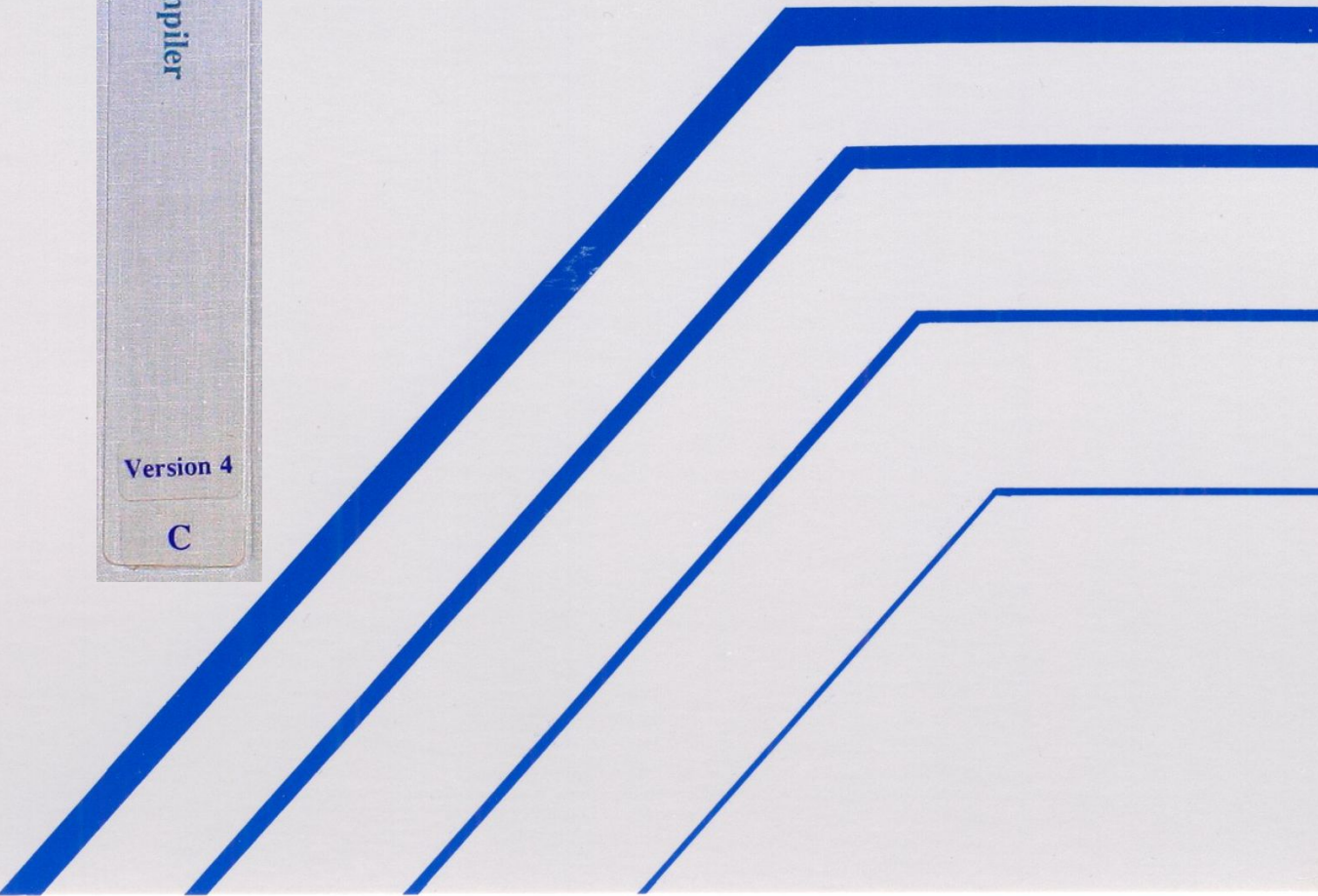
GNX Compiler

Version 4

C



Series 32000®

GNX — Version 4.0
C Optimizing Compiler
Reference Manual

Customer Order Number 424010516-004

May 1990

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
4.0	June 1990	First Release. Support for the Application Specific Instruction Set (ASIS). Function prototypes. Introduction of a source level profiler. Support for the *Series 32000/EP*.

PREFACE

This is a reference manual for National Semiconductor Corporation's GNX—Version 4 C optimizing compiler. The C optimizing compiler generates high-quality code for the *Series 32000*® architecture, therefore improving the performance of the *Series 32000* system.

The main difference between the C optimizing compiler and other compilers is the advanced optimizing component of the compiler. The optimizer uses advanced optimization techniques to improve speed or save space. This reference manual provides guidelines for using the optimizer as well as a discussion of the compiler's optimization techniques. In addition, this reference manual provides information regarding the compilation process, extensions to the C programming language, and implementation issues.

This manual corresponds to the GNX—Version 4 C compiler.

A complete list of National Semiconductor's international offices may be found on the inside back cover of this manual.

The information contained in this manual is for reference only and is subject to change without notice.

GENIX, GNX, ISE, ISE16, ISE32 and SYS32 are trademarks of National Semiconductor Corporation.

Series 32000 is a registered trademark of National Semiconductor Corporation.
UNIX is a registered trademark of AT&T.
VAX, VMS, and DEC are trademarks of Digital Equipment Corporation.

CONTENTS

FIGURES

TABLES

INDEX

Chapter 1

OVERVIEW

1.1 INTRODUCTION

This manual describes National Semiconductor's GNX—Version 4 C Optimizing Compiler. The compiler is one of a family of compatible optimizing compilers for the *Series 32000* family of microprocessors.* The GNX—Version 4 C Compiler replaces and makes obsoletes the previous GNX—Version 4 C Compiler. It implements the C language as described in *C Programming Language* by Kernighan and Ritchie, together with most of the important features in the ANSI C standard like function prototypes, const and volatile type qualifiers, and the void data type (see Section 1.3.1). The compiler is fully compatible with the System V C compiler, a compiler derived from the portable C compiler (`pcc`).

In addition, the GNX—Version 4 C Optimizing Compiler includes important extensions for programming embedded applications like ASIS (Application Specific Instructions) support, interrupt/trap handling in C, and `asm` statement. The compiler is available as a cross-support compiler running on VMS™ and UNIX® operating systems as well as a native compiler running on *Series 32000* operating systems derived from UNIX System V, Release 3. Additional information on other tools in the *Series 32000* family can be found in the *GNX — Version 4 Commands and Operations Manual*.

