Data General

Customer Documentation

Using GNU CC

AViiON™
PRODUCT LINE

Using GNU CC

069-100317-01

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 069-100317
Copyright © Free Software Foundation 1988, 1989
Copyright © Data General Corporation 1990
All Rights Reserved
Printed in the United States of America
Rev. 01, April 1990

NOTICE

DATA GENERAL CORPORATION (DGC) HAS PREPARED AND/OR HAS DISTRIBUTED THIS DOCUMENT FOR USE BY DGC PERSONNEL, LICENSEES, CUSTOMERS, AND PROSPECTIVE CUSTOMERS. THE INFORMATION CONTAINED HEREIN IS THE PROPERTY OF THE COPYRIGHT HOLDER(S); AND THE CONTENTS OF THIS MANUAL SHALL NOT BE REPRODUCED IN WHOLE OR IN PART NOR USED OTHER THAN AS ALLOWED IN THE APPLICABLE LICENSE AGREEMENT.

The copyright holders reserve the right to make changes in specifications and other information contained in this document without prior notice, and the reader should in all cases determine whether any such changes have been made.

THE TERMS AND CONDITIONS GOVERNING THE SALE OF DGC HARDWARE PRODUCTS AND THE LICENSING OF DGC SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE WRITTEN CONTRACTS BETWEEN DGC AND ITS CUSTOMERS, AND THE TERMS AND CONDITIONS GOVERNING THE LICENSING OF THIRD PARTY SOFTWARE CONSIST SOLELY OF THOSE SET FORTH IN THE APPLICABLE LICENSE AGREEMENT. NO REPRESENTATION OR OTHER AFFIRMATION OF FACT CONTAINED IN THIS DOCUMENT INCLUDING BUT NOT LIMITED TO STATEMENTS REGARDING CAPACITY, RESPONSE-TIME PERFORMANCE, SUITABILITY FOR USE OR PERFORMANCE OF PRODUCTS DESCRIBED HEREIN SHALL BE DEEMED TO BE A WARRANTY BY DGC FOR ANY PURPOSE, OR GIVE RISE TO ANY LIABILITY OF DGC WHATSOEVER.

IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS DOCUMENT OR THE INFORMATION CONTAINED IN IT, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY OF SUCH DAMAGES.

All software is made available solely pursuant to the terms and conditions of the applicable license agreement which governs its use.

AViiON is a trademark of Data General Corporation.

Restricted Rights Legend: Use, duplication, or disclosure by the U. S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at [FAR] 52.227-7013 (May 1987).

Data General Corporation
4400 Computer Drive
Westboro, MA 01580

Using GNU CC
069-100317-01
069-100322-01 (Japan only)

Revision History: Effective with:

Original Release - October, 1989
First Revision - April, 1990 GNU CC, Revision 1.35.21

This revision involves only the addition of an index.

Using GNU CC

Excerpted from
Using and Porting GNU CC
by Richard M. Stallman
August 17, 1989

Copyright c 1988, 1989 Free Software Foundation, Inc.

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

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the section entitled "GNU General Public License" is included exactly as in the original, and provided 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 manual into another language, under the above conditions for modified versions, except that the section entitled "GNU General Public License" and this permission notice may be included in translations approved by the Free Software Foundation instead of in the original English.

Preface

This manual, excerpted from *Using and Porting GNU CC* by Richard M. Stallman, documents the version of GNU CC that runs on Data General's AViiON™ systems. It includes all information from the original that will be immediately useful to the programmer on the AViiON platform, while eliminating information that is specific to other platforms or to the internals of the compiler. The unabridged manual is available with the release of GNU CC, in the file /usr/lib/gcc/gcc.ls.

This manual describes GNU CC command options (some of them unique to AViiON systems), problems and incompatibilities that have been known to affect the programmer familiar with other compilers, and extensions to the C language that are available in GNU CC.

Appendix A, the GNU General Public License, specifies terms for distribution of the GNU C compiler. Note that the General Public License does not in any way restrict the distribution of software compiled with GNU CC on AViiON systems.

Sources for GNU C and its associated documentation are available on the AViiON Contributed Software distribution (Model #R006AZN20A) from Data General Corporation.

End of Preface

Contents

Chapter 1 GNU CC Command Options

Chapter 2 Known Causes of Trouble with GNU CC

Chapter 3 Incompatibilities of GNU CC

Chapter 4 GNU Extensions to the C Language

Statements and Declarations inside of Expressions 4–1
Naming an Expression's Type 4–2
Referring to a Type with typeof 4–2
Generalized Lvalues 4–3
Conditional Expressions with Omitted Middle–Operands 4–4
Arrays of Length Zero 4–5
Arrays of Variable Length 4–5
Non–Lvalue Arrays May Have Subscripts 4–6
Arithmetic on void–Pointers and Function Pointers 4–6
Non–Constant Initializers 4–6
Constructor Expressions 4–6
Declaring Attributes of Functions 4–7
Dollar Signs in Identifier Names 4–8
Inquiring about the Alignment of a Type or Variable 4–8
An Inline Function is As Fast As a Macro 4–8
Assembler Instructions with C Expression Operands 4–9
Controlling Names Used in Assembler Code 4–12
Global Variables in Registers 4–12
Alternate Keywords 4–14

Appendix A GNU General Public License

Preamble A–1
Terms and Conditions A–2
No Warranty A–3
How to Apply These Terms to Your New Programs A–4

Index

Chapter 1
GNU CC Command Options

The GNU C compiler uses a command syntax much like the UNIX C compiler. The gcc program accepts options and file names as operands. Multiple single–letter options may *not* be grouped: '–dr' is very different from '–d –r'.

When you invoke GNU CC, it normally does preprocessing, compilation, assembly and linking. File names which end in '.c' are taken as C source to be preprocessed and compiled; file names ending in '.i' are taken as preprocessor output to be compiled; compiler output files plus any input files with names ending in '.s' are assembled; then the resulting object files, plus any other input files, are linked together to produce an executable.

Command options allow you to stop this process at an intermediate stage. For example, the '–c' option says not to run the linker. Then the output consists of object files output by the assembler.

Other command options are passed on to one stage of processing. Some options control the preprocessor and others the compiler itself. Yet other options control the assembler and linker; these are not documented here, but you rarely need to use any of them.

Here are the options to control the overall compilation process, including those that say whether to link, whether to assemble, and so on.

'–o *file*'
: Place output in file *file*. This applies regardless to whatever sort of output is being produced, whether it be an executable file, an object file, an assembler file or preprocessed C code. If '–o' is not specified, the default is to put an executable file in 'a.out', the object file '*source*.c' in '*source*.o', an assembler file in '*source*.s', and preprocessed C on standard output.

'–c'
: Compile or assemble the source files, but do not link. Produce object files with names made by replacing '.c' or '.s' with '.o' at the end of the input file names. Do nothing at all for object files specified as input.

'–S'
: Compile into assembler code but do not assemble. The assembler output file name is made by replacing '.c' with '.s' at the end of the input file name. Do nothing at all for assembler source files or object files specified as input.

'–E'
: Run only the C preprocessor. Preprocess all the C source files specified and output the results to standard output.

'–v'
: Compiler driver program prints the commands it executes as it runs the preprocessor, compiler proper, assembler and linker. Some of these are directed to print their own version numbers.

'–V'
: Print the version numbers of the compiler driver, preprocessor, compiler proper, assembler, and linker.

'-pipe' Use pipes rather than temporary files for communication between the various stages of compilation.

'-B*prefix*' Compiler driver program tries *prefix* as a prefix for each program it tries to run. These programs are 'cpp', 'cc1', 'as' and 'ld'.

For each subprogram to be run, the compiler driver first tries the '-B' prefix, if any. If that name is not found, or if '-B' was not specified, the driver tries two standard prefixes, which are '/usr/lib/gcc-' and '/usr/local/lib/gcc-'. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your 'PATH' environment variable.

The run-time support file 'gnulib' is also searched for using the '-B' prefix, if needed. If it is not found there, the two standard prefixes above are tried, and that is all. The file is left out of the link if it is not found by those means. Most of the time, on most machines, you can do without it.

You can get a similar result from the environment variable 'GCC_EXEC_PREFIX'; if it is defined, its value is used as a prefix in the same way. If both the '-B' option and the 'GCC_EXEC_PREFIX' variable are present, the '-B' option is used first and the value of the environment variable second.

If the environment variable 'TMPDIR' is defined, GNU CC places all of the compiler's temporary files in the directory specified by 'TMPDIR'.

These options control the details of C compilation itself.

'-ansi' Support all ANSI standard C programs. This turns off certain features of GNU C that are incompatible with ANSI C, such as the asm, inline and typeof keywords. Predefined macros such as unix and m88k that identify the type of system you are using, but which do not begin with two leading underscore characters, are not defined. It also enables the rarely used ANSI trigraph feature.

The alternate keywords __asm, __inline and __typeof continue to work when you select '-ansi'. You would not want to use them in an ANSI C program, of course, but it is useful to put them in header files that might be included in compilations done with '-ansi'. Alternate predefined macros such as __unix__ are also available, with or without '-ansi'.

The '-ansi' option does not cause non-ANSI programs to be rejected gratuitously. For that, '-pedantic' is required in addition to '-ansi'.

The macro __STRICT_ANSI__ is predefined when the '-ansi' option is used. Some header files may notice this macro and refrain from declaring certain functions or defining certain macros that the ANSI standard doesn't call for; this is to avoid interfering with any programs that might use these names for other things.

'-traditional' Attempt to support some aspects of traditional C compilers. Specifically:

- All extern declarations take effect globally even if they are written inside of a function definition. This includes implicit declarations of functions.
- The keywords typeof, inline, signed, const and volatile are not recognized.
- Comparisons between pointers and integers are always allowed.
- Integer types unsigned short and unsigned char promote to unsigned int.
- Out-of-range floating point literals are not an error.
- All automatic variables not declared register are preserved by longjmp. Ordinarily, GNU C follows ANSI C: automatic variables not declared volatile may be clobbered.
- In the preprocessor, comments convert to nothing at all, rather than to a space. This allows traditional token concatenation.
- In the preprocessor, macro arguments are recognized within string constants in a macro definition (and their values are stringified, though without additional quote marks, when they appear in such a context). The preprocessor always considers a string constant to end at a newline.
- The predefined macro __STDC__ is not defined when you use '-traditional', but __GNUC__ is (since the GNU extensions which __GNUC__ indicates are not affected by '-traditional'). If you need to write header files that work differently depending on whether '-traditional' is in use, by testing both of these predefined macros you can distinguish four situations: GNU C, traditional GNU C, other ANSI C compilers, and other old C compilers.
- String literals are put into the writable data section instead of into read-only text.

