystrokes is identical to the notation used within the Emacs manual, and the GNU Readline manual. See “Characters, Keys and Commands” in *GNU Emacs Manual*[‡], if you are unfamiliar with the notation.

The following table lists the basic `info` cursor movement commands.

Each entry consists of the key sequence to type to perform the cursor movement: a command like **Meta-x**^{**}, and a short description of what it does.

Cursor motion also uses a *numeric* argument; for further discussion, see “Miscellaneous Commands” on page 298. A numeric argument executes a command that many times; for example, `4` given to `next-line` moves the cursor *down* 4 lines. A negative numeric argument reverses the motion; thus, an argument of `-4` for the `next-line` command moves the cursor *up* 4 lines.

Ctrl-n (`next-line`)

Moves the cursor down to the next line.

Ctrl-p (`prev-line`)

Move the cursor up to the previous line.

Ctrl-a (`beginning-of-line`)

Move the cursor to the start of the current line.

Ctrl-e (`end-of-line`)

Moves the cursor to the end of the current line.

[‡] *GNU Emacs Manual* is published by the Free Software Foundation (ISBN 1-882114-03-5).

^{**} **Meta-x** is also a command; it invokes `execute-extended-command`. See “Keyboard Input” in the *GNU Emacs Manual* for more detailed information.

Ctrl-f (`forward-char`)

Move the cursor forward a character.

Ctrl-b (`backward-char`)

Move the cursor backward a character.

Meta-f (`forward-word`)

Moves the cursor forward a word.

Meta-b (`backward-word`)

Moves the cursor backward a word.

Meta-< (`beginning-of-node`)

b

Moves the cursor to the start of the current node.

Meta-> (`end-of-node`)

Moves the cursor to the end of the current node.

Meta-r (`move-to-window-line`)

Moves the cursor to a specific line of the window. Without a numeric argument, **Meta-r** moves the cursor to the start of the line in the center of the window. With a numeric argument of n , **Meta-r** moves the cursor to the start of the n th line in the window.

Moving Text within a Window

Sometimes you are looking at a full screen of text, and only part of the current paragraph you are reading is visible on the screen. The commands detailed in the following section are used to shift which part of the current node is visible on the screen.

SPC / Spacebar (`scroll-forward`)

Ctrl-v

Shifts the text in this window up to show more of the node which is currently below the bottom of the window. A numeric argument shows that many more lines at the bottom of the window; a numeric argument of 4 shifts all text in the window up 4 lines (discarding the top 4 lines), and shows four new lines at the bottom of the window. Without a numeric argument, **Spacebar** takes the bottom two lines of the window and places them at the top of the window, redisplaying almost a completely new screenful of lines.

Del /Delete (`scroll-backward`)

Meta-v

Shifts the text in this window down, the inverse of scroll-forward.

`scroll-forward` and `scroll-backward` moves forward and backward through

the node structure of the file. If you press **Spacebar** while viewing the end of a node, or **Del** while viewing the beginning of a node, what happens is controlled by the variable `scroll-behavior`. See “Manipulating Variables,” page Manipulating Variables for more information.

Ctrl-l (`redraw-display`)

Redraws the display from scratch, or shifts the line with the cursor to a specified location. With no numeric argument, **Ctrl-l** clears the screen, and then redraws its entire contents. With a numeric argument of *n*, the line with the cursor is shifted to the *n*th line of the window.

Ctrl-x w (`toggle-wrap`)

Toggles the state of line wrapping in the current window. Normally, when lines *wrap* when they are longer than the screen width,; i.e., they continue on the next line. Lines which wrap display a `\` in the rightmost column of the screen. You can cause such lines to be terminated at the rightmost column by changing the state of line wrapping in the window with **Ctrl-x, w**. When a line contains more than one screen width, `$` appears in the rightmost column of the line, and the remainder is invisible.

Selecting a New Node

The following documentation details the numerous `info` commands which select a new node to view in the current window.

The most basic node commands are **n**, **p**, **u**, and **l**.

When you are viewing a node, the top line of the node contains some `info pointers` which describe where the *next*, *previous*, and *up* nodes are. `info` uses this line to move about the node structure of the file when you type the following commands.

n (`next-node`)

Selects the `Next` node.

p (`prev-node`)

Selects the `Prev` (previous) node.

u (`up-node`)

Selects the `Up` node.

You can easily select a node that you have already viewed in this window by using the **l** command (this name stands for *last*), to actually move through the list of already visited nodes for this window. **l** with a negative numeric argument moves forward through the history of nodes for this window, so you can quickly step between two adjacent (in viewing history) nodes.

l (`history-node`)

Selects the most recently selected node in this window.

Two additional commands make it easy to select the most commonly selected nodes; they are **t** and **d**.

t (`top-node`)

Selects the node `Top` in the current info file.

d (`dir-node`)

Selects the directory node (i.e., the node, `(dir)`).

The following are some other commands which immediately result in the selection of a different node in the current window.

< (`first-node`)

Selects the first node which appears in this file. This node is most often `Top`, but it doesn't have to be.

> (`last-node`)

Selects the last node which appears in this file.

] (`global-next-node`)

Moves forward or down through node structure. If the node that you are currently viewing has a `Next` pointer, that node is selected. Otherwise, if this node has a menu, the first menu item is selected. If there is no `Next` and no menu, the same process is tried with the `Up` node of this node.

[(`global-prev-node`)

Moves backward or up through node structure. If the node that you are currently viewing has a `Prev` pointer, that node is selected. Otherwise, if the node has an `Up` pointer, that node is selected, and if it has a menu, the last item in the menu is selected.

`global-next-node` and `global-prev-node` behave the same as simply scrolling through the file with **Spacebar** and **Del**; see `scroll-behavior` in “Manipulating Variables” on page 299 for more information.

g (`goto-node`)

Reads the name of a node and selects it. No completion is done while reading the node name, since the desired node may reside in a separate file. The node must be typed exactly as it appears in the `info` file. A file name may be included as with any node specification, as in the following example.

```
g(emacs)Buffers
```

This input finds the `Buffers` node in the `emacs info` file.

Ctrl-x, k (`kill-node`)

Kills a node. The node name is prompted for in the echo area, with a default of the current node. *Killing* a node means that `info` tries hard to forget about it,

removing it from the list of history nodes kept for the window where that node is found. Another node is selected in the window which contained the killed node.

Ctrl-x, Ctrl-f (*view-file*)

Reads the name of a file and selects the entire file.

Ctrl-x, Ctrl-f *filename*, is equivalent to typing **g**(*filename*)*

Ctrl-x, Ctrl-b (*list-visited-nodes*)

Makes a window containing a menu of all of the currently visited nodes. This window becomes the selected window, and you may use the standard *info* commands within it.

Ctrl-x, b (*select-visited-node*)

Selects a node which has been previously visited in a visible window. This is similar to **Ctrl-x, Ctrl-b** followed by **m**, but no window is created.

Searching an info File

GNU *info* allows you to search for a sequence of characters throughout an entire *info* file, search through the indices of an *info* file, or find areas within an *info* file which discuss a particular topic.

s (*search*)

Reads a string in the echo area and searches for it.

Ctrl-s (*isearch-forward*)

Interactively searches forward through the *info* file for a string you type.

Ctrl-r (*isearch-backward*)

Interactively searches backward through the *info* file for a string as you type it.

i (*index-search*)

Looks up a string in the indices for this *info* file, and selects the node that the found index entry points to.

, (*next-index-match*)

Moves to the node containing the next matching index item from the last **i** command.

The most basic searching command is **s** (*search*). The **s** command prompts you for a string in the echo area, and then searches the remainder of the *info* file for an occurrence of that string. If the string is found, the node containing it is selected, and the cursor is left positioned at the start of the found string. Subsequent **s** commands show you the default search string within [and]'; pressing **Enter**, instead of typing a new string will use the default search string.

Incremental searching is similar to basic searching, but the string is looked up while you are typing it, instead of waiting until the entire search string has been specified.

Selecting Cross References

We have already discussed the `Next`, `Prev`, and `Up` pointers which appear at the top of a node. In addition to these pointers, a node may contain other pointers which refer you to a different node, perhaps in another `info` file. Such pointers are called *cross references*, or *xrefs* for short.

Parts of a Cross Reference

Cross references have two major parts: the first part is called the *label*; it is the name that you can use to refer to the cross reference, and the second is the *target*; it is the full name of the node to which the cross reference points.

The target is separated from the label by a colon `:`; first, the label appears, and then the target. For instance, the following example's input shows a cross reference menu, where the single colon separates the label from the target.

```
* Foo Label: Foo Target. More information about Foo.
```

The `.` is not part of the target; it serves only to let `info` know where the target name ends.

A shorthand way of specifying references allows two adjacent colons to stand for a target name, as in the following example.

```
* Foo Commands:: Commands pertaining to Foo.
```

In the previous example, the name of the target is the same as the name of the label, in this case `Foo Commands`.