This manual is organized as follows:

- Introduction (Chapter 1)
- Compilation Process (Chapter 2)
- Extensions to the C language(Chapter 3)
- Implementation Issues (Chapter 4)
- Optimization Techniques (Chapter 5)
- Guidelines on Using the Optimizer (Chapter 6)
- Profile Information (Chapter 7)
- Intrinsic Routines (Chapter 8)
- *Series 32000* Calling Standard Conventions (Appendix A)

---

* At this writing, the family consists of a C Optimizing Compiler, Pascal Optimizing Compiler, and FORTRAN 77 Optimizing Compiler.

- Mixed-Language Programming (Appendix B)
- Error Messages (Appendix C)
- Compiler Options (Appendix D)
- Embedded Programming Hints (Appendix E)
- Glossary (Appendix F)

1.2 INTENDED AUDIENCE

This manual is for experienced C programmers. The information provided covers compiler options, extensions to the standard C programming language, and implementation issues. Knowledge of optimization techniques is useful but not essential; Chapter 5 provides an overview of optimization techniques used by the optimizer. And Chapter 6 provides further guidelines to help the programmer avoid problems that can occur when using the optimizer.

Recommended C reference books include:

ANSI C standard (ANSI X3.159-1989).

Harbison, Samuel and Steele, Guy. *C, A Reference Manual*, 2nd. ed., Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.

Kernighan, Brian and Ritchie, Dennis. *The C Programming Language*, 2nd, ed., Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1989.

1.3 FEATURES AND SUPPORTED LANGUAGE EXTENSIONS

1.3.1 Compiler Features

The following are the main features of the C Optimizing Compiler:

- Accepts most ANSI C features.
- `pcc` compatible.
- Allows for use of Application Specific Instructions in C via intrinsic routines.
- Allows for programming of interrupt/trap handlers in C (see Section 3.3.1).
- Optimizations can be tuned to either improve speed or save space.
- Optimization level is controlled by the user.
- Code can be generated that is tuned to the specific target system.
- Full support of mixed-language programming.
- User controlled alignment of variables and structure members.
- Improved structure handling.
- Assembly output can be annotated with source lines.
- Fast compilation mode.
- Advanced error handling, recovers from simple syntax errors.

1.3.2 Supported C Language Features

The compiler implements the full C language as defined in Appendix A of *C Programming Language* by Kernighan and Ritchie. In addition most of the ANSI C standard features and important extensions for embedded programming are supported. The following extensions are supported:

- ANSI C features:
  - — `const` for defining read-only entities.
  - — `volatile` for sensitive variables.
  - — Function prototypes.
  - — `Signed` keyword.
  - — `pragma` preprocessor command (specifically a pragma that enables marking a trap/interrupt routine).
  - — `void` data type.
  - — Structures may be assigned, passed as arguments and returned from functions.

— Initialization of automatic aggregated types.

— Structure/union member names need not be globally unique.

— Structure and union size is not limited.

— Unsigned constants (to save run-time conversions).

— Single-precision floating constants (to save run-time conversions).

— Enumeration datatypes can be used as "int".

— Unsigned or signed bitfields.

All of the above extensions are discussed in Chapter 3.

1.4 DOCUMENTATION CONVENTIONS

The following documentation conventions are used in text, syntax descriptions, and examples in describing commands and parameters.

1.4.1 General Conventions

Nonprinting characters are indicated by enclosing a name for the character in angle brackets <>. For example, <CR> indicates the RETURN key, <ctrl/B> indicates the character input by simultaneously pressing the control key and the `B` key.

Constant-width type is used within text for filenames, directories, command names and program listings; it is also used to highlight individual numbers and letters. For example,

the C preprocessor, `cpp`, resides in the `GNXDIR/lib` directory.

1.4.2 Conventions in Syntax Descriptions

The following conventions are used in syntax descriptions:

Constant-width boldface type indicates actual user input.

Italics indicate user-supplied items. The italicized word is a generic term for the actual operand that the user enters. For example,

cc [[*option*] ... [*filename*] ...] ...

Spaces or blanks, when present, are significant; they must be entered as shown. Multiple blanks or horizontal tabs may be used in place of a single blank.

{ } Large braces enclose two or more items of which one, and only one, must be used. The items are separated from each other by a logical OR sign " | ."

[ ] Large brackets enclose optional item(s).

| Logical OR sign separates items of which one, and only one, may be used.

... Three consecutive periods indicate optional repetition of the preceding item(s). If a group of items can be repeated, the group is enclosed in large parentheses "( )."

,,, Three consecutive commas indicate optional repetition of the preceding item. Items must be separated by commas. If a group of items can be repeated, the group is enclosed in large parentheses "( )."

( ) Large parentheses enclose items which need to be grouped together for optional repetition. If three consecutive commas or periods follow an item, only that item may be repeated. The parentheses indicate that the group may be repeated.

␣ Indicates a space. ␣ is only used to indicate a specific number of required spaces.

All other characters or symbols appearing in the syntax must be entered as shown. Brackets, parentheses, or braces which must be entered, are smaller than the symbols used to describe the syntax. (Compare user-entered [], with [ ] which show optional items.)

1.4.3 Example Conventions

In interactive examples where both user input and system responses are shown, the machine output is in constant-width regular type; user-entered input is in constant-width boldface type. Output from the machine which varies (*e.g.*, the date) is in italic type. For example,

```
--> g <CR>
   Breakpoint 2 reached at filename _main: .3
   .3   printf("hello\r\n");
```

1.5 Incompatibilities With GNX C Compiler Version 3

The incompatibilities with the GNX-Version 3 C compiler are:

1. `signed` keyword added.

   GNX version 4 is extended to support the ANSI C `signed` type specifier. Programs using this keyword in another context (i.e. as identifier or typedef names) are not legal and will be viewed as an error.

   Example:

   ```
   float signed;
   ```

The above example defines an identifier whose name is `signed` and will not pass compilation.

2. `Const` and `volatile` type qualifiers fully suported.

   The GNX version 3 C compiler includes a partial implementation of the ANSI C `const` and `volatile` type qualifiers. In GNX version 4 C compiler `const` and `volatile` are fully implemented as defined in ANSI C standard. Programs which rely on version 3 partial implementation may compile differently in GNX—Version 4 C .

3. `-N(/TABLES` on VMS) was deleted.

   The compiler internal tables are allocated dynamically. Hence the `-N` (`/TABLE_SIZE` on VMS) no longer has any meaning. The compiler will silently ignore this option when used.

Chapter 2

COMPILATION PROCESS

2.1 INTRODUCTION

The GNX—Version 4 C Compiler is a modular language processor consisting of five separate programs. This chapter describes the five programs, the GNX—Version 4 C Compiler invocation, options available to the user, and file generation during compilation.