'-O' Optimize. Optimizing compilation takes somewhat more time, and a lot more memory for a large function. Without '-O', the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. Statements are independent: if you stop the program with a breakpoint between statements, you can then assign a new value to any variable or change the program counter to any other statement in the function and get exactly the results you would expect from the source code.

Without '-O', only variables declared register are allocated in registers. The resulting compiled code is a little worse than produced by PCC without '-O'.

With '-O', the compiler tries to reduce code size and execution time.

Some of the '-f' options described below turn specific kinds of optimization on or off.

'-g'
Produce debugging information in the operating system's native format. Unlike most other C compilers, GNU CC allows you to use '-g' with '-O'. The shortcuts taken by optimized code may occasionally produce surprising results: some variables you declared may not exist at all; flow of control may briefly move where you did not expect it; some statements may not be executed because they compute constant results or their values were already at hand; some statements may execute in different places because they were moved out of loops. Nevertheless it proves possible to debug optimized output. This makes it reasonable to use the optimizer for programs that might have bugs.

'-w'
Inhibit all warning messages.

'-W'
Print extra warning messages for these events:

- An automatic variable is used without first being initialized. These warnings are possible only in optimizing compilation, because they require data flow information that is computed only when optimizing. If you don't specify '-O', you won't get these warnings.

  These warnings occur only for variables that are candidates for register allocation. Therefore, they do not occur for a variable that is declared `volatile`, or whose address is taken, or whose size is other than 1, 2, 4 or 8 bytes. Also, they do not occur for structures, unions or arrays, even when they are in registers.

  Note that there may be no warning about a variable that is used only to compute a value that itself is never used, because such computations may be deleted by data flow analysis before the warnings are printed.

  These warnings are made optional because GNU CC is not smart enough to see all the reasons why the code might be correct despite appearing to have an error. Here is one example of how this can happen:

```
{
  int x;
  switch (y) {
    case 1: x = 1;
      break;
    case 2: x = 4;
      break;
    case 3: x = 5;
  }
  foo (x);
}
```

  If the value of y is always 1, 2 or 3, then x is always initialized, but GNU CC doesn't know this. Here is another common case:

```
{
   int save_y;
   if (change_y) save_y = y, y = new_y;
   ...
   if (change_y) y = save_y;
}
```

This has no bug because save_y is used only if it is set.

Some spurious warnings can be avoided if you declare as `volatile` all the functions you use that never return. See the function attributes section in Chapter 4.

- A nonvolatile automatic variable might be changed by a call to `longjmp`. These warnings as well are possible only in optimizing compilation. The compiler sees only the calls to `setjmp`. It cannot know where `longjmp` will be called; in fact, a signal handler could call it at any point in the code. As a result, you may get a warning even when there is in fact no problem because `longjmp` cannot in fact be called at the place which would cause a problem.

- A function can return either with or without a value. (Falling off the end of the function body is considered returning without a value.) For example, this function would evoke such a warning:

```
foo (a) {
   if (a > 0)
      return a;
}
```

  Spurious warnings can occur because GNU CC does not realize that certain functions (including `abort` and `longjmp`) will never return.

- An expression-statement contains no side effects.

In the future, other useful warnings may also be enabled by this option.

'`-Wimplicit`'
Warn whenever a function is implicitly declared.

'`-Wreturn-type`'
Warn whenever a function is defined with a return-type that defaults to `int`. Also warn about any `return` statement with no return-value in a function whose return-type is not `void`.

'`-Wunused`'
Warn whenever a local variable is unused aside from its declaration, and whenever a function is declared `static` but never defined.

'`-Wshadow`'
Warn whenever a local variable shadows another local variable.

'`-Wid-clash-`*len*'
Warn whenever two distinct identifiers match in the first *len* characters. This may help you prepare a program that will compile with certain obsolete compilers.

'-Wswitch'
: Warn whenever a switch statement has an index of enumeral type and lacks a case for one or more of the named codes of that enumeration. (The presence of a default label prevents this warning.) Outside the enumeration range, case labels also provoke warnings when this option is used.

'-Wcomment'
: Warn whenever a comment-start sequence '/*' appears in a comment.

'-Wtrigraphs'
: Warn if any trigraphs are encountered (assuming they are enabled).

'-Wall'
: All of the above '-W' options combined.

'-Wcast-qual'
: Warn whenever a pointer is cast so as to remove a type qualifier from the target type. For example, warn if a const char * is cast to an ordinary char *.

'-Wwrite-strings'
: Give string constants the type const char[*length*] so that copying the address of one into a non-const char * pointer will get a warning. These warnings will help you find at compile time code that can try to write into a string constant, but only if you have been very careful about using const in declarations and prototypes. Otherwise, it will just be a nuisance; this is why we did not make '-Wall' request these warnings.

'-p'
: Generate extra code to write profile information suitable for the analysis program prof.

'-pg'
: Generate extra code to write profile information suitable for the analysis program gprof.

'-a'
: Generate extra code to write profile information for basic blocks, suitable for the analysis program tcov. Eventually GNU gprof should be extended to process this data.

'-l*library*'
: Search a standard list of directories for a library named *library*, which is actually a file named 'lib*library*.a'. The linker uses this file as if it had been specified precisely by name.

  The directories searched include several standard system directories plus any that you specify with '-L'.

  Normally the files found this way are library files---archive files whose members are object files. The linker handles an archive file by scanning through it for members which define symbols that have so far been referenced but not defined. But if the file that is found is an ordinary object file, it is linked in the usual fashion. The only difference between using an '-l' option and specifying a file name is that '-l' searches several directories.

'-L*dir*'
: Add directory *dir* to the list of directories to be searched for '-l'.

'-nostdlib'
: Don't use the standard system libraries and startup files when linking. Only the files you specify (plus 'gnulib') will be passed to the linker.

'-m*machinespec*'
Machine-dependent option specifying something about the type of target machine. These options are defined by the macro TARGET_SWITCHES in the machine description. The default for the options is also defined by that macro, which enables you to change the defaults.

These are the '-m' options defined in the 88000 machine description:

'-mblock-move'
: Enable the compiler to generate inline code for copying structures and for calls to the builtin functions _builtin_memcpy and _builtin_strcpy where it is reasonable to do so. In addition, if optimization is specified with -O, memcpy is defined to be _builtin_memcpy, and strcpy is defined to be _builtin_strcpy. This is the default behavior.

'-mno-block-move'
: Do not attempt to do block moves with inline code.

'-mcheck-zero-division'
: Emit code to check if an integer division by zero occurs and issue trap number 503 if it occurs. The current 88000 processors do not reliably check the dividend for zero. This is the default behavior of the GNU C compiler.

'-mno-check-zero-division'
: Do not emit code to check if an integer division by zero occurs and issue trap number 503 if it occurs.

'-mconst-uses-data'
: Put constant data into the data section rather than the text section. Use this option if your linker does not perform literal synthesis on constants within the text section and you use the '-mno-literal-synthesis' option; if your system does not support read access to the text section; or if you are doing some post-processing on the assembly language with tools that don't understand about constants going in the text area.

'-mno-const-uses-data'
: Same as the '-mconst-uses-text' option.

'-mconst-uses-text'
: Put constant data into the text section. This option is on by default.

'-mdelay-slot'
: Attempt to reorder instructions to make use of the delay slot following branch and subroutine call instructions (i.e., try to use a 'jsr.n' instruction instead of a 'jsr' instruction). This is on by default if you are optimizing.

'-mno-delay-slot'
: Do not reorder instructions to take advantage of the delay slot following branch and subroutine call instructions, to make some forms of debugging easier.

'-mdouble-memory-refs'

Assume that it is ok to use the ld.d/st.d instructions (some revisions of the silicon do not support these instructions).

'-mno-double-memory-refs'

Do not use the ld.d/st.d instructions. This is on by default.

'-mhandle-large-shift'

Emit a four instruction sequence for each shift by a non–constant amount if the shift count is less than zero or greater than 31. Logical shifts and arithmetic shifts left produce a 0 result if the shift count is out of bounds. Arithmetic shifts right produce a copy of the sign bit if the shift count is out of bounds. The ANSI standard for C specifies that shifts outside of the range of 0 to number_bits – 1 is undefined. It is an error to specify both '-mtrap-large-shift' and '-mhandle-large-shift'.

'-mno-handle-large-shift'

Emit a single instruction to handle all shifts (unless '-mtrap-large-shift' is specified).

'-midentify-revision'

Emit an assembly 'ident' directive which gives the filename, date, time, and compiler revision, for use with the 'what' command.

'-mliteral-synthesis'

Emit two or three instruction sequences to reference static data. This is on by default. If selected, register 'r13' is used as a scratch register to build up addresses with the 'or.u' instruction, and is unavailable for normal allocation by the compiler.

'-mno-literal-synthesis'

Emit one instruction sequence to reference static data, assuming that the linker or assembler will patch up these references to use the appropriate linker registers ('r26' – 'r29'). Most linkers for 88000 machines will not do this optimization.

'-mocs-debug-info'

Put out additional debug information to comply with the proposed '88Open OCS' text description information.

'-mno-ocs-debug-info'

Do not put out any additional debugging information.

'-mocs-frame-position'

When emitting COFF debug information for automatic variables and parameters stored on the stack, the offset written out is from the OCS canonical frame pointer (CFA), which is the stack pointer (register 'r31') when the function is entered. This is specified by the 88Open OCS (Object Compatibility Standard), but not all of the current debuggers support this yet. If this option is in effect, optimization will automatically attempt to eliminate the frame pointer, even when debugging.

'-mno-ocs-frame-position'

When emitting COFF debug information for automatic variables and parameters stored on the stack, the offset written out is from the frame pointer register 'r30.' This is the default behavior of the GNU C compiler. If this option is in effect, no automatic frame pointer elimination is done when debugging information is being written out via the -g switch.

'-msilicon-filter'

Run the silicon filter ("sifilter") before calling the assembler. This is no longer needed, since the GNU C compiler knows about the various features of the current Motorola 88100 chips.

'-mno-silicon-filter'

Do not run the silicon filter ("sifilter") before calling the assembler. This is the default behavior of the GNU C compiler.