You will normally see two types of cross references while viewing nodes: *menu* references, and *note* references. Menu references appear within a node's menu; they begin with a `*` at the beginning of a line, and continue with a label, a target, and a comment which describes what the contents of the node pointed to contains.

IMPORTANT! References appear within the body of the node text; they begin with `*Note`, and continue with a label and a target.

Like `Next`, `Prev` and `Up` pointers, cross references can point to any valid node. They are used to refer you to a place where more detailed information can be found on a particular subject.

See “Cross References” in *Texinfo (The GNU Documentation Format)*^{††}, for more information on creating your own Texinfo cross references.

^{††} *Texinfo (The GNU Documentation Format)* is published by the Free Software Foundation (ISBN 1-882114-12-4).

Selecting Cross References

The following lists the `info` commands that operate on menu items.

1 (`menu-digit`)

2 ... 9

Within an `info` window, pressing a single digit, (such as **1**), selects that menu item, and places its node in the current window. For convenience, there is one exception; pressing **0** selects the *last* item in the node's menu.

0 (`last-menu-item`)

Select the last item in the current node's menu.

m (`menu-item`)

Reads the name of a menu item in the echo area and selects its node. Completion is available while reading the menu label.

Meta-x `find-menu`

Moves the cursor to the start of this node's menu.

The following lists the `info` commands which operate on note cross references.

f (`xref-item`)

r

Reads the name of a note cross reference in the echo area and selects its node. Completion is available while reading the cross reference label.

Finally, the next few commands operate on both menu or note references.

Tab (`move-to-next-xref`)

Moves the cursor to the start of the next nearest menu item or note reference in the current node. You can also then use the following command, **Return** (`select-reference-this-line`), to select the menu or note reference.

Meta-Tab (`move-to-prev-xref`)

Moves the cursor to the start of the nearest previous menu item or note reference in the current node.

Enter / Return (`select-reference-this-line`)

Selects the menu item or note reference appearing on the line where the cursor currently is.

Manipulating Multiple Windows

A *window* is a place to show the text of a node. Windows have a view area where the text of the node is displayed, and an associated *mode* line, which briefly describes the node being viewed. GNU `info` supports multiple windows appearing in a single screen; each window is separated from the next by its modeline. At any time, there is only one *active* window, that is, the window in which the cursor appears. There are

commands available for creating windows, changing the size of windows, selecting which window is active, and for deleting windows.

The Mode Line

A *mode* line is a line of inverse video which appears at the bottom of an `info` window. It describes the contents of the previously displayed window; this information includes the name of the file and node appearing in that window, the number of screen lines it takes to display the node, and the percentage of text that is above the top of the window. It can also tell you if the indirect tags table for this `info` file needs to be updated, and whether or not the `info` file was compressed when stored on disk. The following is a sample mode line for a window containing an uncompressed file named `dir`, showing the node `Top`.

```
-----Info: (dir)Top, 40 lines --Top-----
           ^^^  ^^^          ^^
           (file)Node #lines where
```

When a node comes from a file which is compressed on disk, this is indicated in the mode line with two small `z`'s. In addition, if the `info` file containing the node has been split into subfiles, the name of the subfile containing the node appears in the mode line as well.

```
--zz-Info: (emacs)Top, 291 lines --Top-- Subfile: emacs-1.Z-
```

When `info` makes a node internally, such that there is no corresponding `info` file on disk, the name of the node is surrounded by asterisks (*). The name itself tells you what the contents of the window are; the following sample mode line shows an internally constructed node showing possible one possible completion.

```
-----Info: *Completions*, 7 lines --All-----
```

Window Commands

To view more than one node at a time, `info` can display more than one window. Each window has its own mode line (see “The Mode Line” on page 294) and history of nodes viewed in that window (for information on `history-node`, see “Selecting a New Node” on page 289).

Ctrl-x, o (`next-window`)

Selects the next window on the screen. The echo area can *only* be selected if it is already in use, and you have left it temporarily. Normally, **Ctrl-x, o** simply moves the cursor into the next window on the screen, or if you are already within the last window, into the first window on the screen. Given a numeric argument, **Ctrl-x, o** moves over that many windows. A negative argument causes **Ctrl-x, o** to select the previous window on the screen.

Meta-x (`prev-window`)

Selects the previous window on the screen. This is identical to **Ctrl-x, o** with a

negative argument.

Ctrl-x, 2 (`split-window`)

Splits the current window into two windows, both showing the same node. Each window is one half the size of the original window, and the cursor remains in the original window. The variable, `automatiCtrl-tiling`, can cause all of the windows on the screen to be resized for you automatically; for more information on `automatiCtrl-tiling`, see “Manipulating Variables” on page 299.

Ctrl-x, 0 (`delete-window`)

Deletes the current window from the screen. If you have made too many windows and your screen appears cluttered, this is the way to get rid of some of them.

Ctrl-x, 1 (`keep-one-window`)

Deletes all of the windows excepting the current one.

Esc Ctrl-v (`scroll-other-window`)

Scrolls the other window, in the same fashion that **Ctrl-v** might scroll the current window. Given a negative argument, the *other* window is scrolled backward.

Ctrl-x, ^ (`grow-window`) Grows (or shrinks) the current window. Given a numeric argument, grows the current window that many lines; with a negative numeric argument, the window is shrunk instead.

Ctrl-x, t (`tile-windows`)

Divides the available screen space among all of the visible windows. Each window is given an equal portion of the screen in which to display its contents. The variable `automatiCtrl-tiling` can cause `tile-windows` to be called when a window is created or deleted. For more information on `automatiCtrl-tiling`, see “Manipulating Variables” on page 299.

The Echo Area

The *echo area* is a one line window which appears at the bottom of the screen. It is used to display informative or error messages, and to read lines of input from you when that is necessary. Almost all of the commands available in the echo area are identical to their Emacs counterparts, so please refer to GNU Emacs documentation for greater depth of discussion on the concepts of editing a line of text.

The following briefly details the commands that are available while input is being read in the echo area.

Ctrl-f (`echo-area-forward`)

Moves forward a character.

Ctrl-b (`echo-area-backward`)

Moves backward a character.

Ctrl-a (`echo-area-beg-of-line`)

Moves to the start of the input line.

Ctrl-e (echo-area-end-of-line)

Moves to the end of the input line.

Meta-f (echo-area-forward-word)

Moves forward a word.

Meta-b (echo-area-backward-word)

Moves backward a word.

Cd (echo-area-delete)

Deletes the character under the cursor.

Del (echo-area-rubout)

Deletes the character behind the cursor.

Ctrl-g (echo-area-abort)

Cancels or quits the current operation. If completion is being read, **Ctrl-g** discards the text of the input line which does not match any completion. If the input line is empty, **Ctrl-g** aborts the calling function.

RET (echo-area-newline)

Accepts (or forces completion of) the current input line.

Ctrl-q (echo-area-quoted-insert)

Inserts the next character verbatim; for example, so you can insert control characters into a search string.

printing character (echo-area-insert)

Inserts the character.

Meta-Tab (echo-area-tab-insert)

Inserts a **Tab** character.

Ctrl-t (echo-area-transpose-chars)

Transposes the characters at the cursor.

The next group of commands deal with killing and yanking text. For an in depth discussion of killing and yanking, see “Killing and Moving Text” in the *GNU Emacs Manual*^{††}.

Meta-d (echo-area-kill-word)

Kills the word following the cursor.

Meta-Del (echo-area-backward-kill-word)

Kills the word preceding the cursor.

Ctrl-k (echo-area-kill-line)

Kills the text from the cursor to the end of the line.

Ctrl-x, Del (echo-area-backward-kill-line)

Kills the text from the cursor to the beginning of the line.

^{††} *GNU Emacs Manual* is published by the Free Software Foundation (ISBN 1-882114-03-5).

Ctrl-y (echo-area-yank)

Yanks back the contents of the last kill.

Meta-y (echo-area-yank-pop)

Yanks back a previous kill, removing the last yanked text first.

Sometimes when reading input in the echo area, the command that needed input will only accept one of a list of several choices. The choices represent the *possible completions*, and you must respond with one of them. Since there are a limited number of responses you can make, `info` allows you to abbreviate what you type, only typing as much of the response as is necessary to uniquely identify it. In addition, you can request `info` to fill in as much of the response as is possible; this is called *completion*.

The following commands are available when completing in the echo area.

Tab (echo-area-complete)**SPACEBAR**

Inserts as much of a completion as is possible.

? (echo-area-possible-completions)

Displays a window containing a list of the possible completions of what you have typed so far. For example, say the available choices are the following if you typed an **f**, followed by **?**.

```
bar foliate
food forget
```

Possible completions would contain the choices which begin with **f**.

```
foliate food forget
```

Pressing **Spacebar** or **Tab** would result in `fo` appearing in the echo area, since all of the choices which begin with `f` continue with `o`. Now, typing **I** followed by pressing **Tab** results in `foliate` appearing in the echo area, since that is the only choice which begins with `foI`.