2.2 COMPILER STRUCTURE

The GNX—Version 4 C Compiler's five programs are:

- Driver
- Macro preprocessor
- C language parser (front end)
- Optimizer
- Code generator

The driver is a program that parses and interprets the command line and then sequentially calls each of the other programs, depending on its input programs and the command line options.

The Macro preprocessor is the C preprocessor, `cpp`. Its input is a program file optionally containing preprocessing commands.

The C language parser is the compiler's front end, `cc_fe`. Its input is a C program. Its output is the same program in a proprietary intermediate form.*

The optimizer, `opt`, is a true global, language-independent optimizer that uses advanced optimization techniques to improve the code. Both its input and output are in the same intermediate form. See Chapter 5 for more detailed information.

---

* The intermediate form is language-independent. This allows the same optimizer and code generator to be used by all National Semiconductor GNX — Version 4 Compilers, *i.e.*, the FORTRAN 77 Compiler, the Pascal Compiler, and the C Compiler.

The *Series 32000* code generator, `cgen_coff`, generates an assembly program from a program in the intermediate form.

The assembly program produced by the code generator must be assembled by the *Series 32000* assembler to produce an object code program. The assembler is automatically called by the driver program.

The user produces an executable program by running the *Series 32000* linker on one or more object code programs with run-time library archives. On UNIX systems the linker is automatically called by the driver program. On VMS systems it must be called separately (for further details on invocation see the *GNX Commands and Operations Manual*).

2.3 COMMAND LINE OPTIONS

The GNX—Version 4 C Compiler operation is controlled by a large number of compilation parameters. Many of these parameters, such as the target system specification can be permanently set by means of the GNX Target Setup (GTS) facility. For details on how to use GTS, see the *Series 32000 GNX — Version 4 Commands and Operations* manual. All compilation parameters can be specified as command line options which override any previously existing default values.

Command line options and default values are the same for all supported host systems, but their syntax varies from host to host. Two host systems are currently supported: the UNIX operating system (in both cross-support and native variants) and the VMS operating system (cross-support only). The next two sections provide details on the various compilation parameters and their syntax on these host systems. The tables in Appendix D summarize the various compilation options of both operating systems and can be used as a quick reference.

2.3.1 UNIX Compilation Options

The invocation syntax of the GNX—Version 4 C Compiler under UNIX is:

cc [[*option*]... [*filename*]...]... (native configuration)

nmcc [[*option*]... [*filename*]...]... (cross-support configuration)

The compiler accepts a variable number of file arguments and compilation options. It produces an executable file, object file(s), or assembly file(s), according to the options specified. The files compiled are normally C program sources, but other types of files are also recognized. A file type is recognized by its suffix. A compilation option is recognized by the UNIX convention of a minus-sign prefix.

Filename Conventions

Files are identified by the compiler according to their suffix. Files with names ending with `.c` or `.i` are C source programs.

Files ending with `.c`, pass through the macro preprocessor (`cpp`) before compilation. Files ending with `.i` compile directly and assemble to produce object programs left in files whose names are those of the source files with `.o` substituted for the given suffix.

The intermediate `.o` file is deleted if a C program consisting of a single file is compiled and linked at the same time.

In support of mixed-language programming, the compiler also recognizes and compiles appropriate files written in other programming languages. Files with a `.s` suffix are assembly source programs and may be assembled (to produce `.o` files) and linked. Pascal and FORTRAN 77 source files are also recognized, and compile appropriately if your system includes the National Semiconductor GNX — Version 4 Compiler for those languages. The suffixes for these files are listed in Table 2-1. See Appendix B for details on mixed-language programming.

All other files (normally `.o` or `.a` files) are compatible object programs or archives of object programs, typically produced by previous runs of the GNX—Version 4 C Compiler, and pass directly to the linker. The object files link into one executable file with the default name `a32.out` (or `a.out` in a native-support environment).

Table 2-1. Filename Conventions

FILE NAME SUFFIX	FILE TYPE
.c .i	C source file Preprocessed C source file
.f, .for .F, .FOR	FORTRAN 77 source file FORTRAN 77 source with cpp directives
.p, .pas .P, .PAS	Pascal source file Pascal source with cpp directives
.s	Assembly source file
other (.o, .a, etc.)	Object code or library-archive file

Compiler Options

The following is a list of the compilation options which may be specified on the invocation line.* The tables in Appendix D summarize the various compilation options and can be used as a quick reference.

`-O` (PERFORMS OPTIMIZATIONS)

`-F`*flags* (SPECIFIES OPTIMIZATION FLAGS)

`-O`*flags* (PERFORMS OPTIMIZATIONS ACCORDING TO FLAGS)
The `-O` option directs the GNX—Version 4 C Compiler to perform global optimizations. The optimizer uses a variety of optimization techniques which ensure the fastest possible code. In certain cases, such as when code density is of greater importance than code speed, it is necessary to specify optimizations. Using the `-F` option with the optimization flags listed in Chapter 6 sets the selected optimization flags. Using the `-F` option by itself will do nothing. `-O`*flags* is a shorthand notation for `-O -F`*flags*. A detailed discussion of optimization techniques is found in Chapter 5 and Chapter 6.

`-Q` (COMPILES QUICK, NO CODE)
This option allows for a quick error-checking compilation. No code is generated.

`-a` (GENERATES RUN-TIME CHECKS)
This option controls the generation of code that checks for run-time errors. See Section 2.5 for more details.

`-a`*flags* (GENERATES RUN-TIME CHECKS)
This option controls the generation of code for selective run-time error checks. See Section 2.5 for more details.

`-g` (PREPARES SYMBOLIC DEBUGGING INFORMATION)
The `-g` option instructs the GNX—Version 4 C Compiler to prepare symbolic debugging information for symbolic debuggers, such as `dbug`. See the discussion on debugging of optimized code in Section 6.4. ERROR line431contains a . No matching

`-p` (PREPARES PROFILE INFORMATION FOR A PROGRAM PROFILER)
This option prepares profile information for a program profiler, such as `prof`.

`-B` (GATHER PROFILE INFORMATION)
This option instructs the compiler to add special code for profile

---

* The GNX—Version 4 C compiler supports the System V Interface Definition (SVID) for C compilers. Where possible, space is allowed between an option and its following flags, *i.e.*, `-o`*out* is the same as `-o` *out*, and `-J2` is the same as `-J 2`. Similarly, `-D`*HOST* is equivalent to `-D` *HOST*. The notation in this section follows traditional UNIX conventions.

information gathering. See Chapter 7 for more details.