'-mstatic-literal-synthesis'

Emit one instruction sequence to reference static data, using register 'r25' as a base register to the beginning of the data section. At present, this option also sets the '-mconst-uses-data' option. This option is experimental, and may change from release to release.

You must compile the file containing the main function with this option if you compile any module with it on. This is because the compiler loads register 'r25' in the prologue of the main function.
This option will not work if either your linker does not align the data section to a 64K boundary, or you have more than 64K of data. Referring to function addresses, except in a call context, does not work at present. No checking is done at present for these cases.

'-mno-static-literal-synthesis'

Do not perform static literal synthesis. This option is on by default.

'-mtrace-function'

Call the function '__enter' before the normal function prologue, and also call the function '__leave' after the normal function epilogue. These routines are passed the address of a dope vector in register 'r10', and the return address of the current function in register 'r13'. The dope vector is a table of values for use in the '__enter' and '__leave' functions, and consists of: 1) a count of the words in the dope vector; 2) a pointer to a null terminated string, giving the function name; 3) a word containing the length of the string; 4) a bitmask giving the fixed registers used (typically 'r0', 'r30', and 'r31'); and 5) a bitmask giving the registers that are expected to be preserved across calls (typically 'r14' through 'r31').

Because of the unusual calling sequence, these routines should be coded in assembler, or carefully written C code with assembler assists. The purpose of these routines is to provide some low level support to track down things like preserved registers getting clobbered, and the like.

'-mno-trace-function'

Do not call the special prologue and epilogue routines. This switch is the default behavior.

'-mtrap-large-shift'

Emit a 'tbnd' instruction before each shift by a non-constant amount, to trap if the shift count is less than zero or greater than 31. The 88000 does not produce a reasonable result in such cases, and the trap will halt the program at the point an unreasonable shift is done, rather than producing bogus results. The ANSI standard for C specifies that shifts outside of the range of 0 to number_bits – 1 is undefined. It is an error to specify both '-mtrap-large-shift' and '-mhandle-large-shift'.

'-mno-trap-large-shift'

Do not emit a 'tbnd' instruction before each shift by a non-constant amount, to trap if the shift count is less than zero or greater than 31.

'-mno-underscores'

Do not emit a leading underscore before all external names. You should not use this switch in general, unless you make sure every module has been compiled with it, including the standard library.

'-mtwo-underscores'

Emit two leading underscores before all external names instead of the normal single underscore. You should not use this switch in general, unless you make sure every module has been compiled with it, including the standard library.

'-muse-div-instruction'

Do not emit code to check both the divisor and dividend when doing normal integer division (as opposed to unsigned division) to see if either is negative, and fix things up so that the division is done with positive numbers. You would use this switch when you are confident that most or all signed divisions are done with positive numbers.

If this fixup is not done, the 88000 will trap to the kernel if either number is negative. The operating system will calculate the correct answer for all negative operands, except for the most negative number (–214783648) divided by negative 1, whose signed result cannot be represented in 32 bits.

'-mno-use-div-instruction'

Emit code to check both the divisor and dividend when doing normal integer division (as opposed to unsigned division) to see if either is negative, and fix things up so that the division is done with positive numbers (except for the case of the most negative number, which does not have a positive counterpart). This is the default behavior of the GNU C compiler.

The cost of doing this check is that up to two more registers are needed; some extra time is spent figuring out whether the operands are negative; and the GNU compiler does not optimize expressions containing division like it would normally.

'-mwarn-passed-structs'

Emit a warning message if a structure is passed to a function, or declared as a function argument. At least one other C compiler for the 88000 does not pass structures according to the 88000 Object Compatibility Standard, and this switch warns about the places where GNU CC will not interoperate with that compiler.

'-f*flag*'

Specify machine-independent flags. Most flags have both positive and negative forms; the negative form of '-ffoo' would be '-fno-foo'. In the table below, only one of the forms is listed---the one which is not the default. You can figure out the other form by either removing 'no-' or adding it.

'-fpcc-struct-return'

Use the same convention for returning struct and union values that is used by the usual C compiler on your system. This convention is less efficient for small structures, and on many machines it fails to be reentrant; but it has the advantage of allowing intercallability between GCC-compiled code and PCC-compiled code.

'-ffloat-store'

Do not store floating-point variables in registers. For most programs, the excess precision does only good, but a few programs rely on the precise definition of IEEE floating point. Use '-ffloat-store' for such programs.

'-fno-asm'

Do not recognize asm, inline, or typeof as a keyword. These words may then be used as identifiers. You can use __asm, __inline and__typeof instead.

'-fno-defer-pop'

Always pop the arguments to each function call as soon as that function returns. Normally the compiler (when optimizing) lets arguments accumulate on the stack for several function calls and pops them all at once.

'-fstrength-reduce'

Perform the optimizations of loop strength reduction and elimination of iteration variables.

'-fcombine-regs'

Allow the combine pass to combine an instruction that copies one register into another. This might or might not produce better code when used in addition to '-O'.

'-fforce-mem'

Force memory operands to be copied into registers before doing arithmetic on them. This may produce better code by making all memory references potential common subexpressions. When they are not common subexpressions, instruction combination should eliminate the separate register-load.

'-fforce-addr'

Force memory address constants to be copied into registers before doing arithmetic on them. This may produce better code just as '-fforce-mem' may.

'-fomit-frame-pointer'

Don't keep the frame pointer in a register for functions that don't need one. This avoids the instructions to save, set up and restore frame pointers; it also makes an extra register available in many functions. It also makes debugging impossible.

On the 88000, if you specify '-O' and do not specify '-g' or '-fno-omit-frame-pointer', this is enabled automatically.

'-finline-functions'

Integrate all simple functions into their callers. The compiler heuristically decides which functions are simple enough to be worth integrating in this way. If all calls to a given function are integrated, and the function is declared static, then the function is normally not output as assembler code in its own right.

'-fcaller-saves'

Enable values to be allocated in registers that will be clobbered by function calls, by emitting extra instructions to save and restore the registers around such calls. Such allocation is done only when it seems to result in better code than would otherwise be produced. This option is enabled by default on certain machines, usually those which have no call preserved registers to use instead.

'-fkeep-inline-functions'

Even if all calls to a given function are integrated, and the function is declared static, nevertheless output a separate run-time callable version of the function.

'-fwritable-strings'

Store string constants in the writable data segment and don't uniquize them. This is for compatibility with old programs which assume they can write into string constants. Writing into string constants is a very bad idea; "constants" should be constant.

'-fcond-mismatch'

Allow conditional expressions with mismatched types in the second and third arguments. The value of such an expression is void.

'-ffunction-cse'

Attempt to put function addresses in registers; an instruction that calls a constant function need not contain the function's address explicitly.

'-fvolatile'

Consider all memory references through pointers to be volatile.

'-fshared-data'

Requests that the data and non-const variables of this compilation be shared data rather than private data. The distinction makes sense only on certain operating systems, where shared data is shared between processes running the same program, while private data exists in one copy per process.

'-funsigned-char'

Let the type char be the unsigned, like unsigned char. Each kind of machine has a default for what char should be. It is either like unsigned char by default or like signed char by default.

The type char is always a distinct type from either signed char or unsigned char, even though its behavior is always just like one of those two.

Note that this is equivalent to '-fno-signed-char', which is the negative form of '-fsigned-char'.

'-fsigned-char'

Let the type char be signed, like signed char. Note that this is equivalent to '-fno-unsigned-char', which is the negative form of '-funsigned-char'.

'-funsigned-bit'

Treat bit fields that do not specify unsigned or signed as if unsigned were specified. This is the default for the 88000.

Note that this is equivalent to '-fno-signed-bit', which is the negative form of '-fsigned-bit'.

'-fsigned-bit'

Treat bit fields that do not specify unsigned or signed as if signed were specified. Note that this is equivalent to '-fno-unsigned-bit', which is the negative form of '-funsigned-bit'.

'-ffixed-*reg*'

Treat the register named *reg* as a fixed register; generated code should never refer to it (except perhaps as a stack pointer, frame pointer or in some other fixed role).

reg must be the name of a register. The register names accepted are machine-specific and are defined in the REGISTER_NAMES macro in the machine description macro file.

This flag does not have a negative form, because it specifies a three-way choice.

'-fcall-used-*reg*'

Treat the register named *reg* as an allocatable register that is clobbered by function calls. It may be allocated for temporaries or variables that do not live across a call. Functions compiled this way will not save and restore the register *reg*.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

This flag does not have a negative form, because it specifies a three-way choice.

'-fcall-saved-*reg*'

Treat the register named *reg* as an allocatable register saved by functions. It may be allocated even for temporaries or variables that live across a call. Functions compiled this way will save and restore the register *reg* if they use it.

Use of this flag for a register that has a fixed pervasive role in the machine's execution model, such as the stack pointer or frame pointer, will produce disastrous results.

A different sort of disaster will result from the use of this flag for a register in which function values may be returned.

This flag does not have a negative form, because it specifies a three-way choice.

'-pedantic' Issue all the warnings demanded by strict ANSI standard C; reject all programs that use forbidden extensions. Valid ANSI standard C programs should compile properly with or without this option (though a rare few will require '-ansi'). However, without this option, certain GNU extensions and traditional C features are supported as well. With this option, they are rejected.

These options control the C preprocessor, which is run on each C source file before actual compilation. If you use the '-E' option, nothing is done except C preprocessing. Some of these options make sense only together with '-E' because they request preprocessor output that is not suitable for actual compilation.

'-C' Tell the preprocessor not to discard comments. Used with the '-E' option.

'-I*dir*' Search directory *dir* for include files.

'-I-' Any directories specified with '-I' options before the'-I-' option are searched only for the case of '#include "*file*"'; they are not searched for '#include <*file*>'.

If additional directories are specified with '-I' options after the '-I-', these directories are searched for all '#include' directives. (Ordinarily *all* '-I' directories are used this way.)

In addition, the '-I-' option inhibits the use of the current directory as the first search directory for '#include "*file*"'. Therefore, the current directory is searched only if it is requested explicitly with '-I.' Specifying both '-I-' and '-I.' allows you to control precisely which directories are searched before the current one and which are searched after.

'-nostdinc'
: Do not search the standard system directories for header files. Only the directories you have specified with '-I' options (and the current directory, if appropriate) are searched.

  Between '-nostdinc' and '-I-', you can eliminate all directories from the search path except those you specify.