Esc Ctrl-v (echo-area-scroll-completions-window)

Scrolls the completions window, if that is visible, or, if not, the *other* window.

Printing Out Nodes

You may wish to print out the contents of a node as a quick reference document for later use. `info` provides you with a command for printing. In general, we recommend that you use the C program utility, `Makeinfo`, to create an `info` file from a Texinfo source file and then, by using the command, `texify`, format the document and print the DVI (*Device Independent*) file. See *Texinfo (The GNU Documentation Format)**** manual for more details.

info also provides you with a command for printing.

Meta-x print-node

Pipes the contents of the current node through the command in the environment variable, `INFO_PRINT_COMMAND`. If the variable doesn't exist, the node is simply piped to `lpr`.

Miscellaneous Commands

GNU info contains several commands which self-document GNU info as the following discussions help to clarify.

Meta-x describe-command

Reads the name of an info command in the echo area and then displays a brief description of what that command does.

Meta-x describe-key

Reads a key sequence in the echo area, and then displays the name and documentation of the info command which a given key sequence invokes.

Meta-x describe-variable

Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

Meta-x where-is

Reads the name of an info command in the echo area, and then displays a key sequence which can be typed in order to invoke that command.

Ctrl-h (get-help-window)

?

Creates (or moves into) the window displaying ***help***, and places a node containing a quick reference card into it. This window displays the most concise information about GNU info available.

h (get-info-help-node)

Tries hard to visit the node **(info)help**. The info file, `info.texi`, distributed with GNU info, contains this node. Of course, the file must first be processed with `makeinfo`, and then placed into the location of your info directory.

The following are the commands for creating a numeric argument.

Ctrl-u (universal-argument)

Starts (or multiplies by 4) the current numeric argument. **Ctrl-u** is a good way to give a small numeric argument to cursor movement or scrolling commands.

Ctrl-u, **Ctrl-v** scrolls the screen 4 lines, while **Ctrl-u**, **Ctrl-u**, **Ctrl-n** moves the

*** *Texinfo (The GNU Documentation Format)* is published by the Free Software Foundation (ISBN 1-882114-12-4).

cursor down 16 lines.

Meta-1 (`add-digit-to-numericCtrl-arg`)

Meta-2... Meta-9

Adds the digit value of the invoking key to the current numeric argument. Once `info` is reading a numeric argument, you may just type the digits of the argument, without the **M** prefix. For example, you might give **Ctrl-1** a numeric argument of 32 by using the keystroke sequence, **Ctrl-u**, **3**, **2**, **Ctrl-1** or **Meta-3**, **2**, **Ctrl-1**.

Ctrl-g is used to abort the reading of a multi-character key sequence, to cancel lengthy operations (such as multi-file searches) and to cancel reading input in the echo area.

Ctrl-g (`abort-key`)

Cancels current operation.

q (`quit`)

Exits `info`.

If the operating system tells `info` that the screen is 60 lines tall, and it is actually only 40 lines tall, the following is a way to tell `info` that the operating system is correct.

Meta-x `set-screen-height`

Reads a height value in the echo area and sets the height of the displayed screen to that value.

Finally, `info` provides a convenient way to display footnotes which might be associated with the current node that you are viewing:

Esc Ctrl-f (`show-footnotes`)

Shows the footnotes (if any) associated with the current node in another window. You can have `info` automatically display the footnotes associated with a node when the node is selected by setting the variable, `automaticCtrl-footnotes`; for more information on `automaticCtrl-footnotes`, see “Manipulating Variables” on page 299.

Manipulating Variables

GNU `info` contains several variables whose values are looked at by various `info` commands. You can change the values of these variables, and thus change the behavior of `info` to more closely match your environment and `info` file reading manner.

Meta-x `set-variable`

Reads the name of a variable, and the value for it, in the echo area and then sets the variable to that value. Completion is available when reading the variable name; often, completion is available when reading the value to give to the

variable, but that depends on the variable itself. If a variable does not supply multiple choices to complete over, it expects a numeric value.

Meta-x describe-variable

Reads the name of a variable in the echo area and then displays a brief description of what the variable affects.

What follows is a list of the variables that you can set in `info`.

`automatiCtrl-footnotes`

When set to `On`, footnotes appear and disappear automatically. This variable is `On` by default. When a node is selected, a window containing the footnotes which appear in that node is created, and the footnotes are displayed within the new window. The window that `info` creates to contain the footnotes is called `*Footnotes*`. If a node is selected which contains no footnotes, and a `*Footnotes*` window is on the screen, the `*Footnotes*` window is deleted. Footnote windows created in this fashion are not automatically tiled so that they can use as little of the display as is possible.

`automatiCtrl-tiling`

When set to `On`, creating or deleting a window *resizes* other windows. This variable is `off` by default. Normally, typing `Ctrl-x, 2` divides the current window into two equal parts. When `automatiCtrl-tiling` is set to `On`, all of the windows are resized automatically, keeping an equal number of lines visible in each window. There are exceptions to the automatic tiling; specifically, the windows `*Completions*` and `*Footnotes*` are *not* resized through automatic tiling; they remain their original size.

`visible-bell`

When set to `On`, GNU `info` attempts to flash the screen instead of ringing the bell. This variable is `Off` by default.

Of course, `info` can only flash the screen if the terminal allows it; in the case that the terminal does not allow it, the setting of this variable has no effect.

However, you can set the `errors-ring-bell` variable to `Off` to make Info perform quietly.

`errors-ring-bell`

When set to `On`, errors cause the bell to ring. The default setting of this variable is `On`.

`gCtrl-compressed-files`

When set to `On`, `info` garbage collects files which had to be uncompressed. The default value of this variable is `Off`. Whenever a node is visited in `info`, the `info` file containing that node is read into core, and `info` reads information about the tags and nodes contained in that file. Once the tags information is read by `info`, it is never forgotten. However, the actual text of the nodes does not need to remain in core unless a particular `info` window needs it. For non-compressed files, the

text of the nodes does not remain in core when it is no longer in use. But decompressing a file can be a time consuming operation, and so `info` tries hard not to do it twice. `gCtrl-compressed-files` tells `info` it is okay to garbage collect the text of the nodes of a file which was compressed on disk.

`show-index-match`

When set to `On`, the portion of the matched search string is highlighted in the message which explains where the matched search string was found. The default value of this variable is `On`. When `info` displays the location where an index match was found, (for more information on `next-index-match`, see “Searching an `info` File” on page 291), the portion of the string that you had typed is highlighted by displaying it in the inverse case from its surrounding characters.

`scroll-behaviour`

Controls what happens when forward scrolling is requested at the end of a node, or when backward scrolling is requested at the beginning of a node. There are three possible values, with `Continuous` being the default variable:

- `Continuous`
Tries to get the first item in this node’s menu, or failing that, the `Next` node, or failing that, the `Next` of the `Up`. This behavior is identical to using the `] (global-next-node)` and `[(global-prev- node)` commands.
- `Next Only`
Only tries to get the `Next` node.
- `Page Only`
Simply gives up, changing nothing. If `scroll-behaviour` is `Page Only`, no scrolling command can change the node that is being viewed.

`scroll-step`

The number of lines to scroll when the cursor moves out of the window. Scrolling happens automatically if the cursor has moved out of the visible portion of the node text when it is time to display. Usually the scrolling is done so as to put the cursor on the center line of the current window. However, if the variable `scroll-step` has a nonzero value, `info` attempts to scroll the node text by that many lines; if that is enough to bring the cursor back into the window, that is what is done. The default value of this variable is 0, thus placing the cursor (and the text it is attached to) in the center of the window. Setting this variable to 1 causes a kind of *smooth scrolling* which some people prefer.

`ISO-Latin`

When set to `On`, `info` accepts and displays ISO Latin characters. By default, `Info` assumes an ASCII character set. `ISO-Latin` tells `info` that it is running in an environment where the European standard character set is in use, and allows you to input such characters to `info`, as well as display them.

3

Making `info` files from Texinfo files

`makeinfo` is the program that builds `info` files from Texinfo files. Before reading this documentation, you should be familiar with reading `info` files. If you want to run `makeinfo` on a Texinfo file prepared by someone else, this documentation contains most of what you need to know. However, to write your own Texinfo files, you should also read *Texinfo (The GNU Documentation)**.

Controlling Paragraph Formats

In general, `makeinfo` *fills* the paragraphs that it outputs to the `info` file. Filling is the process of breaking up and connecting lines such that the output is nearly justified. With `makeinfo`, you can control the following.

- The width of each paragraph (the *fill-column*).
- The amount of indentation that the first line of the paragraph receives (the *paragraph-indentation*).

* *Texinfo (The GNU Documentation Format)* is published by the Free Software Foundation (ISBN 1-882114-12-4).

makeinfo Command Line Options

The following command line options are available for `makeinfo`.