-c (COMPILES BUT DOES NOT LINK)
The `-c` option directs the GNX—Version 4 C Compiler to perform the compilation process up to, but not including, linking. Output is left in object files whose names end with `.o`. This option is useful when compiling only a portion of a program's modules. For example,

```
cc -c sample.c
```

creates the file `sample.o`. No executable file is created.

-S (COMPILES AND LEAVES ASSEMBLY FILES)
The `-S` option directs the GNX—Version 4 C Compiler to terminate the compilation process before assembly. The assembly output is left in files whose names are those of the source, with `.s` substituted for the original suffix. For example,

```
cc -S sample.c utils.c
```

creates the files `sample.s` and `utils.s`. No executable or object file is created.

-n (EMBEDS C SOURCE LINES AS COMMENTS IN ASSEMBLY)
This option puts the C source lines into the assembly output file as comments. If the optimizer is enabled, explanatory optimizer comments are also put into the assembly output file. Note that the `-n` option is useful only in conjunction with the `-S` option.

-C (LEAVES COMMENTS IN)
The preprocessor normally removes the comments from its output. The **-C** option prevents this. This option can be useful when `cpp`'s output must be examined or when the `-n` option is used and C comments are required in the assembly file.

-R (PUTS LITERAL STRINGS IN READ-ONLY MEMORY)
C literal strings are, by default, writable and are thus allocated in the writable data space. The `-R` option allocates literal strings in a read-only area.

-o *out* (RENAMES THE OUTPUT FILE)
The `-o` option redirects the output file from the compilation process to a file named `out`. For example,

```
cc sample.c utils.c -o sample
```

generates the executable file `sample` from the two source files, and

```
cc -S sample.c -o new_sample.s
```

generates the assembly file `new_sample.s`.

-J*width* (ALIGNMENT WITHIN STRUCTURES)
This option allows the user to set structure-member alignment on bytes (*width* = 1), words (*width* = 2), or double-words (*width* = 4). Default value for *width* is 4 (double-word-aligned).

-w (NO WARNING DIAGNOSTICS)
The GNX—Version 4 C Optimizing Compiler normally prints warnings regarding inconsistencies in the input program. The `-w` option suppresses these warning diagnostics. See Appendix C for a complete list of the warning diagnostics.

-w66 (SUPPRESSES FORTRAN 66 WARNINGS)
This is only useful when compiling FORTRAN 77 programs.

-T (UNDEFINED VARIABLE TYPE)
This is only useful when compiling FORTRAN 77 programs.

-A (ALLOCATES VARIABLES AS STANDARD)
This option directs the compiler to adhere to the ANSI C standard, with respect to the declaration and allocation of global variables. When this option is used, there must be exactly one declaration of each global variable without the keyword *extern* within the entire program. This declaration is considered the *definition* of the variable.

-m (USES THE m4 PREPROCESSOR)
With this option, the `m4` preprocessor is used on assembly and FORTRAN 77 files before assembling and compiling them.

-d (CASE SENSITIVITY)
This is only useful when compiling Pascal and FORTRAN 77 programs.

-N[*parameter*][*size*] (SET INTERNAL TABLE SIZE)
This option is only useful for FORTRAN programs.

-v (VERBOSE)
This option lists the subprograms of the GNX—Version 4 C Compiler as they are executed by the driver program.

-vn (SHOWS BUT DOES NOT ACTUALLY EXECUTE)
This option lists the compiler subprograms that are called by the compiler's driver program, without actually executing them. This option can be used to verify how other compiler options work.

-K*parameter* (SETS TARGET CPU, FPU, OR BUSWIDTH)
The `-K` option allows the user to "tune" the GNX—Version 4 C Compiler by specifying the CPU, the FPU (or absence of), and/or buswidth of the target system. See Sections 2.4 and 2.6 for more details.

-Z*c* (USES ALTERNATIVE LIBRARY)
This option directs the compiler to link an alternative library and

initialization file, determined by the character which follows the option. For example,

```
cc -Z2 unix.c
```

links `unix.o` with `crt2.o` and `lib2.a`.

-X (GENERATES MODULAR CODE)
This option directs the compiler to generate code that conforms to the *Series 32000* architectural feature of modularity (which allows the modular use of external references). For further information see the *Series 32000 GNX — Version 4 Language Tools Technical Notes* and the *Series 32000 Programmer's Reference Manual*.

-f (FLOATING-POINT EMULATION)
This option tells the compiler driver that there is no FPU on the target and floating-point emulation is desired. See Section 2.6 for a discussion of this option and floating-point emulation.

The compiler accepts the following options and passes them to the C preprocessor.

-D*name*[=*def*] (DEFINES)
The `-D` switch defines *name* equal to *def* to the preprocessor. If no explicit value is given, *name* is defined as having the value 1. The use of this option is equivalent to putting a "`#define` *name def*" at the beginning of each C source file.

For example:

```
cc -DHOST=VAX sample.c
```

works as if the following define was at the head of `sample.c`:

```
#define HOST     VAX
```

-E (RUNS `cpp` ONLY)
This option terminates the compilation after preprocessing; only the `cpp` preprocessor is invoked, and its output is sent to the standard output, `stdout`.

-I*dir* (SPECIFIES DIRECTORY FOR INCLUDED FILES)
This option tells to use the specified directory as the default directory for included files. Include files that are called using double quotes, *e.g.*, `#include "`*filename*`"`, are sought first in the directory of the compiled file, then in the directories specified by `-I`, and finally in directories on a standard list (`/usr/include`). If the user explicitly names the file to be included by using the complete path, *e.g.*, `#include "/a/mydir/filename"`, the named file is sought directly. If angle brackets are used instead of double quotes, *e.g.*, `#include <`*filename*`>`, the file is sought in the directories on a standard list (`/usr/include`).

-M (RUNS `cpp` ONLY, GENERATES MAKEFILE DEPENDENCIES)
This option runs only the `cpp` macro preprocessor on the named C programs, requests it to generate makefile dependencies and then sends the result to the standard output, `stdout`. For example:

```
cc -M *.c > new.makefile
```

runs `cpp` on all of the C programs in the current directory and generates all makefile dependencies. These dependencies are then sent to the file `new.makefile`.

-P (RUNS cpp ONLY, REDIRECTS OUTPUT TO .i FILE)
This option is similar to `-E`, except that the output of `cpp` is sent to a file with a `.i` extension. For example:

```
cc -P sample.c utils.c
```

creates the files `sample.i` and `utils.i`.

-U*name* (UNDEFINES)
Using this option is equivalent to putting "#undef *name*" at the beginning of each C source file.

In addition, the compiler accepts the following compiler options and passes them to the linker. See the *GNX — Version 4 Linker User's Guide* manual for details.