'-M'
: Tell the preprocessor to output a rule suitable for make describing the dependencies of each source file. For each source file, the preprocessor outputs one make-rule whose target is the object file name for that source file and whose dependencies are all the files '#include'd in it. This rule may be a single line or may be continued with '\'-newline if it is long.

  '-M' implies '-E'.

'-MM'
: Like '-M' but the output mentions only the user-header files included with '#include "*file*"'. System header files included with '#include <*file*>' are omitted.

  '-MM' implies '-E'.

'-H'
: Tell the preprocessor to output the names of include files to the standard error file, in addition to the normal processing.

'-D*macro*'
: Define macro *macro* with the empty string as its definition.

'-D*macro*=*defn*'
: Define macro *macro* as *defn*.

'-U*macro*'
: Undefine macro *macro*.

'-trigraphs'
: Support ANSI C trigraphs. The '-ansi' option also has this effect.

End of Chapter

Chapter 2
Known Causes of Trouble with GNU CC

Here are some of the things that have caused trouble for people using GNU CC.

- Users often think it is a bug when GNU CC reports an error for code like this:

```
int foo (short);

int foo (x)
short x;
{...}
```

  The error message is correct: this code really is erroneous, because the old-style non-prototype definition passes subword integers in their promoted types. In other words, the argument is really an `int`, not a `short`. The correct prototype is this:

```
int foo (int);
```

- Users often think it is a bug when GNU CC reports an error for code like this:

```
int foo (struct mumble *);

struct mumble { ... };

int foo (struct mumble *x)
{ ... }
```

  This code really is erroneous, because the scope of `struct mumble` the prototype is limited to the argument list containing it. It does not refer to the `struct mumble` defined with file scope immediately below—they are two unrelated types with similar names in different scopes.

  But, in the definition of `foo`, the file-scope type is used because that is available to be inherited. Thus, the definition and the prototype do not match, and you get an error.

  This behavior may seem unreasonable, but it's what the ANSI standard specifies. It is easy enough for you to make your code work by moving the definition of `struct mumble` above the prototype.

End of Chapter

Chapter 3
Incompatibilities of GNU CC

There are several noteworthy incompatibilities between GNU C and most existing (non-ANSI) versions of C.

Ultimately our intention is that the '-traditional' option will eliminate most of these incompatibilities by telling GNU C to behave like the other C compilers.

- GNU CC normally makes string constants read-only. If several identical-looking string constants are used, GNU CC stores only one copy of the string.

  One consequence is that you cannot call `mktemp` with a string constant argument. The function `mktemp` always alters the string its argument points to.

  Another consequence is that `sscanf` does not work on some systems when passed a string constant as its format control string. This is because `sscanf` incorrectly tries to write into the string constant; `fscanf` and `scanf` do the same.

  The best solution to these problems is to change the program to use `char`-array variables with initialization strings for these purposes instead of string constants. But if this is not possible, you can use the '-fwritable-strings' flag, which directs GNU CC to handle string constants the same way most C compilers do.

- GNU CC does not substitute macro arguments when they appear inside of string constants. For example, the following macro in GNU CC

  ```
  #define foo(a) "a"
  ```

  will produce output '"a"' regardless of what the argument `a` is.

  The '-traditional' option directs GNU CC to handle such cases (among others) in the old-fashioned (non-ANSI) fashion.

- When you use `setjmp` and `longjmp`, the only automatic variables guaranteed to remain valid are those declared `volatile`. This is a consequence of automatic register allocation. Consider the following function:

```
jmp_buf j;

foo () {
     int a, b;

     a = fun1 ();
     if (setjmp (j))
        return a;

     a = fun2 ();      /* longjmp (j) may be occur in fun3. */
     return a + fun3 ();
}
```

Here a may or may not be restored to its first value when the `longjmp` occurs. If a is allocated in a register, then its first value is restored; otherwise, it keeps the last value stored in it.

If you use the '`-W`' option with the '`-O`' option, you will get a warning when GNU CC thinks such a problem might be possible.

The '`-traditional`' option directs GNU C to put variables in the stack by default, rather than in registers, in functions that call `setjmp`. This results in the behavior found in traditional C compilers.

- Declarations of external variables and functions within a block apply only to the block containing the declaration. In other words, they have the same scope as any other declaration in the same place.

  In some other C compilers, an `extern` declaration affects all the rest of the file even if it happens within a block.

  The '`-traditional`' option directs GNU C to treat all `extern` declarations as global, like traditional compilers.

- In traditional C, you can combine `long`, etc., with a typedef name, as shown here:

  ```
  typedef int foo;
  typedef long foo bar;
  ```

  In ANSI C, this is not allowed: `long` and other type modifiers require an explicit `int`. Because this criterion is expressed by Bison grammar rules rather than C code, the '`-traditional`' flag cannot alter it.

- PCC allows typedef names to be used as function parameters. The difficulty described immediately above applies here too.

- PCC allows whitespace in the middle of compound assignment operators such as '+='. GNU CC, following the ANSI standard, does not allow this. The difficulty described immediately above applies here too.

- GNU CC will flag unterminated character constants inside of preprocessor conditionals that fail. Some programs have English comments enclosed in conditionals that are guaranteed to fail; if these comments contain apostrophes, GNU CC will probably report an error. For example, this code would produce an error:

  ```
  #if 0
  You can't expect this to work.
  #endif
  ```

  The best solution to such a problem is to put the text into an actual C comment delimited by '`/*...*/`'. However, '`-traditional`' suppresses these error messages.

- When compiling functions that return `float`, PCC converts it to a double. GNU CC actually returns a `float`. If you are concerned with PCC compatibility, you should declare your functions to return `double`; you might as well say what you mean.

- When compiling functions that return structures or unions, GNU CC output code normally uses a method different from that used on most versions of UNIX. As a result, code compiled with GNU CC cannot call a structure-returning function compiled with PCC, and vice versa.

  The method used by GNU CC is as follows: a structure or union which is 1, 2, 4 or 8 bytes long is returned like a scalar. A structure or union with any other size is stored into an address supplied by the caller in a special, fixed register.

  PCC usually handles all sizes of structures and unions by returning the address of a block of static storage containing the value. This method is not used in GNU CC because it is slower and non-reentrant.

  You can tell GNU CC to use the PCC convention with the option '`-fpcc-struct-return`'.

End of Chapter

Chapter 4
GNU Extensions to the C Language

GNU C provides several language features not found in ANSI standard C. (The '-pedantic' option directs GNU CC to print a warning message if any of these features is used.) To test for the availability of these features in conditional compilation, check for a predefined macro __GNUC__, which is always defined under GNU CC.

Statements and Declarations inside of Expressions

A compound statement in parentheses may appear inside an expression in GNU C. This allows you to declare variables within an expression. For example:

```
({ int y = foo (); int z;
   if (y > 0) z = y;
   else z = - y;
   z; })
```

is a valid (though slightly more complex than necessary) expression for the absolute value of `foo ()`.

This feature is especially useful in making macro definitions "safe" (so that they evaluate each operand exactly once). For example, the "maximum" function is commonly defined as a macro in standard C as follows:

```
#define max(a,b) ((a) > (b) ? (a) : (b))
```

But this definition computes either a or b twice, with bad results if the operand has side effects. In GNU C, if you know the type of the operands (here, let's assume `int`), you can define the macro safely as follows:

```
#define maxint(a,b) \
  ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
```

Embedded statements are not allowed in constant expressions, such as the value of an enumeration constant, the width of a bit field, or the initial value of a static variable.

If you don't know the type of the operand, you can still do this, but you must use `typeof` (see the `typeof` section later in this chapter) or type naming (see the section below).

Naming an Expression's Type

You can give a name to the type of an expression using a `typedef` declaration with an initializer. Here is how to define *name* as a type name for the type of *exp*:

```
typedef name = exp;
```

This is useful in conjunction with the statements-within-expressions feature. Here is how the two together can be used to define a safe "maximum" macro that operates on any arithmetic type:

```
#define max(a,b) \
   ({typedef _ta = (a), _tb = (b); \
   _ta _a = (a); _tb _b = (b); \
   _a > _b ? _a : _b; })
```

The reason for using names that start with underscores for the local variables is to avoid conflicts with `typedef` variable names that occur within the expressions that are substituted for `a` and `b`. Eventually we hope to design a new form of declaration syntax that allows you to declare variables whose scopes start only after their initializers; this will be a more reliable way to prevent such conflicts.

Referring to a Type with typeof

Another way to refer to the type of an expression is with `typeof`. The syntax of using of this keyword looks like `sizeof`, but the construct acts semantically like a type name defined with `typedef`.

There are two ways of writing the argument to `typeof`: with an expression or with a type. Here is an example with an expression:

```
typeof (x[0](1))
```

This assumes that `x` is an array of functions; the type described is that of the values of the functions.

Here is an example with a typename as the argument:

```
typeof (int *)
```

Here the type described is that of pointers to `int`.

If you are writing a header file that must work when included in ANSI C programs, write `__typeof` instead of `typeof`. See the section "Alternate Keywords" at the end of this chapter.

A `typeof`-construct can be used anywhere a typedef name could be used. For example, you can use it in a declaration, in a cast, or inside of `sizeof` or `typeof`.

- This declares `y` with the type of what `x` points to.

  ```
  typeof (*x) y;
  ```

- This declares y as an array of such values.

```
typeof (*x) y[4];
```

- This declares y as an array of pointers to characters:

```
typeof (typeof (char *)[4]) y;
```

  It is equivalent to the following traditional C declaration:

```
char *y[4];
```

To see the meaning of the declaration using `typeof`, and why it might be a useful way to write, let's rewrite it with these macros:

```
#define pointer(T) typeof(T *)
#define array(T, N) typeof(T [N])
```

Now the declaration can be rewritten this way:

```
array (pointer (char), 4) y;
```

Thus, '`array (pointer (char), 4)`' is the type of arrays of 4 pointers to `char`.

Generalized Lvalues

Compound expressions, conditional expressions and casts are allowed as lvalues provided their operands are lvalues. This means that you can take their addresses or store values into them.