- I *dir*
Adds *dir* to the directory search list for finding files which are included with the `@include` command. By default, only the current directory is searched.
- D *var*
Defines the `texinfo` variable flag; this is equivalent to `@set var` in the Texinfo file.
- U *var*
Makes the `texinfo` variable flag, **var**, undefined; this is equivalent to `@clear var` in the Texinfo file.
- error-limit *num*
Sets the maximum number of errors that `makeinfo` will print before exiting (on the assumption that continuing would be useless). The default number of errors printed before `makeinfo` gives up on processing the input file is 100.
- fill-column *num*
Specifies the maximum right-hand edge of a line. Paragraphs that are filled will be filled to this width. The default value for `fill-column` is 72.
- footnote-style *style*
Sets the footnote style to *style*. *style* should either be `separate` to have `makeinfo` create a separate node containing the footnotes which appear in the current node, or `end` to have `makeinfo` place the footnotes at the end of the current node.
- no-headers
Suppress the generation of menus and node headers. This option is useful together with the `--output file` and `--no-split` options (see following options) to produce a simple formatted file (suitable for printing on a dumb printer) from Texinfo source. If you do not have TEX, these two options may allow you to get readable hard copy.
- no-split
Suppress the splitting stage of `makeinfo`. In general, large output files (where the size is greater than 70k bytes) are split into smaller subfiles, each one approximately 50k bytes.

If you specify `--no-split`, `makeinfo` will not split up the output file.
- no-pointer-validate
- no-validate
Suppress the validation phase of `makeinfo`. Normally, after the file is processed,

some consistency checks are made to ensure that cross references can be resolved, and so forth. See “What Makes a Valid info File?” on page 305.

`--no-warn`

Suppress the output of warning messages. This does not suppress the output of error messages, simply warnings. You might want this if the file you are creating has texinfo examples in it, and the nodes that are referenced don't actually exist.

`--no-number-footnotes`

Suppress the automatic numbering of footnotes. The default is to number each footnote sequentially in a single node, resetting the current footnote number to 1 at the start of each node.

`--output file`

`-o file`

Specify that the output should be directed to `file` instead of the file name specified in the `@setfilename` command found in the Texinfo source. `file` can be the special token `-`, which specifies standard output.

`--paragraph-indent num`

Sets the paragraph indentation to a number, `num`. The value of `num` is interpreted as follows:

- A value of 0 (or none) means not to change the existing indentation (in the source file) at the start of paragraphs.
- A value less than zero means to indent paragraph starts to column zero by deleting any existing indentation.
- A value greater than zero is the number of spaces to leave at the front of each paragraph start.

`--reference-limit num`

When a node has many references in a single Texinfo file, this may indicate an error in the structure of the file. `num` is the number of times a given node may be referenced before `makeinfo` prints a warning message about it (with `@prev`, `@next`, or `@note` appearing in an `@menu`, for example).

`--verbose`

Causes `makeinfo` to inform you as to what it is doing. Normally `makeinfo` only outputs text if there are errors or warnings.

`--version`

Displays the `makeinfo` version number.

What Makes a Valid info File?

If you have not used `--no-pointer-validate` to suppress validation, `makeinfo` will check the validity of the final info file. Mostly, this means ensuring that nodes you

have referenced really exist. What follows is a complete list of what is checked.

- If a node reference such as `Prev`, `Next` or `Up` is a reference to a node in this file, meaning that it is not an external reference such as `(DIR)`; then the referenced node must exist.
- In a given node, if the node referenced by the `Prev` is different than the node referenced by the `Up`, then the node referenced by the `Prev` must have a `Next` which references this node.
- Every node except `Top` must have an `Up` field.
- The node referenced by `Up` must contain a reference to this node, other than a `Next` reference. Obviously, this includes menu items and followed references.
- If the `Next` reference is not the same as the `Next` reference of the `Up` reference, then the node referenced by `Next` must have a `Prev` reference pointing back at this node. This rule still allows the last node in a section to point to the first node of the next chapter.

Defaulting the `Prev`, `Next`, and `Up` Pointers

If you write the `@node` commands in your Texinfo source file without `Next`, `Prev`, and `Up` pointers, `makeinfo` will fill in the pointers from context (by reference to the menus in your source file). Although the definition of an info file allows a great deal of flexibility, there are some conventions that you are urged to follow. By letting `makeinfo` default the `Next`, `Prev`, and `Up` pointers you can follow these conventions with a minimum of effort.

A common error occurs when adding a new node to a menu; often the nodes which are referenced in the menu do not point to each other in the same order as they appear in the menu.

`makeinfo` node defaulting helps with this particular problem by not requiring any explicit information beyond adding the new node (so long as you do include it in a menu). The node to receive the defaulted pointers must be followed immediately by a sectioning command, such as `@chapter` or `@section`, and must appear in a menu that is one sectioning level or more above the sectioning level that this node is to have.

What follows is an example of how to use this feature.

```
@setfilename default-nodes.info
@node Top
@chapter Introduction
@menu
* foo:: the foo node
```

```
* bar:: the bar node
@end menu
@node foo
@section foo
this is the foo node.
@node bar
@section Bar
This is the Bar node.
@bye
```

The previous input produces the following output.

```
Info file default-nodes.info, produced by makeinfo, -*- Text -*-
```

```
from input file default-nodes.texinfo.
```

```
File: default-nodes.info, Node: Top
```

```
Introduction *****
```

```
* Menu:
```

```
* foo:: the foo node
```

```
* bar:: the bar node
```

```
File: default-nodes.info, Node: foo, Next: bar, Up: Top
```

```
foo
```

```
===
```

```
this is the foo node.
```

```
File: default-nodes.info, Node: bar, Prev: foo, Up: Top
```

```
Bar
```

```
===
```

```
This is the Bar node.
```


Index

Symbols

- !, for assembler 25
- ", for assembler 25
- #, for assembler 25
- \$ in symbol names 25
- (info)Help 298
- *, in comments 173
- + or %, as operators 42
- +, line continuation character 173
- , 291
- , in comments 173
- , in macro definitions 60
- , option 8
- , special token, standard output 305
- , standard input file 8
- ., as current address 39
- ., in names 45
- .ABORT 48

- .abort 47, 48
- .app-file 53
- .ascii 49
- .asciz 49
- .balign 49
- .block 66
- .byte 50
- .code32 directive 88, 112
- .comm 50
- .data 50
- .def 50, 51
- .def and .endif directives 40
- .def file 205
- .def/.endif 69
- .desc 51
- .dfloat , for Vax 155
- .dim 40
- .double 51, 87, 153

- .eject 51
- .else 51
- .elseif 51
- .endif 51, 52
- .endif directives 40
- .endif 52, 55
- .endm 59, 60
- .endr 56, 63
- .equ 52
- .exitm 60
- .EXPORT 54
- .extern 53
- .ffloat , for Vax 155
- .file 53
- .fill 53
- .float 54, 87, 153
- .global 54
- .globl 54
- .hword 54
- .ident 55
- .if 55
- .ifdef 55
- .ifndef 56
- .include 56
- .include directive 24
- .int 56, 87
- .irp 56
- .irpc 57
- .lcomm on HPPA 57
- .lflags 57
- .line 40, 47, 58
- .list 59
- .ln 59
- .long 59, 87
- .macro 59
- .mri 60
- .nolist 59, 60
- .octa 61
- .org 61
- .psize 62
- .quad 62, 87
- .rept 63
- .sbttl 63
- .scl 40, 63
- .section 63
- .set 65
- .set on HPPA 65
- .short 65
- .single 54, 65, 87
- .size 40, 65
- .space 66
 - warnings 66
- .space and .subspace directives 32
- .stab 188
- .stab directives 116
- .stab.excl 188
- .stab.index 188
- .stabd 66
- .stabn 66
- .stabs 66
- .string 67
- .subspace directives 32
- .tag 40, 68
- .text 68
- .tfloat 87
- .title 69
- .type 69
- .val 69
- .word 87, 153
- .zero 70
- /* to */ 24
- :, for labels 37
- < 290
 - <, first-node 290
- =, in expressions 38
- > 290
 - >, last-node 290
- ? 38, 297
- ?, object file format specific 176
- @menu, for referencing nodes 305
- @next, for referencing nodes 305
- @node 306
- @note, for referencing nodes 305
- @prev, for referencing nodes 305
- _ 196
- |, for assembler 25
- ‘/’ for directory paths 227

Numerics

- 0, in Info windows 293
- 1...9, in Info windows 293
- 32-bit extender 222
- 80386
 - instruction naming 84
 - jump instructions 86
 - opcode prefixes 85
 - register naming 84
 - registers 86
 - syntax for Intel or AT&T 86
- 80387
 - 16-, 32-, and 64-bit integer formats 87