-v (LINKER VERSION)

-l*lib* (SPECIFIES A PROGRAM LIBRARY)

-s (STRIPS THE EXECUTABLE FILE OF SYMBOL TABLE AND RELOCATION BITS)

-r (RETAINS RELOCATION)

-u *symname* (UNDEFINES SYMBOL IN SYMBOL TABLE)

-e *epname* (DEFINES ENTRY POINT)

-x (NO LOCAL SYMBOLS IN OUTPUT SYMBOL TABLE)

-i (RUN-TIME INITIALIZATIONS)

The following option can be used as an "escape" to pass additional options (not recognized by the GNX—Version 4 C Compiler) to the C preprocessor, assembler, or linker.

-W*x*,*options* (PASSES OPTIONS TO COMPILATION PHASE *x*)
This option passes options to the C preprocessor (*x* = p), the assembler (*x* = a), or the linker (*x* = l). The *options* must be a single argument (no embedded space, unless quoted). For example, the command,

```
cc -Wl,-mmu382 sample.c
```

passes the option `-mmu382` to the linker.

2.3.2 VMS Compilation Qualifiers

The command line invocation syntax of the GNX—Version 4 C Compiler is as follows:

nmcc [*qualifier*]... *filename*

The normal operation of the GNX—Version 4 C Compiler compiles and assembles a file specified on the command line to create an object file. Command qualifiers (preceded by a /) are applied as necessary. Most qualifiers can be preceded by NO to reverse their function. The usual VMS conventions regarding default filename extensions, case insensitivity, qualifier syntax and abbreviation rules apply. The GNX—Version 4 C Compiler accepts only one C source file as input and produces an object file with optional intermediate results (such as an assembly file). If the source file has no extension, a .C extension is assumed.

The following is a list of the compilation qualifiers which may be specified on the invocation line.
The tables in Appendix D summarize the various compilation qualifiers and can be used as a quick reference.

/[**NO**]**OBJECT**[=*filename*]

This qualifier directs the compiler to leave the object code in a file named *filename*. If *filename* has no suffix, .OBJ is added as a suffix. If *filename* is not specified, the object code is placed in a file with the source's filename, with the .OBJ suffix substituted for the original suffix. Default of this qualifier is /OBJECT. For example,

```
NMCC/OBJ=NEW_UTILS.OBJ  UTILS.C
```

compiles the file utils.c, and leaves the result in a file called new_utils.obj.

The command:

```
NMCC/NOOBJ/ASM/OPT/ANNO  SAMPLE.C
```

results in an annotated, optimized assembly translation of sample.c and does not generate an object file.

The command **NMCC/NOOBJ x.c** results in a quick compilation of x.c without producing any output. This is useful for error checking.

/[**NO**]**OPTIMIZE**[=(*flags*)]

This qualifier directs the GNX—Version 4 C Compiler to perform global optimizations. A detailed discussion of the GNX—Version 4 C Compiler optimization techniques is located in Chapter 5 and Chapter 6. Default is /NOOPTIMIZE.

/[NO]CHECK

This qualifier controls the generation of code that checks for run-time errors. Default is /NOCHECK. See Section 2.5 for more details.

/[NO]DEBUG

The /DEBUG qualifier instructs the GNX—Version 4 C Compiler to prepare symbolic debugging information for symbolic debuggers, such as DBUG. See the discussion on debugging of optimized code in Section 6.4. Default is /NODEBUG.

/[NO]GATHER (GATHER PROFILE INFORMATION)

This qualifier instructs the compiler to add special code for profile information gathering. The default is /NOGATHER. See Chapter 7 for more details.

/[NO]ASM[=*filename*]

This qualifier directs the compiler to leave the intermediate assembly file in a file named *filename*. If *filename* has no suffix, .ASM is added as a suffix. If *filename* is not given, the source filename is used substituting the .ASM suffix with the source filename's suffix. Default of this qualifier is /NOASM. For example,

```
NMCC/ASM=NEW_UTILS.ASM UTILS.C
```

compiles the file UTILS.C, and produces NEW_UTILS.ASM and UTILS.OBJ.

/[NO]ANNOTATE

This qualifier directs the compiler to put GNX—Version 4 C source lines as comments into the assembly output file. If the optimizer is enabled, explanatory optimizer comments are also added into the assembly output. Note that this qualifier is useful only in conjunction with the /ASM qualifier. Default is /NOANNOTATE.

/[NO]ROM_STRINGS

C literal strings are, by default, writable and are thus allocated in the writable data space. This qualifier directs the compiler to put all literal strings in read-only memory.

/ALIGN[=*width*]

This qualifier allows the user to set structure member alignment on bytes (*width* = 1), words (*width* = 2), or double-words (*width* = 4). Default value for *width* is 4 (double-word-aligned). See Section 4.2.4 for details of the GNX—Version 4 C Compiler's alignment scheme.

/[NO]WARNING

The GNX—Version 4 C Compiler prints warnings regarding inconsistencies found in the input program. The /NOWARNING qualifier suppresses these warning diagnostics. Default is /WARNING. See Appendix C for details on warning diagnostics.

/[NO]STANDARD

This qualifier directs the compiler to adhere to the draft-proposed ANSI C standard, with respect to the declaration and allocation of global variables. When /STANDARD is used, there must be exactly only one declaration of each global variable without the keyword extern within the entire program. This declaration is considered the "definition" of the variable. Default is /NOSTANDARD.

/TABLE_SIZE=(*table_name=size*[,...])

This option is only useful for compiling FORTRAN programs.

/[NO]VERBOSE

This qualifier lists the parts of the GNX—Version 4 C Compiler as they are called by the driver program. Default is /NOVERBOSE.

/[NO]VN

With this qualifier, the compiler lists the subprograms that are called by the driver program, without actually executing them. This qualifier can be used to verify how the other qualifiers work. Default is /NOVN.

/TARGET=(CPU=*cpu*,**FPU=***fpu*,**BUSWIDTH=***bus*)

The /TARGET qualifier allows the user to "tune" the GNX—Version 4 C Compiler by specifying the CPU, the FPU (or absence of), and/or buswidth of the target system. See Sections 2.4 and 2.6 for more details.

/[NO]MODULAR

This qualifier directs the compiler to generate code that conforms to the *Series 32000* architectural feature of modularity (which allows the use of external references). For further information see the *Series 32000 GNX — Version 4 Language Tools Technical Notes* and the *Series 32000 Programmer's Reference Manual*. Default is /NOMODULAR.