For example, a compound expression can be assigned, provided the last expression in the sequence is an lvalue. These two expressions are equivalent:

```
(a, b) += 5
a, (b += 5)
```

Similarly, the address of the compound expression can be taken. These two expressions are equivalent:

```
&(a, b)
a, &b
```

A conditional expression is a valid lvalue if its type is not `void` and the true and false branches are both valid lvalues. For example, these two expressions are equivalent:

```
(a ? b : c) = 5
(a ? b = 5 : (c = 5))
```

A cast is a valid lvalue if its operand is valid. Taking the address of the cast is the same as taking the address without a cast, except for the type of the result. For example, these two expressions are equivalent (but the second may be valid when the type of 'a' does not permit a cast to 'int *').

```
&(int *)a
(int **)&a
```

A simple assignment whose left-hand side is a cast works by converting the right-hand side first to the specified type, then to the type of the inner left-hand side expression. After this is stored, the value is converted back to the specified type to become the value of the assignment. Thus, if 'a' has type 'char *', the following two expressions are equivalent:

```
(int)a = 5
(int)(a = (char *)5)
```

An assignment-with-arithmetic operation such as '+=' applied to a cast performs the arithmetic using the type resulting from the cast, and then continues as in the previous case. Therefore, these two expressions are equivalent:

```
(int)a += 5
(int)(a = (char *) ((int)a + 5))
```

Conditional Expressions with Omitted Middle–Operands

The middle operand in a conditional expression may be omitted. Then if the first operand is nonzero, its value is the value of the conditional expression.

Therefore, the expression

```
x ?: y
```

has the value of x if that is nonzero; otherwise, it has the value of y.

This example is perfectly equivalent to

```
x ? x : y
```

In this simple case, the ability to omit the middle operand is not especially useful. When it becomes useful is when the first operand does, or may (if it is a macro argument), contain a side effect. Then repeating the operand in the middle would perform the side effect twice. Omitting the middle operand uses the value already computed without the undesirable effects of recomputing it.

Arrays of Length Zero

Zero-length arrays are allowed in GNU C. They are very useful as the last element of a structure which is really a header for a variable-length object:

```
struct line {
  int length;
  char contents[0];
};

{
  struct line *thisline = (struct line *) malloc (sizeof (struct line) +
    this_length);
  thisline->length = this_length;
}
```

In standard C, you would have to give `contents` a length of 1, which means either you waste space or complicate the argument to `malloc`.

Arrays of Variable Length

Variable-length automatic arrays are allowed in GNU C. These arrays are declared like any other automatic arrays, but with a length that is not a constant expression. The storage is allocated at that time and deallocated when the brace-level is exited. For example:

```
FILE *concat_fopen (char *s1, char *s2, char *mode) {
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

You can also use variable-length arrays as arguments to functions:

```
struct entrytester (int len, char data[len]) {
  ...
}
```

The length of an array is computed on entry to the brace-level where the array is declared and is remembered for the scope of the array in case you access it with `sizeof`.

Jumping or breaking out of the scope of the array name will also deallocate the storage. Jumping into the scope is not allowed; you will get an error message for it.

You can use the function `alloca` to get an effect much like variable-length arrays. The function `alloca` is available in many other C implementations (but not in all). On the other hand, variable-length arrays are more elegant.

There are other differences between these two methods. Space allocated with `alloca` exists until the containing *function* returns. The space for a variable-length array is deallocated as soon as the array name's scope ends. (If you use both variable-length arrays and `alloca` in the same function, deallocation of a variable-length array will also deallocate anything more recently allocated with alloca.)

Non-Lvalue Arrays May Have Subscripts

Subscripting is allowed on arrays that are not lvalues, even though the unary '&' operator is not. For example, this is valid in GNU C though not valid in other C dialects:

```
struct foo {int a[4];};

struct foo f();

bar (int index){
   return f().a[index];
}
```

Arithmetic on void–Pointers and Function Pointers

In GNU C, addition and subtraction operations are supported on pointers to `void` and on pointers to functions. This is done by treating the size of a `void` or of a function as 1.

A consequence of this is that `sizeof` is also allowed on `void` and on function types, and returns 1.

Non–Constant Initializers

The elements of an aggregate initializer are not required to be constant expressions in GNU C. Here is an example of an initializer with run–time varying elements:

```
foo (float f, float g){
   float beat_freqs[2] = { f-g, f+g };
   ...
}
```

Constructor Expressions

GNU C supports constructor expressions. A constructor looks like a cast containing an initializer. Its value is an object of the type specified in the cast, containing the elements specified in the initializer. The type must be a structure, union or array type.

Assume that `struct foo` and `structure` are declared as shown:

```
struct foo {int a; char b[2];} structure;
```

Here is an example of constructing a 'struct foo' with a constructor:

```
structure = ((struct foo) {x + y, 'a', 0});
```

This is equivalent to writing the following:

```
{
   struct foo temp = {x + y, 'a', 0};
   structure = temp;
}
```

You can also construct an array. If all the elements of the constructor are (made up of) simple constant expressions, suitable for use in initializers, then the constructor is an lvalue and can be coerced to a pointer to its first element, as shown here:

```
char **foo = (char *[]) { "x", "y", "z" };
```

Array constructors whose elements are not simple constants are not very useful, because the constructor is not an lvalue. There are only two valid ways to use it: to subscript it, or initialize an array variable with it. The former is probably slower than a switch statement, while the latter does the same thing an ordinary C initializer would do.

```
output = ((int[]) { 2, x, 28 }) [input];
```

Declaring Attributes of Functions

In GNU C, you declare certain things about functions called in your program which help the compiler optimize function calls.

A few functions, such as abort and exit, cannot return. These functions should be declared volatile. For example,

```
extern volatile void abort ();
```

tells the compiler that it can assume that abort will not return. This makes slightly better code, but more importantly it helps avoid spurious warnings of uninitialized variables.

Many functions do not examine any values except their arguments, and have no effects except the return value. Such a function can be subject to common subexpression elimination and loop optimization just as an arithmetic operator would be. These functions should be declared const. For example,

```
extern const void square ();
```

says that the hypothetical function square is safe to call fewer times than the program says.

Note that a function that has pointer arguments and examines the data pointed to must *not* be declared `const`. Likewise, a function that calls a non-`const` function must not be `const`.

Some people object to this feature, claiming that ANSI C's `#pragma` should be used instead. There are two reasons I did not do this.

1. It is impossible to generate `#pragma` commands from a macro.
2. The `#pragma` command is just as likely as these keywords to mean something else in another compiler.

These two reasons apply to *any* application whatever: as far as I can see, `#pragma` is never useful.

Dollar Signs in Identifier Names

In GNU C, you may use dollar signs in identifier names. This is because many traditional C implementations allow such identifiers.

Inquiring about the Alignment of a Type or Variable

The keyword `__alignof` allows you to inquire about how an object is aligned, or the minimum alignment usually required by a type. Its syntax is just like `sizeof`.

For example, if the target machine requires a `double` value to be aligned on an 8-byte boundary, then `__alignof (double)` is 8. This is true on many RISC machines. On more traditional machine designs, `__alignof (double)` is 4 or even 2.

Some machines never actually require alignment; they allow reference to any data type even at odd addresses. For these machines, `__alignof` reports the *recommended* alignment of a type.

When the operand of `__alignof` is an lvalue rather than a type, the value is the largest alignment that the lvalue is known to have. It may have this alignment as a result of its data type, or because it is part of a structure and inherits alignment from that structure. For example, after this declaration:

```
struct foo { int x; char y; } foo1;
```

the value of `__alignof (foo1.y)` is probably 2 or 4, the same as `__alignof (int)`, even though the data type of `foo1.y` does not itself demand any alignment.

An Inline Function is As Fast As a Macro

By declaring a function `inline`, you can direct GNU CC to integrate that function's code into the code for its callers. This makes execution faster by eliminating the function-call overhead; in addition, if any of the actual argument values are constant, their known values may permit simplifications at compile time so that not all of the inline function's code needs to be included.

To declare a function inline, use the `inline` keyword in its declaration, like this:

```
inline int inc (int *a) {
   (*a)++;
}
```

(If you are writing a header file to be included in ANSI C programs, write `__inline` instead of `inline`. See the section "Alternate Keywords" at the end of this chapter.)

You can also make all "simple enough" functions inline with the option '`-finline-functions`'. Note that certain usages in a function definition can make it unsuitable for inline substitution.

When a function is both inline and `static`, if all calls to the function are integrated into the caller, then the function's own assembler code is never referenced. In this case, GNU CC does not actually output assembler code for the function, unless you specify the option '`-fkeep-inline-functions`'. Some calls cannot be integrated for various reasons (in particular, calls that precede the function's definition cannot be integrated, and neither can recursive calls within the definition). If there is a nonintegrated call, then the function is compiled to assembler code as usual.

When an inline function is not `static`, then the compiler must assume that there may be calls from other source files; since a global symbol can be defined only once in any program, the function must not be defined in the other source files, so the calls therein cannot be integrated. Therefore, a non-`static` inline function is always compiled on its own in the usual fashion.

Assembler Instructions with C Expression Operands

In an assembler instruction using `asm`, you can now specify the operands of the instruction using C expressions. This means no more guessing which registers or memory locations will contain the data you want to use.

You must specify an assembler instruction template much like what appears in a machine description, plus an operand constraint string for each operand.

For example, here is how to use the 68881's `fsinx` instruction:

```
asm ("fsinx %1,%0" : "=f" (result) : "f" (angle));
```

Here `angle` is the C expression for the input operand while `result` is that of the output operand. Each has '`"f"`' as its operand constraint, saying that a floating-point register is required. The '=' in '`=f`' indicates that the operand is an output; all output operands' constraints must use '='. The constraints use the same language used in the machine description.

Each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand, and another separates the last output operand from the first input, if any. Commas separate output operands and separate inputs. The total number of operands is limited to the maximum number of operands in any instruction pattern in the machine description.

If there are no output operands, and there are input operands, then there must be two consecutive colons surrounding the place where the output operands would go.

Output operand expressions must be lvalues; the compiler can check this. The input operands need not be lvalues. The compiler cannot check whether the operands have data types that are reasonable for the instruction being executed. It does not parse the assembler instruction template and does not know what it means, or whether it is valid assembler input. The extended `asm` feature is most often used for machine instructions that the compiler itself does not know exist.

The output operands must be write-only; GNU CC will assume that the values in these operands before the instruction are dead and need not be generated. For an operand that is read-write, or in which not all bits are written and the other bits contain useful information, you must logically split its function into two separate operands, one input operand and one write-only output operand. The connection between them is expressed by constraints which say they need to be in the same location when the instruction executes. You can use the same C expression for both operands, or different expressions. For example, here we write the (fictitious) '`combine`' instruction with `bar` as its read-only source operand and `foo` as its read-write destination:

```
asm ("combine %2,%0" : "=r" (foo) : "0" (foo), "g" (bar));
```

The constraint '`"0"`' for operand 1 says that it must occupy the same location as operand 0. A digit in constraint is allowed only in an input operand, and it must refer to an output operand.

Only a digit in the constraint can guarantee that one operand will be in the same place as another. The mere fact that `foo` is the value of both operands is not enough to guarantee that they will be in the same place in the generated assembler code. The following would not work:

```
asm ("combine %2,%0" : "=r" (foo) : "r" (foo), "g" (bar));
```

Various optimizations or reloading could cause operands 0 and 1 to be in different registers; GNU CC knows no reason not to do so. For example, the compiler might find a copy of the value of `foo` in one register and use it for operand 1, but generate the output operand 0 in a different register (copying it afterward to `foo`'s own address). Of course, since the register for operand 1 is not even mentioned in the assembler code, the result will not work, but GNU CC can't tell that.

Unless an output operand has the '&' constraint modifier, GNU CC may allocate it in the same register as an unrelated input operand, on the assumption that the inputs are consumed before the outputs are produced. This assumption may be false if the assembler code actually consists of more than one instruction. In such a case, use '&' for each output operand that may not overlap an input.

Some instructions clobber specific hard registers. To describe this, write a third colon after the input operands, followed by the names of the clobbered hard registers (given as strings). Here is a realistic example for the vax:

```
asm volatile ("movc3 %0,%1,%2"
              : /* no outputs */
              : "g" (from), "g" (to), "g" (count)
              : "r0", "r1", "r2", "r3", "r4", "r5");
```

You can put multiple assembler instructions together in a single asm template, separated either with newlines (written as '\n') or with semicolons if the assembler allows such semicolons. The GNU assembler allows semicolons and all UNIX assemblers seem to do so. The input operands are guaranteed not to use any of the clobbered registers, and neither will the output operands' addresses, so you can read and write the clobbered registers as many times as you like. Here is an example of multiple instructions in a template; it assumes that the subroutine _foo accepts arguments in registers 9 and 10:

```
asm ("movl %0,r9;
      movl %1,r10;
      call _foo"
      : /* no outputs */
      : "g" (from), "g" (to)
      : "r9", "r10");
```

If you want to test the condition code produced by an assembler instruction, you must include a branch and a label in the asm construct, as follows:

```
asm ("clr %0;
      frob %1;
      beq 0f;
      mov #1,%0;
      0:"
      : "g" (result)
      : "g" (input));
```

This assumes your assembler supports local labels, as the GNU assembler and most UNIX assemblers do.

Usually the most convenient way to use these asm instructions is to encapsulate them in macros that look like functions. For example,

```
#define sin(x) \
  ({ double __value, __arg = (x); \
  asm ("fsinx %1,%0": "=f" (__value): "f" (__arg)); \
  __value; })
```

Here the variable __arg is used to make sure that the instruction operates on a proper double value, and to accept only those arguments x which can convert automatically to a double.

Another way to make sure the instruction operates on the correct data type is to use a cast in the asm. This is different from using a variable __arg in that it converts more different types. For example, if the desired type were int, casting the argument to int would accept a pointer with no complaint, while assigning the argument to an int variable named __arg would warn about using a pointer unless the caller explicitly casts it.

If an asm has output operands, GNU CC assumes for optimization purposes that the instruction has no side effects except to change the output operands. This does not mean that instructions with a side effect cannot be used, but you must be careful, because the compiler may eliminate them if the output operands aren't used, or move them out of loops, or replace two with one if they constitute a common subexpression. Also, if your instruction does have a side effect on a variable that otherwise appears not to change, the old value of the variable may be reused later if it happens to be found in a register.

You can prevent an asm instruction from being deleted, moved or combined by writing the keyword volatile after the asm. For example:

```
#define set_priority(x) \
  asm volatile ("set_priority %0": /* no outputs */ : "g" (x))
```

(However, an instruction without output operands will not be deleted or moved, regardless, unless it is unreachable.)

It is a natural idea to look for a way to give access to the condition code left by the assembler instruction. However, when we attempted to implement this, we found no way to make it work reliably. The problem is that output operands might need reloading, which would result in additional following "store" instructions. On most machines, these instructions would alter the condition code before there was time to test it. This problem doesn't arise for ordinary "test" and "compare" instructions because they don't have any output operands.

If you are writing a header file that should be includable in ANSI C programs, write __asm instead of asm. See the section "Alternate Keywords" at the end of this chapter.

Controlling Names Used in Assembler Code

You can specify the name to be used in the assembler code for a C function or variable by writing the asm (or __asm) keyword after the declarator as follows:

```
int foo asm ("myfoo") = 2;
```

This specifies that the name to be used for the variable foo in the assembler code should be 'myfoo' rather than the usual '_foo'.

On systems where an underscore is normally prepended to the name of a C function or variable, this feature allows you to define names for the linker that do not start with an underscore.

You cannot use asm in this way in a function *definition*; but you can get the same effect by writing a declaration for the function before its definition and putting asm there, like this:

```
extern func () asm ("FUNC");

func (x, y)
int x, y;
...
```

It is up to you to make sure that the assembler names you choose do not conflict with any other assembler symbols. Also, you must not use a register name; that would produce completely invalid assembler code. GNU CC does not as yet have the ability to store static variables in registers. Perhaps that will be added.

Global Variables in Registers

A few programs, such as programming language interpreters, may have a couple of global variables that are accessed so often that it is worth while to reserve registers throughout the program just for them.

You can define a global register variable in GNU C like this:

```
register int *foo asm ("a5");
```

Here a5 is the name of the register which should be used. Choose a register which is normally saved and restored by function calls on your machine, so that library routines will not clobber it.

Naturally the register name is cpu-dependent, so you would need to conditionalize your program according to cpu type. The register a5 would be a good choice on a 68000 for a variable of pointer type. On machines with register windows, be sure to choose a "global" register that is not affected by the function call mechanism.

In addition, operating systems on one type of cpu may differ in how they name the registers; then you would need additional conditionals. For example, some 68000 operating systems call this register %a5.

Eventually there may be a way of asking the compiler to choose a register automatically, but first we need to figure out how it should choose and how to enable you to guide the choice. No solution is evident.

Defining a global register variable in a certain register reserves that register entirely for this use, at least within the current compilation. The register will not be allocated for any other purpose in the functions in the current compilation. The register will not be saved and restored by these functions. Stores into this register are never deleted even if they would appear to be dead, but references may be deleted or moved or simplified.

It is not safe to access the global register variables from signal handlers, or from more than one thread of control, because the system library routines may temporarily use the register for other things (unless you recompile them specially for the task at hand).

It is not safe for one function that uses a global register variable to call another such function foo by way of a third function lose that was compiled without knowledge of this variable (i.e., in a different source file in which the variable wasn't declared). This is because lose might save the register and put some other value there. For example, you can't expect a global register variable to be available in the comparison-function that you pass to qsort, since qsort might have put something else in that register. (If you are prepared to recompile qsort with the same global register variable, you can solve this problem.)

If you want to recompile qsort or other source files which do not actually use your global register variable, so that they will not use that register for any other purpose, then it suffices to specify the compiler option '-ffixed*reg*'. You need not actually add a global register declaration to their source code.

A function that can alter the value of a global register variable cannot safely be called from a function compiled without this variable, because it could clobber the value the caller expects to find there on return. Therefore, the function which is the entry point into the part of the program that uses the global register variable must explicitly save and restore the value which belongs to its caller.

On most machines, longjmp will restore to each global register variable the value it had at the time of the setjmp. On some machines, however, longjmp will not change the value of global register variables. To be portable, the function that called setjmp should make other arrangements to save the values of the global register variables, and to restore them if a longjmp. This way, the the same thing will happen regardless of what longjmp does.

All global register variable declarations must precede all function definitions. If such a declaration could appear after function definitions, the declaration would be too late to prevent the register from being used for other purposes in the preceding functions.

Global register variables may not have initial values, because an executable file has no means to supply initial contents for a register.

Alternate Keywords

The option '-traditional' disables certain keywords; '-ansi' disables certain others. This causes trouble when you want to use GNU C extensions, or ANSI C features, in a general-purpose header file that should be usable by all programs, including ANSI C programs and traditional ones. The keywords asm, typeof, and inline cannot be used since they won't work in a program compiled with '-ansi', while the keywords const, volatile, signed, typeof and inline won't work in a program compiled with '-traditional'.

The way to solve these problems is to put '__' in front of each problematical keyword. For example, use __asm instead of asm, __const instead of const, and __inline instead of inline.

Other C compilers won't accept these alternative keywords; if you want to compile with another compiler, you can define the alternate keywords as macros to replace them with the customary keywords. It looks like this:

```
#ifndef __GNUC__
#define __asm asm
#endif
```

End of Chapter

Appendix A
GNU General Public License

Version 1, February 1989
Copyright c 1989 Free Software Foundation, Inc.
675 Mass Ave, Cambridge, MA 02139, USA

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

Preamble

The license agreements of most software companies try to keep users at the mercy of those companies. By contrast, our General Public License is intended to guarantee your freedom to share and change free software---to make sure the software is free for all its users. The General Public License applies to the Free Software Foundation's software and to any other program whose authors commit to using it. You can use it for your programs, too.

When we speak of free software, we are referring to freedom, not price. Specifically, the General Public License is designed to make sure that you have the freedom to give away or sell copies of free software, that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of a such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must tell them their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

The precise terms and conditions for copying, distribution and modification follow.

Terms and Conditions

1. This License Agreement applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program," below, refers to any such program or work, and a "work based on the Program" means either the Program or any work containing the Program or a portion of it, either verbatim or with modifications. Each licensee is addressed as "you."

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this General Public License and to the absence of any warranty; and give any other recipients of the Program a copy of this General Public License along with the Program. You may charge a fee for the physical act of transferring a copy.

3. You may modify your copy or copies of the Program or any portion of it, and copy and distribute such modifications under the terms of Paragraph 2 above, provided that you also do the following:

   a. cause the modified files to carry prominent notices stating that you changed the files and the date of any change; and

   b. cause the whole of any work that you distribute or publish, that in whole or in part contains the Program or any part thereof, either with or without modifications, to be licensed at no charge to all third parties under the terms of this General Public License (except that you may choose to grant warranty protection to some or all third parties, at your option).

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the simplest and most usual way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this General Public License.

   d. You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   Mere aggregation of another independent work with the Program (or its derivative) on a volume of a storage or distribution medium does not bring the other work under the scope of these terms.

4. You may copy and distribute the Program (or a portion or derivative of it, under Paragraph 3) in object code or executable form under the terms of Paragraphs 2 and 3 above provided that you also do one of the following:

   a. accompany it with the complete corresponding machine–readable source code, which must be distributed under the terms of Paragraphs 2 and 3 above; or,

   b. accompany it with a written offer, valid for at least three years, to give any third party free (except for a nominal charge for the cost of distribution) a complete machine–readable copy of the corresponding source code, to be distributed under the terms of Paragraphs 2 and 3 above; or,

   c. accompany it with the information you received as to where the corresponding source code may be obtained. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form alone.)

Source code for a work means the preferred form of the work for making modifications to it. For an executable file, complete source code means all the source code for all modules it contains; but, as a special exception, it need not include source code for modules which are standard libraries that accompany the operating system on which the executable file runs, or for standard header files or definitions files that accompany that operating system.

5. You may not copy, modify, sublicense, distribute or transfer the Program except as expressly provided under this General Public License. Any attempt otherwise to copy, modify, sublicense, distribute or transfer the Program is void, and will automatically terminate your rights to use the Program under this License. However, parties who have received copies, or rights to use copies, from you under this General Public License will not have their licenses terminated so long as such parties remain in full compliance.

6. By copying, distributing or modifying the Program (or any work based on the Program) you indicate your acceptance of this license to do so, and all its terms and conditions.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein.

8. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

   Each version is given a distinguishing version number. If the Program specifies a version number of the license which applies to it and "any later version," you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the license, you may choose any version ever published by the Free Software Foundation.

9. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

No Warranty

BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING, THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT, UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING, WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING, BUT NOT LIMITED TO, LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

How to Apply These Terms to Your New Programs

If you develop a new program and you want it to be of the greatest possible use to humanity, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; each file should have at least the "copyright" line and a pointer to where the full notice is found.

one line to give the program's name and a brief idea of what it does.
Copyright (C) 19*yy*
name of author

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 1, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19*yy* *name of author*
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type 'show w'. This is free software, and you are welcome to redistribute it under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than 'show w' and 'show c'; they could even be mouse clicks or menu items---whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here's a sample; alter the names:

Yoyodyne, Inc., hereby disclaims all copyright interest in the program 'Gnomovision' (a program to direct compilers to make passes at assemblers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice

That's all there is to it!

End of Appendix

Index

Symbols

$, 4–8

A

-a, 1–6

Adding directories to search lists, 1–6

Alignment of a type or variable, 4–8

__alignof, 4–8

alloca, variable-length arrays and, 4–5

Allocatable registers, 1–13, 1–14

Alternate
  keywords, 1–2, 4–14
  predefined macros, 1–2

-ansi, 1–2, 4–14

ANSI standard C, 1–2, 1–14

Arguments
  macro, 3–1
  popping after each function, 1–11

Array constructors, 4–7

Arrays
  of length zero, 4–5
  of variable length, 4–5
    vs. alloca, 4–5
  subscripting, 4–6

asm, 4–9, 4–12
  not recognizing as a keyword, 1–11
  volatile and, 4–12

Assembler code
  instructions with C expression operands, 4–9
  output, 4–9
  specifying names, 4–12

Assembler version, 1–1

Assembling source files, 1–1

Automatic variables, 1–4, 3–1

B

Bits
  signed, 1–13
  unsigned, 1–13

Block moves, 1–7
  not attempting, 1–7

-B*prefix*, 1–2

C

-C, 1–14

-c, 1–1

C
  Gnu extensions to, 4–1
  incompatibilities between GNU and others, 3–1

.c suffix, 1–1

Calling functions, special prologue and epilogue routines for, 1–9

Canonical frame pointer, 1–8

Casts, 1–6
  as lvalues, 4–3

char
  signed, 1–13
  unsigned, 1–13

Character constants, unterminated, 3–3

Code size, reducing, 1–3

COFF debugging information, 1–8, 1–9

Combining instructions that copy registers, 1–11

Command options, 1–1

Commands, printing compiler driver, 1–1

Comments, 1–6
  not discarding during preprocessing, 1–14

Compiler
  options, 1–1
  version of, 1–1

Compiler driver
  printing commands, 1–1
  version of, 1–1

Compiling source files, 1–1

Compound expressions as lvalues, 4–3

Conditional expressions
  allowing mismatched types in, 1–12
  as lvalues, 4–3
  with omitted middle-operands, 4–4

const, 4–7

Constant data
  putting in data section, 1–7
  putting in text section, 1–7

Constants
  string, 1–6, 3–1
    storing in the writable data segment, 1–12
  unterminated character, 3–3

Constructor expressions, 4–6

Copying
  memory address constants into registers, 1–12
  memory operands into registers, 1–11

D

Data, shared, 1–13

Data section vs. text section, 1–7

Debugging information
  COFF, 1–8, 1–9
  complying with OCS text description information, 1–8
  optimization and, 1–4
  producing, 1–4
  specifying no additional, 1–8

Defining macros
  as *defn*, 1–15
  as empty strings, 1–15

Delay slot
  following branch and subroutine call instructions, 1–7
  not reordering instructions to use, 1–7

Directories
  adding to library search list, 1–6
  searching for include files, 1–14
  standard system, not searching, 1–15

Division
  checking
    for zero division, 1–7
    the divisor and dividend, 1–10
  not checking
    for zero division, 1–7
    the divisor and dividend, 1–10

-D*macro*, 1–15

-D*macro=defn*, 1–15

Dollar signs in identifier names, 4–8

E

-E, 1–1, 1–14

Erroneous code, 2–1

Errors, 2–1

Execution time, reducing, 1–3

Expression-statements, 1–5

Expressions
  assembler instructions and, 4–9
  compound, as lvalues, 4–3
  conditional
    allowing mismatched types in, 1–12
    as lvalues, 4–3
    with omitted middle-operands, 4–4
  constructor, 4–6
  types, 4–2
    naming, 4–2

Extensions to C, 4–1

External names
  emitting two underscores before, 1–10
  not emitting underscores before, 1–10

F

-fcall-saved-*reg*, 1–14

-fcall-used-*reg*, 1–13

-fcaller-saves, 1–12

-fcombine-regs, 1–11

-fcond-mismatch, 1–12

-ffixed-*reg*, 1–13, 4–13

-f*flag*, 1–11

-ffloat-store, 1–11

-fforce-addr, 1–12

-fforce-mem, 1–11

-ffunction-cse, 1–12

File names, 1–1

-finline-functions, 1–12, 4–9

Fixed registers, 1–13

-fkeep-inline-functions, 1–12, 4–9

Flags, specifying machine-independent, 1–11

float and PCC compatibility, 3–3

Floating-point variables, not storing in registers, 1–11

-fno-asm, 1–11

-fno-defer-pop, 1–11

-fomit-frame-pointer, 1–12

-fpcc-struct-return, 1–11, 3–3

Frame pointers, not keeping in registers, 1–12

-fshared-data, 1–13

-fsigned-bit, 1–13

-fsigned-char, 1–13

-fstrength-reduce, 1–11

Function addresses, putting in registers, 1–12

Function parameters, typedef and, 3–2

Functions, 1–5
  calling special prologue and epilogue routines, 1–9
  declaring attributes of, 4–7
  external, scope of, 3–2
  implicitly declared, 1–5
  inline, 1–12, 4–8, 4–9
  not calling special prologue and epilogue routines, 1–10
  outputting separate callable versions of, 1–12
  pointers to, 4–6
  popping arguments to, 1–11
  return types of, 1–5
  returning structures or unions from, 3–3
  saving and restoring registers around calls to, 1–12
  static, 1–5
  that cannot return, 4–7