A

- A 176, 191
- a 176, 185, 193
- a 8
- a, adding new archive files 171
- a, listing 16
- a.out file format 40
- a.out object file format 175
- a.out or b.out 58
- a.out, or b.out, for Intel 80960 12
- a.out-hp300bsd 214
- a.out-sunos-big 214
- abort-key 299
- absolute expression 55
- absolute expressions
 - result, size, value 53
- absolute number 41
- access control element 240
- access control list 240
- Access Control Lists 226
- access tokens 245
- accumulators 82
- ACL 240
- add-digit-to-numeric-arg 299
- ADDLIB 173
- ADDMOD 173
- addr2line 167, 198
- address 181
- addresses 31
- addressing modes 6
- add-section 183
- adjust-section-vma 182
- adjust-start 182
- adjust-vma 182
- adjust-vma= 185
- adjust-warnings 183
- align 48
- alignment 81
- alignment, specified 48
- all 193
- all-header 189
- alphanumerics 196
- AMD
 - 3DNow! instruction set 88, 112
- AMD 29K
 - command-line options 73
 - floating-point numbers 74
 - machine directives 74
 - macros 73
 - opcodes 75
 - register names 73
 - special characters 73
- ANSI C library 228
- ANSI C standard calls 222
- APCS (ARM Procedure Call Standard) 80
- ar 169
 - command line control 169
 - configuring 169
 - controlling with library scripts 172
 - creating indexes to symbols 169
- ar (creates, modifies, extracts archives) 169
- ar -s 190
- ar, brief description 167
- ARC
 - command line options 77
 - floating point 77
 - machine directives 77
- ARC assembler options
 - EB 77
 - EL 77
- architecture selection 214
- architecture values 216
- architecture, defined 216

- architecture= 187
- archive
 - contents 169
 - defined 169
 - extraction 169
 - formats 169
 - generating index of symbols 190
 - members 169
- archive header information 185
- archive-header 185
- archives 167
- argument, for assembly 41
- arguments 42, 285
- arguments, delimited 42
- arithmetic expressions 27
- arithmetic operands 42
- ARM
 - assembler options 79
 - directives 80
 - dlltool 207
 - local labels with 17
- arm 196
- ARM processors 79–82
- assembler
 - !, comment character 25
 - #, comment character 25
 - \$, with symbols 25
 - + or - constants 29
 - , line separator 26
 - , with symbols 25
 - ., escape characters 28
 - . names 45
 - ., addresses 39
 - ., with symbols 25
 - .data 20
 - .eject 16
 - .include directives 17
 - .list 16
 - .nolist 16
 - .psize 16
 - .sbttl 16
 - .text 20
 - .title 16
 - .word 17
 - \ 28
 - |, comment character 25
 - `, escapes 28
 - a.out file format 40
 - a.out or b.out 20
 - absolute expressions 25
 - ad, to omit debugging 16
 - address displacements 20
 - addresses 31
 - ah, high-level language listing 16
 - al, assembly listing 16
 - an, forms preprocessing 16
 - argument 41
 - ARM 79
 - as, symbol table listing 16
 - asm statements 24
 - ASM68K, ASM960 18
 - base-10 flonums 29
 - bignums 29
 - character constants, converting 24
 - characters 28
 - COFF format 40
 - COFF output 20
 - command line options 15–21
 - comment character 25
 - comments 24
 - comments, removing 24
 - conditionals, omitting 16
 - constants 26
 - constants, converting 24
 - converting character constants 24
 - D, for compatibility 16
 - D10V options 119
 - D30V options 123
 - data-section data 20
 - debugging directives 16
 - double-quotes 27
 - error message 21
 - escape characters 28
 - escapes 27
 - exponents, as number constants 29
 - expression 41
 - expressions 29

- f, faster preprocessing 17, 24
 - false conditionals, omitting 16
 - file processing 24
 - floating point number 29
 - flonums 29
 - g, debugging 16
 - global symbols 18
 - hardware floating point 79
 - HPPA targets with -R 20
 - I, adding path 17
 - include file handling 24
 - integers for assembly use 28
 - integers, for processing 29
 - Intel 80960 options 107
 - K, difference tables 17
 - L\$, local labels (HPPA) 17
 - listing output 16
 - listing output directives 16
 - logical line 25
 - macro processing 24
 - MD, dependency tracking 20
 - Microtec 18
 - MIPS options 113
 - Motorola 68000 options 133
 - Motorola 68HC11 or 68HC12 options 136
 - newline character () 26
 - newlines 24, 26
 - o, object file 20
 - object file (-o) 20
 - object file output 21
 - opcodes 79
 - operator 41
 - periods 45
 - picoJava options 151
 - prefix operator (-) 29
 - preprocessing 23
 - processing, using integers 29
 - pseudo-ops 18
 - R, for data-section data 20
 - R, warnings 20
 - searches 17
 - sections 31
 - separators 26
 - SOM format 40
 - standard input 11
 - statements 26
 - statistics 20
 - string 25, 27
 - symbol 25
 - symbol attributes 39
 - symbol type 40
 - symbol value 39
 - syntax 23–29
 - syntax and pseudo-op handling 18
 - THUMB 79
 - updating files 20
 - v, version 21
 - version, finding 21
 - W, warning or error messages 21
 - warning message 21
 - whitespace 23, 24
 - Z, object file output warning 21
 - assembler & linker 31
 - assembler directives 45–70
 - assembly
 - machine-independent syntax 23
 - AT&T
 - instruction naming 84
 - register operands 83
 - AT&T System V/386 assembler syntax 83
 - attributes 40
 - automatic-footnotes 300
 - automatic-tiling 300
 - auxiliary symbol table 40
- B**
- B 176, 191
 - b 181, 186, 214
 - b, adding new archive files 171
 - b, as a command in Info 288
 - b, as a meta-command in Info 288
 - b, backwards 38
 - b.out 48
 - backslash (26
 - backward-char 288
 - backward-word 288
 - beginning-of-line 287

beginning-of-node 288
Berkeley
 size 191
Berkeley sockets 222
bfdname 180
bignum 29, 42
binary file utilities
 target, architecture, linker emulation 213
binary files 179
binary integer 28
binary integers 28
binary utility, defined 169
blocks 31
branch optimization, special instructions 71
bsd 177
BSD 4.2 assembler 23, 28
BSD and SVR4 services 222
bss section 35
bss subsections 35
--byte= 181
--bytes=min-len 193

C

-C 176
C language-specific services, for Cygwin with
 POSIX compliance 234
C++ Annotated Reference Manual 196
C++ function names 177, 196
C++ function overloading 196
c++filt 196
 inverse mapping for linkers 196
c++filt option symbol 197
c++filt symbol 197
c++filt, brief description 167
c, creating new archive files 171
callj 109
canonicalized 214
case 8
case... instruction 153
--change-leading-char 183
character constants 26, 27
class-specific functions, for Cygwin with POSIX
 compliance 234
CLEAR 173

coff 202
COFF format
 auxiliary symbol attributes 40
COFF format output 69
 with .def and .endef 52
COFF output 32
command
 next-line 287
command line options 285
command-line options 8
commands
 key sequence 287
commas 51, 54
commas, in commands 173
common symbols 175
compare-and-branch instructions 110
compiler
 floating point instructions 79
 hardware floating point 79
concatenation 11
config/target.mt 217
configuration triplet 214
configuring 214
Continuous 301
copying and translating object files 167
c-r 291
CREATE 173
create, modify, and extract from archives 167
cross references 292
 followed by a colon 292
 label 292
 menu 292
 menu, followed by * 292
 note 292
 periods within a cross-reference 292
 pointers 292
 Next 292
 Prev 292
 Up 292
 specifying with adjacent colons 292
 target 292
cross-references
 (DIR) 306
 errors in Texinfo files 305

-
- Next 306
 - pointers
 - Next 306
 - Prev 306
 - Up 306
 - Prev 306
 - Up 306
 - c-s 291
 - Ctrl-a 295
 - Ctrl-a, in Info windows 287
 - Ctrl-b 295
 - Ctrl-b, in Info windows 288
 - Ctrl-d 296
 - Ctrl-e 296
 - Ctrl-e, in Info windows 287
 - Ctrl-f 295
 - Ctrl-f, in Info windows 288
 - Ctrl-g 296, 299
 - Ctrl-h 298
 - Ctrl-k 296
 - Ctrl-l 289
 - Ctrl-n 287
 - Ctrl-p 287
 - Ctrl-q 296
 - Ctrl-t 296
 - Ctrl-u 298
 - Ctrl-v 288
 - Ctrl-w 289
 - Ctrl-x, ^ 295
 - Ctrl-x, 0 295
 - Ctrl-x, 1 295
 - Ctrl-x, 2 295
 - Ctrl-x, Delete 296
 - Ctrl-x, t 295
 - Ctrl-y 297
 - ctype functions, ANSI/Cygwin compliance 233
 - cursor, moving in an Info file 287
 - CVTRES 202
 - cygpath 227
 - Cygwin 221
 - .dll files, building example 258
 - access tokens 245
 - ACE (access control element) 240
 - ACL (access control list) 240
 - allocating a handle 276
 - ANSI compliance 233
 - Apache web server 232
 - binary linking 225
 - buffers 278
 - C programming language 234
 - chmod 242, 253
 - chown 242
 - compatibility 233
 - configuration 224
 - console mode applications 255
 - converting a path 276
 - converting a POSIX path to a Win32 path 277
 - converting paths 277
 - CreateProcess call 229
 - CVS 231
 - cygcheck 263
 - cygpath 264
 - CYGWIN environment variable 247
 - cygwin_conv_to_full_win32_path 276
 - cygwin_conv_to_win32_path 277
 - cygwin_detach_dll 277
 - cygwin_getshared 277
 - cygwin_internal 277
 - cygwin_posix_path_list_p 277
 - cygwin_posix_to_win32_path_list 278
 - cygwin_posix_to_win32_path_list_buf_size 278
 - cygwin_win32_to_posix_path_list 278
 - cygwin_win32_to_posix_path_list_buf_size 278
 - cygwin_winpid_to_pid 279
 - debugger 256
 - directory structure 238
 - DLLs, building and using 257
 - dlltool 258
 - ENOSYS 235
 - environment variables 248
 - file descriptors 230
 - file utilities 224
 - fork call 228
 - global symbols 258
 - GNU inetutils 231
-