/[NO]ERROR[=*filename*]

The /ERROR qualifier instructs the GNX—Version 4 C Compiler to direct compilation error messages to an error log file in addition to the standard output. If *filename* has no suffix, the suffix .ERR is added. If no destination file is given, the source filename is used, substituting .ERR for the source filename's suffix. Default sends the errors to the standard output only. For instance,

```
NMCC /ERROR=FILE1 FILE1.C
```

creates an error log file named FILE1.ERR.

/[NO]PRE_PROCESSOR

This qualifier causes the source file to be passed to the GNX C preprocessor before the normal processing by the GNX—Version 4 C language parser.

Default is /PRE_PROCESSOR.

In addition, the compiler recognizes the following compiler qualifiers and passes them to the C preprocessor. These qualifiers must be used in conjunction with the /PRE_PROCESSOR qualifier.

/DEFINE=(*name*[=*def*] [,...])

The use of this option is equivalent to putting a #define *name def* at the beginning of the C source file. The /DEFINE switch defines *name* equal to the value *def* to the preprocessor. If no explicit value is given, *name* is defined as having the value 1. For example:

```
NMCC/PRE_PROCESSOR/DEFINE=("VAX", "TARGET_IS_NS32000") SAMPLE.C
```

works as if the following two defines were at the head of SAMPLE.C:

```
#define VAX    1
#define TARGET_IS_NS32000    1
```

/[NO]COMMENT

The preprocessor normally removes the comments from its output. The /COMMENT qualifier prevents this. This qualifier is useful when cpp's output must be examined or when the /ANNOTATE qualifier is used and C comments are required in the assembly file. Default is /NOCOMMENT.

/[NO]EXPAND[=*filename*]

This qualifier controls whether the output of the preprocessor is saved to a file. If *filename* has no suffix, the suffix .MAC is added. If *filename* is not given, the source file name is used substituting the suffix .MAC for the source file name's suffix. (Default is /NOEXPAND.)

/INCLUDE=(*include_dir* [,...])

This qualifier tells the cpp preprocessor to use the specified directory as the default directory for included files. Include files that are specified using double quotes, *e.g.*, #include "*filename*", are sought first in the directory of the compiled file, then in the directories specified by the /INCLUDE option, and finally in directories on a standard list (GNXDIR:INCLUDE). If the user explicitly names the file to be included by using the complete path, *i.e.*, #include "[MYDIR]filename", the named file is sought directly. If angle brackets are used instead of double quotes, *e.g.*, #include <*filename*>, the file is sought in the directories on a standard list (GNXDIR:INCLUDE).

/UNDEFINE=(*name* [,...])

Using this qualifier is equivalent to putting #undef *name* at the beginning of each C source file.

2.4 TARGET MACHINE SPECIFICATION

The compiler provides a way for the user to tune the code for a specific target system by specifying its CPU, FPU and buswidth. This tuning is performed by setting permanent defaults using the GNX Target Setup (GTS) facility, or by specifying `-K` (`/TARGET` on VMS) on the command line. Table 2-2 lists the flags and the possible settings. The values for the CPU and FPU can either be the complete device name *e.g.*, `NS32332` or `NS32081`, or the last characters of the device name, *e.g.* `332` or `cg16`. The absence of an FPU on the target system can be indicated by specifying `emulation` or `nofpu` (for more details see Section 2.6). The buswidth is specified in bytes.

Table 2-2. Target Selection Parameters

CPU (C)	FPU (F)	BUSWIDTH (B)
[NS32]008	[NS32]081	1
[NS32]016	[NS32]181	2
[NS32]cg16	[NS32]381	4
[NS32]fx16	[NS32]580	
[NS32]cg160	emulation	
[NS32]032	nofpu	
[NS32]332		
[NS32]532		
[NS32]gx32		
[NS32]gx320		

Example: The following example specifies an NS32CG16 CPU, an NS32081 FPU, and a buswidth of 4 bytes.

UNIX

```
nmcc -KCcg16 -KF081 -KB4 temp.c (cross-support)
or cc -KCcg16 -KF081 -KB4 temp.c
```

VMS

```
NMCC /TARGET=(CPU=cg16,FPU=081,BUS=4) TEMP.C
```

2.5 RUN-TIME CHECKS

Run-time checks detect and report run-time errors. The compiler by default does not generate code to perform run-time checks. If run-time checks are required, they can be turned on selectively or all at once on the command line by using the `-a` option on UNIX systems (`/CHECK` qualifier on VMS).

The `-a` option (`/CHECK` qualifier on VMS) causes all run-time checks to be performed. The full syntax for UNIX is:

-a*flags*

And for VMS:

/ [NO]CHECK [= (*flags* [,...])]

By adding flags, only specified checks are performed. Table 2-3 lists the flags for each run-time error check.

Table 2-3. Run-time Check Flags

UNIX	VMS	CHECK PERFORMED
p	PARAMETER	Intrinsic routines parameters
i	INDEX	Index exceeding array bounds
n	NIL_POINTER	Dereferencing through a pointer to the 0 address

An example for generating all checks in the UNIX environment is:

```
cc -a x.c
nmcc -a x.c
```

An example for generating only index and NIL pointer checks is:

```
cc -ain x.x
nmcc -ain x.c
```

When a run-time error occurs, a detailed message is displayed describing the file, error, and line at which the error occurred is displayed. The program terminates after the error information is displayed.

2.5.1 Parameter Check

The parameter check option generates code to check for incorrect parameter values on calls to intrinsic routines (see Chapter 9). The following calls are checked:

- `mask1` and `mask2` parameters in `bitblt` routines. The value of the actual `mask1` and `mask2` parameters in a call to a `bitblt` routine must be in the range of 0 to the maximum unsigned value of a word (65535).
- `shift_val` parameter in `bitwt` routines. The value of the actual `shift_val` parameter in a call to a `bitwt` routine must be in the range of 0 to 15.
- `length` in `ext` and `ins` routines. The value of the actual `length` parameter in calls to `ext` and `ins` routines must be in the range of 1 to 32.
- `src_addr` and `dest_addr` in `extblt` routines. The value of the actual `src_addr` and `dest_addr` parameters in calls to a `extblt` routine must be an even number.
- `width` in `extblt` routines. The value of the actual `width` in calls to a `extblt` routine must be an even number and a multiple of the value of the actual `horiz_inc` parameter.
- `horiz_inc` in `extblt` routines. The value of the actual `horiz_inc` parameter in calls to a `extblt` routine must be either (+2) or (-2).

2.5.2 Array Checks

Each array index is checked to be within the array bounds (i.e. greater or equal to 0 and less then the array's dimension).