-funsigned-bit, 1–13

-funsigned-char, 1–13

-fvolatile, 1–12

-fwritable-strings, 1–12, 3–1

G

-g, 1–4

gcc program, 1–1

General public license, A–1

Generating inline code, for structures and function calls, 1–7

Global variables, 4–12

GNU, general public license, A–1

GNU C
  behaving like other versions of C, 3–1
  extensions to C, 4–1
  incompatibilities of, 3–1
  sources and documentation, v

GNU CC, 1–1
  command options, 1–1
  known troubles, 2–1

gprof (analysis program), 1–6

Grouping options, 1–1

H

-H, 1–15

I

.i suffix, 1–1

-I-, 1–14

ident directive, for identifying compiler revision, 1–8

Identifiers, matching, 1–5

Identifying, compiler revision, 1–8

-I*dir*, 1–14

Implicitly declared functions, 1–5

Include files
  outputting names of, 1–15
  searching directories for, 1–14

Incompatibilities between GNU C and other versions of C, 3–1

Inhibiting warning messages, 1–4

Initializers, non-constant, 4–6

inline, 4–8
  not recognizing as a keyword, 1–11

Inline code
  generating for structures and function calls, 1–7
  not doing block moves with, 1–7

Inline functions, 1–12, 4–9

Instructions
  ld.d, 1–8
    not using, 1–8
  reordering to use the delay slot, 1–7
  sequences for shifts, 1–8
  st.d, 1–8
    not using, 1–8

Integrating functions into their callers, 1–12

Iteration variables, eliminating for optimization, 1–11

K

Keywords
  alternate, 4–14
    macros for replacing, 4–14
  not recognizing, 1–11

L

ld.d instructions, 1–8
  not using, 1–8

-L*dir*, 1–6

Libraries
  searching for, 1–6
  standard system, not using, 1–6

License, GNU general public, A–1

Linker version, 1–1

-l*library*, 1–6

Local variables, shadowing and, 1–5

long, typedef and, 3–2

longjmp, 3–1

Loop strength reduction, 1–11

lvalues, 4–3

M

-M, 1–15

Machine-dependent options, specifying, 1–7

Macros
  alternate predefined, 1–2
  arguments in, 3–1
  defining, 1–15
    as empty strings, 1–15
  undefining, 1–15

make, rules for, 1–15

-mblock-move, 1–7

-mcheck-zero-division, 1–7

-mconst-uses-data, 1–7

-mconst-uses-text, 1–7

-mdelay-slot, 1–7

-mdouble-memory-refs, 1–8

Memory
  address constants, copying into registers, 1–12
  operands, copying into registers, 1–11
  references through pointers, 1–12

-mhandle-large-shift, 1–8

-midentify-revision, 1–8

Mismatched types, in conditional expressions, 1–12

mktmp, 3–1

-mliteral-synthesis, 1–8

-MM, 1–15

-m*machinespec*, 1–7

-mno-block-move, 1–7

-mno-check-zero-division, 1–7

-mno-const-uses-data, 1–7

-mno-delay-slot, 1–7

-mno-double-memory-refs, 1–8

-mno-handle-large-shift, 1–8

-mno-literal-synthesis, 1–8

-mno-ocs-debug-info, 1–8

-mno-ocs-frame-position, 1–9

-mno-silicon-filter, 1–9

-mno-static-literal-synthesis, 1–9

-mno-trace-function, 1–10

-mno-trap-large-shift, 1–10

-mno-underscores, 1–10

-mno-use-div-instruction, 1–10

-mocs-debug-info, 1–8

-mocs-frame-position, 1–8

-msilicon-filter, 1–9

-mstatic-literal-synthesis, 1–9

-mtrace-function, 1–9

-mtrap-large-shift, 1–10

-mtwo-underscores, 1–10

-muse-div-instruction, 1–10

-mwarn-passed-structs, 1–11

N

Naming an expression's type, 4–2

-nostdinc, 1–15

-nostdlib, 1–6

O

-O, 1–3

-o, 1–1

OCS
  canonical frame pointer, 1–8
  text description information, 1–8

Optimizations
  compilation, 1–3
  iteration variable elimination, 1–11
  strength reduction, 1–11

Options
  compiler, 1–1
  grouping, 1–1
  specifying machine-dependent, 1–7

Output, placing in a file, 1–1

P

-p, 1–6

-pedantic, 1–2, 1–14, 4–1

-pg, 1–6

-pipe, 1–2

Pipes, 1–2

Placing output in a file, 1–1

Pointers and arithmetic operations, 4–6

Prefixes and the -B option, 1–2

Preprocessing source files, 1–1

Preprocessor
  comments, 1–14
  conditionals, 3–3
  output
    make rules, 1–15
    make rules for specific include files, 1–15
    names of include files, 1–15
  version, 1–1

Printing
  compiler driver commands, 1–1
  names of include files, 1–15
  version numbers, 1–1
  warning messages, 1–4

Producing debugging information, 1–4

prof (analysis program), 1–6

Profile information, generating extra code
  for gprof, 1–6
  for prof, 1–6
  for tcov, 1–6

R

Reducing
  code size, 1–3
  execution time, 1–3

Referencing static data, 1–8, 1–9

Registers
  allocatable, 1–13, 1–14
  assembler code and names of, 4–12
  copying, 1–11
    memory address constants into, 1–12
    memory operands into, 1–11
  global variables in, 4–12
  not storing floating-point variables in, 1–11
  putting function addresses in, 1–12
  specifying fixed, 1–13

Reordering instructions, to use the delay slot, 1–7

return statements, 1–5

Returning struct and union values, 1–11

Revision, identifying, 1–8

S

-S, 1–1

.s suffix, 1–1

Scopes of external variables and functions, 3–2

Search lists, adding directories to, 1–6

Searching
  directories for include files, 1–14
  for libraries, 1–6
  specific directories, 1–15

setjmp, 3–1

Shared data, 1–13

Shifts
  four-instruction sequences for, 1–8
  no tbnd instructions before, 1–10
  single instructions for, 1–8
  tbnd instructions before, 1–10

sifilter, 1–9
  not running before calling the assembler, 1–9

Signed
  bits, 1–13
  char, 1–13

Silicon filter, 1–9
  not running before calling the assembler, 1–9

sizeof, 4–6

Source files
  assembling, 1–1
  compiling, 1–1
    without assembling, 1–1
  preprocessing, 1–1

Specifying
  machine-dependent options, 1–7
  machine-independent flags, 1–11
  no additional debugging information, 1–8

sscanf, 3–1

st.d instructions, 1–8
  not using, 1–8

Stack, 1–11

Statements, compound, in parentheses, 4–1

Static data
  not performing static literal synthesis, 1–9
  referencing
    with instruction sequences, 1–8
    with one instruction sequence, 1–8, 1–9

String constants, 1–6, 3–1
  storing in writable data segment, 1–12

Structures, 3–3
  passing to functions, 1–11
  returning values from, 1–11

Subscripting non-lvalue arrays, 4–6

Substituting macro arguments, 3–1

Supporting
  ANSI standard C programs, 1–2
  traditional C compilers, 1–3

switch statements, 1–6

T

tbnd instructions, 1–10

tcov (analysis program), 1–6

-traditional, 1–3, 3–1, 4–14

Traditional C compilers, supporting, 1–3

-trigraphs, 1–15

Trigraphs, 1–6
  supporting, 1–15

typedef, 4–2
  names as function parameters, 3–2

typeof, 4–2
  not recognizing as a keyword, 1–11

U

-U*macro*, 1–15

Undefining macros, 1–15

Underscores, 4–14
  emitting two before external names, 1–10
  not emitting before external names, 1–10

Unions, 3–3
  returning values from, 1–11

Unsigned
  bits, 1–13
  char, 1–13

Using
  alternate keywords, 1–2
  ld.d instructions, 1–8
  PCC conventions, 3–3
  pipes, 1–2
  st.d instructions, 1–8

V

-V, 1–1

-v, 1–1

Variable-length arrays, 4–5
  vs. alloca, 4–5

Variables
  automatic, 1–4, 3–1
  declaring within an expression, 4–1
  external, scope of, 3–2
  floating-point, not storing in registers, 1–11
  global, 4–12
  iteration, 1–11
  local, 1–5
    shadowing, 1–5

Version numbers, printing, 1–1

void, pointers to, 4–6

volatile, 1–12, 3–1, 4–7

W

-W, 1–4

-w, 1–4

-Wall, 1–6

Warning messages
  all options, 1–6
  casts, 1–6
  comments, 1–6
  implicitly declared functions, 1–5

Warning messages (cont.)
  inhibiting, 1–4
  issuing according to ANSI standard C, 1–14
  matching identifiers, 1–5
  printing extra, 1–4
  return types, 1–5
  shadowing local variables, 1–5
  string constants, 1–6
  structures passed to functions, 1–11
  switch statements, 1–6
  trigraphs, 1–6
  unused local variables and functions, 1–5

-Wcast-qual, 1–6

-Wcomment, 1–6

Whitespace in compound assignment operators, 3–2

-Wid-clash-*len*, 1–5

-Wimplicit, 1–5

-Wreturn-type, 1–5

-Wshadow, 1–5

-Wswitch, 1–6

-Wtrigraphs, 1–6

-Wunused, 1–5

-Wwrite-strings, 1–6

Z

Zero division
  checking for, 1–7
  not checking for, 1–7

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

   Send your order form with payment to: Data General Corporation
   ATTN: Educational Services/TIPS G155
   4400 Computer Drive
   Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME ______	COMPANY NAME ______
ATTN: ______	ATTN: ______
ADDRESS ______	ADDRESS (NO PO BOXES) ______
CITY ______	CITY ______
STATE ______ ZIP ______	STATE ______ ZIP ______

Priority Code ____________ (See label on back of catalog)

______ Authorized Signature of Buyer (Agrees to terms & conditions on reverse side) ______ Title ______ Date ______ Phone (Area Code) ______ Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

☐ UPS	ADD
1-4 Items	$ 5.00
5-10 Items	$ 8.00
11-40 Items	$ 10.00
41-200 Items	$ 30.00
200+ Items	$100.00

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

☐ UPS Blue Label (2 day shipping)
☐ Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 – $149.99	0%
$150 – $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) ______

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling – See A	+
TOTAL – See C	

C PAYMENT METHOD

☐ Purchase Order Attached ($50 minimum)
P.O. number is________. (Include hardcopy P.O.)
☐ Check or Money Order Enclosed
☐ Visa ☐ MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

______
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508–870–1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION

Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES

Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS

Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY

DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY

EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY

A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL

A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)

Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

134-755-02

moisten & seal

CUSTOMER DOCUMENTATION COMMENT FORM

Your Name ______________________ Your Title ______________

Company ______________________ Phone ______________

Street ____

City ______________________ State __________ Zip ______

We wrote this book for you, and we made certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve the manual. Please take a few minutes to respond. Thank you.

Manual Title ______________________ Manual No. ______________

Who are you?
- ☐ EDP/MIS Manager
- ☐ Senior Systems Analyst
- ☐ Engineer
- ☐ Analyst/Programmer
- ☐ Operator
- ☐ End User
- ☐ Other ______________ ______________

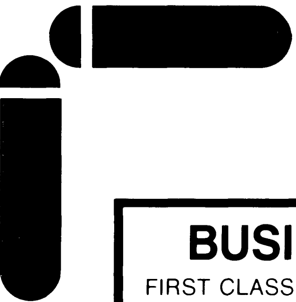
How do you use this manual? *(List in order: 1 = Primary Use)*

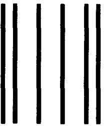
- ___ Introduction to the product
- ___ Reference
- ___ Tutorial Text
- ___ Operating Guide
- ___ Other ______________

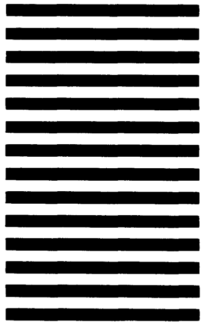
About the manual:		Yes	No
	Is it easy to read?	☐	☐
	Is it easy to understand?	☐	☐
	Are the topics logically organized?	☐	☐
	Is the technical information accurate?	☐	☐
	Can you easily find what you want?	☐	☐
	Does it tell you everything you need to know?	☐	☐
	Do the illustrations help you?	☐	☐

If you wish to order manuals, use the enclosed TIPS Order Form (USA only) or contact your sales representative or dealer.

Comments:







NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

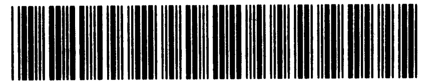
FIRST CLASS PERMIT NO. 26 WESTBORO, MA 01581

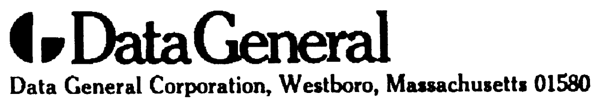
POSTAGE WILL BE PAID BY ADDRESSEE

Customer Documentation
MS E-111
4400 Computer Drive
P.O. Box 4400
Westboro, MA 01581-9890

134-755-02

Cut here and insert in binder spine pocket





Data General Corporation, Westboro, Massachusetts 01580