- GNU profile analysis tool 231
- GPL 224
- group ID 226
- GUI mode applications 255
- HOME environment variable 248
- internal data and functions, accessing 277
- KerbNet 231
- kernel 226
- kill 266
- LD_LIBRARY_PATH environment variable 248
- library 233
- Linux 231
- ls 242
- MAKE_MODE environment variable 248
- Makefile 256
- mapping 243
- Microsoft DLLs 232
- mkgroup 238, 243, 268
- mkpasswd 238, 242, 243, 269
- mount 270, 277
- mount table 250
- mount utility 250
- ncurses 232
- ntsec 238, 250
- options, turned off ("no") 248
- passwd 243
- PATH environment variable 248
- paths (Win32 and POSIX styles) 226
- paths, splitting 278
- performance 230
- PERL 5 232
- permissions 241
- POSIX paths 278
- POSIX signal requirements 230
- POSIX standards 225
- POSIX.1 compliance 234
- POSIX-style path 277
- predefined groups 239
- process IDs 229
- ps 274, 277
- RCS 231
- relative identifier (RID) 239
- SA (security attributes) 240
- SD (security descriptor) 238
- security 238
- semaphore 240
- seteuid 245
- setuid 245
- shared information 277
- shell utilities 224
- shells 232
- SID (security identifiers) 238
- signaling 240
- signals 229, 266
- socket-related calls 230
- ssh 232
- standards 235
- storing inode data 249
- Tcl/Tk 8 232
- TERM environment variable 248
- text utilities 224
- umount 275
- UNC pathnames 227
- Unix select function 230
- Unix standards, matching 223
- utilities 262
- vim editors 231
- Windows resource file 258
- windres 258
- Winsock DLL 230
- X applications 231
- X/Open Release 4 232
- X11R6 client libraries 231
- xemacs and vim editors 231
- CYGWIN environment variable 247
- Cygwin functions 276
- Cygwin library 224
- Cygwin memory structure 277
- Cygwin processes 277
- Cygwin utilities 262
- cygwin_attach_handle_to_fd 276
- cygwin_conv_to_full_posix_path 276
- cygwin_conv_to_full_win32_path 276
- cygwin_conv_to_posix_path 277
- cygwin_conv_to_win32_path 277
- cygwin_detach_dll 277
- cygwin_getshared 277
- cygwin_internal 277

cygwin_posix_path_list_p 277
cygwin_posix_to_win32_path_list 278
cygwin_posix_to_win32_path_list_bu
f_size 278
cygwin_split_path 278
cygwin_win32_to_posix_path_list 278
cygwin_win32_to_posix_path_list_bu
f_size 278
Cyrix 87, 112

D

-D 177, 186
-D 9
-d 186
d 290
-D var 304
d, deleting modules in archive 170
dashes in arguments 170
data objects 175
--debug 201
--debugging 181, 186
debugging 50, 176
symbols 195
debugging information 177, 186
debugging mips 116
debugging symbols 11, 180
debugging symbol-table entries 188
debugging, porting to Win32 hosts 222
--debug-syms 176
DEC mnemonics 153
decimal integer 28
decimal integers 28
DEFAULT_EMULATION 217
defaulting to Makeinfo 306
--defined-only 178
definition file 205
defintion file 205
--defsym 9
DELETE 173
Delete 296
Delete, in Info windows 288
delete-window 295
--demangle 176, 186
demangle 177

demangling 177, 196
demangling low-level symbol names 176
Device-specific functions, for Cygwin with POSIX
compliance 234
difference tables 17
directives, assembler 45–70
DIRECTORY 173
--directory, command line option 286
directory-path 286
dir-node 290
--disassemble 186
--disassemble-all 186
disassembling instructions 188
--discard-all 181, 195
discarded symbols 167
--discard-locals 181, 195
displaying information from object files 167
DLL, creating 205
DLLs 205
DLLs and Cygwin 222
dlltool 205
definition file 205
exports file 206
invocation 205
library file 205, 206
object files 205
dlltool 258
dot (.), in symbol names 38
--dump-config 9
dynamic link libraries 205
dynamic objects 189
dynamic relocation 188
--dynamic, for symbols 177
--dynamic-reloc 188
--dynamic-syms 189

E

-EB 77
echo area 294, 295
completion 297
echo-area-abort 296
echo-area-backward 295
echo-area-backward-kill-line 296

- echo-area-backward-kill-word 296
 - echo-area-backward-word 296
 - echo-area-beg-of-line 295
 - echo-area-complete 297
 - echo-area-delete 296
 - echo-area-end-of-line 296
 - echo-area-forward 295
 - echo-area-forward-word 296
 - echo-area-insert 296
 - echo-area-kill-line 296
 - echo-area-kill-word 296
 - echo-area-newline 296
 - echo-area-possible-completions 297
 - echo-area-quoted-insert 296
 - echo-area-rubout 296
 - echo-area-scroll-completions-window 297
 - echo-area-tab-insert 296
 - echo-area-transpose-chars 296
 - echo-area-yank 297
 - echo-area-yank-pop 297
 - ecoff-littlemips 214
 - EL 77
 - ELF files, 64-bit 210
 - ELF format object files 210
 - ELF output
 - for HPPA 32
 - empty lines of command language 173
 - EMUL 217
 - emulation= 9
 - enable-targets=all 214
 - encoded C++ symbols 167
 - END 174
 - end-of-file 26
 - end-of-line 287
 - end-of-node 288
 - environment variables 214
 - error messages 12
 - error-limit num 304
 - errors-ring-bell 300
 - Esc Ctrl-f 299
 - Esc Ctrl-v 295, 297
 - expression 52
 - absolute numbers 41
 - empty 41
 - expression, for assembly 41
 - extern-only 177
 - EXTRACT 174
- ## F
- F 180, 214
 - f 177, 187
 - f 9
 - f 293
 - F bfdname 194
 - f, forwards 38
 - f, truncating names in archive 171
 - file 286
 - file section sizes and total size 167
 - file-header 187
 - Files and directories for Cygwin with POSIX compliance 234
 - fill-column num 304
 - find-menu 293
 - first-node 290
 - fldt, load temporary real to stack top 87
 - floating point numbers 51, 54
 - flonum 42
 - flonum, defined 29
 - flonums 51, 54
 - on ARC (D, F, R, or S) 29
 - on H8/300, H8/500, Hitachi SH, and AMD 29K (D, F, P, R, S, or X) 29
 - on HPPA (E) 29
 - on Intel 960 (D, F, or T) 29
 - footnote-style style 304
 - for 32-, 64-, and 80-bit formats 87
 - fork call 228
 - format= 177, 191
 - format=compatibility 191
 - format=format 196
 - forward-char 288
 - forward-word 288
 - fstpt, store temporary real and pop stack 87
 - full-contents 188
 - function overloading 196

G

-g 177, 180, 195
g 290
-g, compiler debugging 16
--gap-fill 181
gc-compressed-files 300
--gdwarf2 9
generating index to archive 167
get-help-window 298
get-info-help-node 298
getting started 284
global array 175
global int variable 175
global symbols 18
global-next-node 290
global-prev-node 290
gnu 196
gnu assembler 5
 .o 12
 contributors 163
 introduction 5
 invoking summary, example 8
 -o option 12
 object file formats 7
 output file 12
 pseudo-ops 7
 syntax 7
gnu compiler 196
GNU General Public License (GPL) and Cygwin
 tools 224
GNUTARGET 214
GO32 222
goto-node 290
grow-window 295
--gstabs 9

H

-h 187
h 298
hash table 66
--header 187
headerfile 200
--header-file=headerfile 200