For example, the following code:

```
main(){
    int index,array[5];
    index = 6;
    array[index] = 1;
}
```

will result in run-time in the error message

```
"bad.c", line 5: value of 6 is out of bounds
```

NOTE: Index run-time checks are generated only for arrays whose dimensions are known during the compilation of the file.

2.5.3 NIL_POINTER Checks

Whenever a pointer is dereferenced, a check is performed for NIL pointers. If a NIL pointer is dereferenced, an error message results.

For example, the following code:

```
main(){
      char *ptr; = ((char *) 0);
      *ptr = 1;
}
```

results in the error message in run-time

```
"badptr.c", line 4: trying to dereference through a NIL
pointer
```

2.6 FLOATING-POINT EMULATION

Two different floating point emulation options are available with the GNX—Version 4 C Compiler: `Hfp` and `fpee`. Additional information, such as the difference between these options and the way they are implemented, can be found in Chapter 6 of the *Series 32000 GNX–Version 4 Support Libraries Reference Manual*. The use of the `Hfp` package is indicated by the `-KFemulation` compiler option (`/TARGET=(FPU=emulation)` on VMS). The `Hfp` package may be used for cross configuration only. The use of the `fpee` package is indicated by the `-f` or `-KFnofpu` compiler option (`/TARGET=(FPU=nofpu)` on VMS). The `fpee` package may be used for cross configuration and for IEEE compatibility in native configuration. These options may also be set permanently by using the GTS facility.

2.6.1 Floating-point Emulation — Native Configuration

There is no way to unconfigure the FPU on the SYS32/50 and no floating-point emulation is therefore required. To use the `fpee` library you must do the following:

1. Include a call to the library routine `fpinit_` at the beginning of the main module.
2. Include a `-lfpe` field after the source and object module in the "compile" command. For example,

   ```
   cc file1.c -lfpe -lm
   ```

   where `file1.c` is the input source file.

2.6.2 Floating-point Emulation — Cross-Configuration

In Cross-Configuration (UNIX system), floating-point emulation is achieved by using either the `-f` option on the `nmcc` invocation line or including a call to the `INIT__` routine prior to any floating-point operations and explicitly linking files and libraries.

When `-f` is used on the `nmcc` invocation line the cross-compiler driver:

- assumes there is no FPU on the target system
- assumes that the user wants to use floating-point emulation
- generates the correct command line and passes this to the linker

For example:

```
nmcc -f file1.c
```

The following is an example of explicitly linking files and libraries:

In cross host:

```
nmcc -c file1.c
nmeld GNXDIR/lib/fcrt0.o file1.o -lfpe -lm -lc
```

In native host (*Series 32000*/UNIX system):

```
nmcc -c file1.c
ld GNXDIR/lib/db_fcrt0.o file1.o -ldb_c -ldb_fpe
```

2.6.3 Floating-Point Emulation — VAX/VMS System

Files and libraries must be explicitly linked to achieve floating-point emulation on a VAX/VMS system. This is a two-step process:

```
nmcc file1.c
nmeld gnxdir:fcrt0.obj, file1.obj, gnxdir:libfpe.a, gnxdir:libc.a
```

2.7 ENVIRONMENT VARIABLES (FOR UNIX ONLY)

On UNIX systems, in addition to the command line options, the compiler accepts several implicit options. These can be set through the environment variables CMDDIR, TMPDIR, LIBPATH, PITFILE, and INCLUDEPATH which are described below:

CMDDIR

The environment variable CMDDIR can be given the value of a directory name, in which the driver looks for the indirectly called programs (`cpp`, `cc_fe`, `opt`, etc.). For example, if `CMDDIR = ''/usr/nsc/lib''`, the driver will look for `/usr/nsc/lib/cpp`, `/usr/nsc/lib/cc_fe`, etc.

TMPDIR

This environment variable redefines the location at which temporary files are created in the compilation process: Default is `/tmp`. This environment variable should be used on small systems with tiny `/tmp` partitions, which overflow when compiling huge files.

LIBPATH

The environment variable LIBPATH can be defined to contain one or more directories (separated by ":"). If `LIBPATH` is defined, then libraries will be taken from one of these directories. For example, if `LIBPATH = /usr/mylib:/usr/yourlib`, then libraries will be in either `/usr/mylib` or `/usr/yourlib`.

PITFILE

The environment variable PITFILE is used to redefine the default filename for profile information table file (PIT) used by sprof and the compiler. See Chapter 7 for more details.

INCLUDEPATH

If the INCLUDEPATH variable is defined (in a similar format as `LIBPATH`), the standard include files (such as `<stdio.h>`) will be searched for in its directories

AVAIL_SWAP

The environment variable AVAIL_SWAP sets the maximum swap sapce of the optimizer in megabyte units. AVAIL_SWAP should be set to the number of megabytes to be used. See Section 6.7 for use of the AVAIL_SWAP environment variable.

Chapter 3

EXTENSIONS TO THE C LANGUAGE

3.1 INTRODUCTION

The GNX—Version 4 C compiler is based on the UNIX portable C compiler, `pcc`. All `pcc` extensions to the C language (as defined by Kernighan and Ritchie) are implemented by the GNX—Version 4 C compiler. In addition, the compiler includes two main types of extensions:

1. ANSI C features - Most non pre-processor features of the ANSI C Standard are implemented.
2. Embedded support extensions - Special features to assist programming embedded applications.

This chapter describes the extensions implemented by the GNX—Version 4 C compiler. Section 3.2 reviews the ANSI C extensions. Section 3.3 describes the embedded support extensions. All other extensions are presented in Section 3.4.

3.2 ANSI FEATURES

This section describes ANSI C features implemented in the GNX - Version 4 C compiler. For more details see *C - A Reference Manual* (second edition) by Harbison and Steele, and the ANSI C standard.

3.2.1 Function Prototypes

Function prototypes are fully implemented.

3.2.2 Volatile and Const Qualifiers

`volatile` and `const` type qualifiers are fully supported. See Appendix E for more details.

3.2.3 Void Data Type

The `void` data type is used as the type mark for a function that returns no result. It may also be used in any context where the value of an expression is discarded to explicitly indicate that a value is ignored. This is done by writing a cast to void.

The type `void *` is used for the generic pointer and is compatible with other pointer types.

3.2.4 Signed Keyword

The `signed` keyword is recognized by the compiler.

3.2.5 The #pragma Directive

The `#pragma` directive is recognized by the preprocessor and by the compiler. However, only the use of `#pragma` for interrupt/trap routines will be recognized by the compiler. Any other use of the `#pragma` directive will be ignored by the compiler.