--help 178, 184, 187, 191, 193, 194, 197, 201,
 286
--help 9
hexadecimal integer 28
hexadecimal integers 28
history-node 290
Hitachi h8/300
 addressing modes 91
 command-line options 91
 floating point numbers 92
 machine directives 92
 opcodes 92
 register names 91
 special characters 91
Hitachi h8/500
 addressing modes 94
 command-line options 93
 opcodes 94
 register names 93
 special characters 93
Hitachi SH
 addressing modes 96
 command-line options 95
 floating point numbers 96
 machine-dependent directives 96
 opcodes 96
 register names 95
 special characters 95
HKEY_CURRENT_USER 250
HOME for directory orientation 248
hook functions 217
how to use Info 284
HP9000 Series 800 Assembly Language Reference
 Manual 32
HPPA 17, 99
 assembler directives 100
 assembler syntax 99
 command-line options 99
 floating point numbers 99
 floating-point numbers 100
 local labels with L\$ 17
 SOM and ELF object file formats 99
HPPA targets with -R 20
HPPA targets, labels 26

I

- I 214
- I 9
- i 187
- i 291
- I bfdname 194
- I dir 304
- i interleave 181
- I, for searches 56
- i, inserting new archive files 171
- i386 code, 32-bit 88, 112
- i960
 - command-line options 107
 - compare & branch instructions 109
 - opcodes 109
- ifnotdef 56
- incr 182
- incremental searching 291
- index generating 190
- index table, listing 169
- index, listing archives 190
- index-search 291
- inetutils 231
- infile 180
- infix operators 43
- info 187
- info 283
- Info files, building Texinfo files from 303
- info files, description of 285
- Info, learning to use 284
- info.texi 298
- INFO_PRINT_COMMAND, environment variable 298
- Input and output primitives for Cygwin with POSIX compliance 234
- input, standard 169
- input-target 180, 214
- input-target=bfdname 194
- Insight 221
- instruction operands 42
- instruction set 5
- instructions in hex 188
- instructions in symbolic form 188
- integer expression 42

- integer formats for 80387 87
- integers 28
- Intel
 - MMX instruction set 87, 112
- Intel syntax 83
- Intel x86
 - instruction naming 84
- interleave 181
- interleave=interleave 181
- interleave-1 181
- internal counters 59
- isearch-backward 291
- isearch-forward 291
- ISO-Latin 301
- itbl 10

J

- J 10
- j 187

K

- K 180
- K 10
- K symbolname 195
- K6 and K6-2 processors 87, 112
- Katmai 88, 112
- keep-locals 10
- keep-one-window 295
- keep-symbol= 180
- keep-symbol=symbolname 195
- key symbol 26
- killing and yanking text 296
- kill-node 290

L

- L 10
- l 177, 185
- l 290
- l, modifier for archive files 171
- label 26
- labels 37
 - local 17

-
- last-menu-item 293
 - last-node 290
 - ld sections 33
 - absolute 33
 - bss 33
 - data 33
 - named 33
 - text 33
 - undefined 33
 - LDEMULATION 217
 - leading character 183
 - learning Info 284
 - libraries, shared 188
 - libraries, subroutines 169
 - line comment character
 - 680x0 25
 - AMD 29K 25
 - ARC 25
 - h8/300, h8/500 25
 - Hitachi SH 25
 - HPPA 25
 - i960 25
 - SPARC 25
 - Vax 25
 - z8000 25
 - line comment characters 25
 - linebreaks in Makeinfo 303
 - line-numbers 177, 187
 - linkage symbols 188
 - linker 17
 - default values 217
 - emulation 217
 - m 217
 - specifying for input target 215
 - specifying for target 215
 - linker & assembler 31
 - linker emulation 213
 - linker input architecture 216
 - linker input target 215
 - linker output architecture 216
 - linker=linker 201
 - linking
 - absolute or a relative pathnames 201
 - faster links to library 190
 - LIST 174
 - listing printable strings from files 167
 - listing symbols from object files 167
 - listing-cont-lines 10
 - listing-lhs-width 10
 - listing-lhs-width2 10
 - listing-rhs-width 10
 - lists 213
 - list-visited-nodes 291
 - literal characters 28
 - local symbol names 38
 - local symbols 38
 - locating a line of input 12
 - location counter 35, 61
 - active 35
 - logical file 25
 - logical files 12
 - ls 185
 - lucid 196
 - Lucid compiler 196
- ## M
- M 169
 - M 10
 - m 187
 - m, moving members in archive 170
 - M, option for controlling ar script 172
 - M2 processor 87, 112
 - m68k
 - options 19
 - machine 187
 - machine instruction sets 71
 - macros, defining 59
 - make 20
 - Makefile 217
 - Makeinfo command line options 304
 - Makeinfo output, controlling 303
 - mangling 196
 - manipulating variables 299
 - math functions, ANSI/Cygwin compliance 234
 - MD 10
 - Memory management for Cygwin with POSIX compliance 235
-

- menu items as options 286
 - menu-digit 293
 - menu-item 293
 - Meta- 288
 - Meta-> 288
 - Meta-1 299
 - Meta-2... Meta-9 299
 - Meta-b 296
 - Meta-d 296
 - Meta-Delete 296
 - Meta-f 296
 - Meta-f, in Info windows 288
 - Meta-Tab 296
 - Meta-Tab, in Info windows 293
 - Meta-v 288
 - Meta-x 294
 - Meta-x describe-command 298
 - Meta-x describe-key 298
 - Meta-x describe-variable 298, 300
 - Meta-x set-screen-height 299
 - Meta-x set-variable 299
 - Meta-x where-is 298
 - Meta-y 297
 - Microsoft Windows
 - mapping 243
 - Microtec 18
 - min-len 193
 - MIPS
 - options 113
 - mips 113
 - (ISA) Instruction Set Architecture 116
 - assembling in 32-bit mode 117
 - debugging 116
 - ECOFF targets 116
 - r2000 113
 - r3000 113
 - r4000 113
 - r6000 113
 - mod 170
 - mode line 294
 - modifiers 169
 - modifiers for p 171
 - Motorola
 - syntax 137
 - Motorola 680x0
 - addressing modes 137
 - compatibility with Sun assembler 138
 - opcodes 138
 - options 134
 - pseudo opcodes 138
 - special characters 140
 - syntax 135
 - mount 270
 - mount table 250
 - mount utility 250
 - move-to-next-xref 293
 - move-to-prev-xref 293
 - mr i 10
 - MRI assembler
 - complex relocations 18
 - global symbols 18
 - specifying pseudo-ops 18
 - MRI compatibility 174
 - MRI librarian program 169
- N**
- N 181
 - n 177, 196
 - n 289
 - N symbolname 195
 - name 187
 - NetWare Loadable Module 200
 - Netware Loadable Module (NLM) 167
 - networking functions for Cygwin 235
 - new-lc 61
 - Next Only 301
 - next-index-match 291
 - next-line 287
 - next-node 289
 - NLM Development and Tools Overview 201
 - NLM Software Developer's Kit 201
 - NLM targets 200
 - nlmconv 167, 200
 - converting relocatable object files to NetWare Loadable Modules 200
 - version 201
 - NLMLINK 200
 - nm 175, 196

- common symbols as uninitialized data 175
 - linking 175
 - specifying for architecture 216
 - specifying for target 215
 - symbol type 175
 - symbol value 175
 - uninitialized data section 175
 - nm --print-armap 169, 190
 - nm -s 169, 190
 - nm, brief description 167
 - nm, listing symbols from object files 175
 - no-adjust-warnings 183
 - node 286
 - node
 - Top 306
 - Up 306
 - node, selection of 288
 - no-demangle 177
 - nodes, description of 284
 - no-headers 304
 - non-global symbols in source files 181
 - no-number-footnotes 305
 - no-pointer-validate 304
 - no-sort 177
 - no-split 304
 - no-strip-underscores 196
 - no-validate 304
 - Novell 201
 - no-warn 305
 - no-warn 11
 - ntsec 250
 - ntsec 238, 250
 - number constants 28
 - bignums 28
 - flonums 28
 - integers 28
 - numeric expressions 27
 - numeric-sort 177
- O**
- O 180
 - o 176, 193
 - o 10
 - O bfdname 194, 200
 - o file 305
 - o, preserving member dates of archive files 171
 - Oasys compilers 186
 - objcopy 179
 - change-addresses 182
 - change-section-address 182
 - change-section-lma 182
 - change-section-vma 182
 - change-start 182
 - change-warnings 183
 - no-chagne-warnings 183
 - specifying for architecture 216
 - specifying for input target 214
 - specifying for output target 214
 - S-records 179
 - temporary files 179
 - objcopy, brief description 167
 - objcopy, for copying contents of object files 179
 - objcopy, leading char 183
 - objdump 185
 - displaying data for object files 185
 - specifying for architecture 216
 - objdump -h 187
 - objdump -i 216
 - objdump, brief description 167
 - object-code format 186
 - objfile 191
 - objfile ... 191
 - octa- (16 bytes), defined 61
 - octal integer 28
 - octal integers 28
 - offsets in an address 43
 - ofmt 215
 - OPEN 173, 174
 - operand 42
 - operation 169
 - operator, for assembly 41
 - options to objcopy 184
 - outfile 180
 - output 286
 - output file 305
 - output file 12
 - output files 174

output to a file 286
OUTPUT_ARCH 216
OUTPUT_FORMAT 215
--output-target= 180
--output-target=bfdname 194, 200
overloaded functions 196

P

-P 177
-p 177
p 169, 289
p, specifying archive members to output file 170
--pad-to 181
Page Only 301
--paragraph-indent num 305
paragraphs, controlling formats with
 Makeinfo 303
PE format 222
Pentium II processors 87, 112
Pentium processors 87, 112
periods, in assembler directives 45
physical files 12
PID 274
PIDs and Cygwin 229
--portability 177
porting layer for UNIX applications 221
porting UNIX compiler tools to Win32 222
posix 177
POSIX path 278
POSIX.1/90 calls 222
POSIX.12 223
POSIX.2 standard output format 177
prefix operators 42
prev-line 287
prev-node 289
prev-window 294
--print-armap 177
--print-file-name 176, 193
printing 297, 298
printing character 296
Process environment for Cygwin 234
process ID 274, 279
Process primitives for Cygwin 234

process status 274
program version 193
ps program 274
pseudo-ops 7
PTY calls 224

Q

q 299
q, quick appends to archive 170
q, update operation 169
quit 299
quotes 53
quotes, escape characters 28

R

-R 180, 188
-R 10
-r 178, 188
r 293
-R sectionname 194
r, replacing member... into archive 170
radix 178
--radix= 178, 192
--radix=number 192
--radix=radix 193
ranlib 169, 190
ranlib, brief description 167
rc 202
readelf 210
 ELF format object files 210
 ELF header 210
 examining 64 bit ELF files 210
 examining archives 210
 object files 210
 options 210
redraw-display 289
--reference-limit num 305
registers 6, 82
relative identifier (RID) 239
--reloc 188
relocatable object modules 169
relocation 31, 188
relocation and symbol information 180

relocation entry 177
relpos argument, before archive specification 171
--remove-leading-char 183
--remove-section= 180
--remove-section=sectionname 194
res 202
Resource Compiler 202
result 53
--reverse-sort 178
RID (relative identifier) 239
run-time addresses 31

S

-S 180, 188, 195
-s 177, 188, 194
s 169, 291
-s format 196
s, writing object-file index to archive 172
SAVE 173, 174
scroll-backward 288
scroll-behavior 289, 290
scroll-behaviour 301
scroll-forward 288
scrolling through node structure 288
scroll-other-window 295
scroll-step 301
SD (security descriptor) 238
search 291
search through indices of info files 291
searching 291
 incremental 291
section 31
 location counter 35
 offsets in address 43
 undefined, undefined U 33
section addresses 185
section headers 187
section sizes 191
--section= 187
section=flags 183
--section-header 187
sectionname 180, 194
sectionname=filename 183

sections
 bss 32
 data 32
 internal use 34
 text 32
 text and data 34
sections, assembly addresses 31
security attributes 240
security descriptor (SD) 238
security identifiers (SID) 238
select-reference-this-line 293
select-visited-node 291
--set-section-flags 183
--set-start 182
show-footnotes 299
show-index-match 301
--show-raw-insn 188
SID (security identifiers) 238
signals functions, ANSI/Cygwin
 compliance 233
size 53, 191
 command line options 191
 formats 192
 specifying for architecture 216
 specifying for target 215
size, brief description 167
--size-sort 178
sockets 222
Solaris 2.0 188
SOM format for HPPA 40
SOM or ELF output
 for HPPA 32
--source 188
source 11
source file information 180
source files
 non-global symbols 181
source, source program 11
SPACEBAR 297
SPACEBAR, in Info windows 288
SPARC
 floating point numbers 151
 machine directives 151
 options 149

- v6, v7, v8, v8+, v9, v9a 149
 - sparclite 149
 - split-window 295
 - S-record 180
 - S-records 186, 187
 - stabs 188
 - standard mnemonics 5
 - standard output streams 174
 - start address 182
 - start-address= 188
 - statistics 10
 - stdio functions, ANSI/Cygwin compliance 233
 - stdlib functions, ANSI/Cygwin compliance 233
 - stop-address= 188
 - storage boundary 61
 - string 53
 - string constants, literals 27
 - string functions, ANSI/Cygwin compliance 233
 - strings 193
 - determining contents of non-text files 193
 - printing character sequences 193
 - specifying for architecture 216
 - specifying for target 215
 - strings, brief description 167
 - strip 194
 - modifying files named in arguments 194
 - specifying for input target 214
 - specifying for output target 214
 - version 195
 - strip -v 195
 - strip, brief description 167
 - strip-all 180, 194
 - strip-debug 180, 195
 - strip-local-absolute 10
 - strip-symbol= 181
 - strip-symbol=symbolname 195
 - strip-underscores 196
 - strip-unneeded 180
 - StrongARM 79
 - assembler options 79
 - directives 80
 - subnodes 286
 - subsections 34
 - Sun
 - compatibility with Motorola 680x0 135
 - SVR4 and BSD services 222
 - symbol
 - debugging 176
 - descriptor 51
 - descriptors with .desc statements 40
 - global (external) 175
 - in the text (code) section 176
 - name 176
 - name setting 40
 - read only 176
 - small objects 176
 - type 40, 175
 - undefined 176
 - weak 176
 - with ld 54
 - symbol characters, using 25
 - symbol information 177
 - symbol names 186
 - symbol table and relocation entries 189
 - symbolic debuggers 66
 - symbolname 180, 195
 - symbols 42
 - members 190
 - non-global, removing 195
 - processing 195
 - relocatable object files 190
 - removing 194
 - value 39
 - symbols as uninitialized data 175
 - symbols, auxiliary attributes 39
 - syms 188
 - System databases for Cygwin with POSIX compliance 234
 - System V
 - size 191
 - sysv 177
- T**
- T 189
 - t 188
 - t 10

-
- t 290
 - T headerfile 200
 - t radix 178, 193
 - t, table, listing contents of archive 170
 - t, temporary real 87
 - Tab 297
 - Tab, in Info windows 293
 - target 214
 - target
 - object file format 214
 - supporting multiple architectures 214
 - target system, specifying 213
 - target= 180, 186, 192
 - target=bfdname 192, 193, 194
 - Tbss 187
 - Tcl/Tk 232
 - Tdata 187
 - text section 20
 - THUMB
 - assembler 79
 - instructions with dlltool 207
 - tiling, automatic 295
 - time functions, ANSI/Cygwin compliance 233
 - toggle-wrap 289
 - top-node 290
 - traditional-format 11
 - triplet 214
 - Ttext 187
 - TTY/PTY calls 224
- U**
- u 178
 - u 289
 - U var 304
 - u, replaces new archive files 172
 - undefined section 33
 - undefined symbol 177
 - undefined-only 178
 - underscore 183, 186
 - underscore (_), in symbol names 38
 - underscores 196
 - universal-argument 298
 - UNIX applications, porting to Windows 221
 - unprintable character 193
 - up-node 289
- V**
- v 178, 184, 190
 - v 21, 177, 184, 190, 193
 - v 11
 - v, for verbose version 172
 - V, version numbering 172
 - val 182
 - valid Info file 305
 - value 53
 - variables, manipulating 299
 - VAX 186
 - Vax
 - command line options 154
 - conversion of floating point numbers 153
 - debugging 154
 - displacement sizing character 153
 - floating point constants 155
 - JUMPify, for longer branches 154
 - JUMPs 154
 - opcodes, pseudo opcodes 153
 - options for VMS 154
 - register names 153
 - symbol table 154
 - temporary file generation 154
 - Vax bit-fields 23
 - VERBOSE 173, 174
 - verbose 184, 195, 305
 - version 178, 184, 189, 193, 195, 197, 201, 286, 305
 - version 11
 - version 21
 - version 11
 - version 190
 - version information 286
 - view-file 291
 - visible-bell 300
 - Visual C++ to port GDB 222
- W**
- w 11

- Wa, passing arguments 16
 - warn 11
 - warnings 12
 - Web support site iii
 - whitespace 41
 - whitespace, in commands 173
 - wide 189
 - Win32
 - API calls 225
 - path with Cygwin 277
 - paths 278
 - porting UNIX compiler tools 222
 - windows
 - automatic-tiling 295
 - in Info, manipulating 293
 - manipulating 295
 - multiple 293
 - Windows 95/98
 - lack of operating system security 226
 - Windows and Cygwin 221
 - Windows registry 250
 - Windows Resource Compiler 202
 - Windows utilities 202
 - windres 202
 - WINPID 274
 - .p2align 61
- ## X
- X 181, 195
 - x 181, 195
 - x 171
 - x, extracting members from archive 171
 - x86
 - assembly directives 86
 - instruction naming 84
 - jump instructions 86
 - machine dependent options 89
 - opcode prefixes 85
 - syntax for Intel or AT&T 86
 - xrefs 292
- ## Y
- yanking text 296
- ## Z
- z 11