3.2.6 Single-Precision Floating Constants

These floating constants allow the explicit specification of constants as single-precision in order to eliminate wasteful run-time conversions. This is accomplished by appending an `f` suffix to a float constant.

Example:

```
fmax += 17.0f
```

The same effect can be achieved by casting the constant to float, as in `fmax += (float)17.0;`. Not using either the suffix or the cast results in both `fmax` and the value `17.0` being converted to double-precision for a double-precision addition; with the result being converted back to single-precision.

3.2.7 Unsigned Constants

Unsigned constants allow the explicit specification of unsigned constants. This is accomplished by appending a `u` suffix to a positive integer constant.

Example: `"65u"`

As with single-precision floating constants, unsigned constants eliminate wasteful run-time conversions.

3.2.8 Enumerated Types

Enumerated types as defined in ANSI C standard are fully supported. In addition, a warning is issued on assignment of different enumeration.

3.2.9 Structure Handling

The GNX—Version 4 C compiler implements the following improvements to structure handling:

- structure assignment
- structures as function arguments and return values
- reuse of structure and union member names
- initialization of first member of a union
- initialization of `auto` storage class structures

NOTE: Unlike initialization of automatic scalar variables , initialization of automatic variables is limited to initializers known at compile time.

3.2.10 Concatenation of Adjacent String Literals

According to the ANSI C standard, string literals that are adjacent tokens are concatenated into one character string literal.

For example the following code:

```
char s = "hello "
             "world";
printf(s);
```

prints the message:

```
hello world
```

3.2.11 Obsolesce of the Old Fashioned Compound Assignment

Since old fashioned compound assignment syntax is obsolete in ANSI C, it is no longer recognized by the GNX compiler.

For example, the following line:

```
int_var =+ 5;  /* used to be equivalent to 'int_var += 5' */
```

is flagged as an error by the compiler.

3.2.12 Obsolesce of the Old Fashioned Initialization

Since the old fashioned initialization syntax is obsolete in ANSI C, it is no longer recognized by the GNX compiler.

For example, the following code:

```
int int_var 14;   /* used to be equivalent to 'int int_var = 14;'
*/
```

is flagged as an error by the compiler.

3.3 EMBEDDED SUPPORT EXTENSIONS

3.3.1 Interrupt/Trap Routines Support

As part of the embedded support, the GNX C compiler enables programming of trap and interrupt handlers in C. Handlers are defined as functions in the regular C syntax, preceded by a `#pragma` directive used to mark these functions as trap/interrupt handler routines.

Special code is produced by the compiler for the enter and exit sequence of routines marked as interrupt/trap handlers. This code is responsible for saving the proper registers (i.e. all registers used by the routine and scratch registers if the routine calls another routine) when entering an interrupt/trap routine. When the routine is exited, the saved registers are restored and `RETI` (for interrupts) or `RETT` (for traps) is performed (see the *Series 32000 Programmer's Reference Manual* for further details).

This section describes the syntax and semantics of writing interrupt/traps handlers in the GNX C Compiler. See Appendix E for more details.

INTERRUPT/TRAP HANDLER DEFINITION

The interrupt/trap handler is written as a regular C routine in the usual C function definition syntax. For example:

```
void hndlr_foo(void)
     {
        printf("division by zero");
        exit (1);
  };
```

The function is designated as an interrupt/trap handler in the following manner (the `#pragma` is used to mark an interrupt/trap handler).

Syntax for interrupts:

`#pragma interrupt` (*function_name* [,save_regs={int_regs | all_regs}])

Syntax for traps:

`#pragma trap` (*function_name* [,save_regs={int_regs | all_regs}])

`function_name` is the name of the function to be marked as an interrupt/trap handler. `save_regs` can be either `all_regs` (save all registers for general purpose and floating point), or `int_regs` (save only general purpose registers).

In many applications the interrupt/trap handlers do not perform floating-point operations. In such applications there is no need to save the scratch floating point registers. The option `save_regs` enables you to specify the register type to be saved (when the handler calls another routine). The default (if `save_regs` is omitted) is `int_regs`. Options different from `all_regs` or `int_regs` are considered errors.

NOTE: Only the registers used in the interrupt/trap routine (and the scratch registers if the interrupt/trap calls another function) are saved.

A warning is issued by the compiler if a function is marked as an interrupt/trap handler using the `#pragma` directive, but no definition of the function was found in the compiled module.

Multiple `#pragma` directives with the same function name are considered errors, unless they are identical.

Restriction

The `#pragma` directive must appear before any declaration or definition of the function. The placement of the `#pragma` interrupt/trap in any other location results in an error message.

USING INTERRUPT/TRAP HANDLERS

It is your responsibility to install the address (or descriptor) of the interrupt/trap handler in the proper entry of the interrupt dispatch table (see the *Series 32000 Development Board Monitor Reference Manual* and the example presented in Appendix E for further information).

Calling an interrupt/trap handler directly from within the C code is not permitted. Any attempt to do so causes an error. This is because different instructions are used for returning from the interrupt/trap routine (`RETI/RETT`) and for returning from a regular routine (`RET`).

Attempts to call an interrupt/trap routine from within the C code is detected by the compiler only for calls in the same module in which the interrupt/trap routine was defined. All other calls are not detected by the compiler.

The default exception mode is direct-exception. In order to make non direct-exception mode possible, you should insert an `asm` statement ".module" before the function definition. It is your responsibility to do so. For more details see Appendix E.

3.3.2 Asm Keyword

The keyword `asm` is recognized for the insertion of assembly instructions directly into the generated instruction stream. The syntax is

asm (*constant-string*) ;

where *constant-string* is a double-quoted character string.

The keyword `asm` can be used within functions as a statement and outside of functions as a global declaration. A newline character will be appended to the given string without causing any change in the assembly code. See Appendix E for a detailed example.

3.3.3 Intrinsic Routines

The compiler enables the use of *Series 32000/EP* application specific instructions without the need of the `asm` keyword, by recognizing a set of intrinsic functions known internally to the C compiler. These intrinsic functions are used in the code as regular C functions, but are translated to an instruction sequence containing the special instructions and not to a function call. See Appendix E for more details.

3.4 OTHER EXTENSIONS

3.4.1 $ Sign in Identifiers

The GNX—Version 4 C compiler allows the use of `$` signs in identifier names.

3.4.2 Bitfields

The GNX—Version 4 C compiler implements signed, unsigned, int, short, and char bitfields. Due to the *Series 32000* architecture, the code for unsigned bitfields is more efficient than the code for signed bitfields.

3.4.3 Ident Preprocessor Command

A new cpp-style directive is recognized for placing strings into the `.comment` section of the object file. The syntax is

`#ident` *constant-string*

where *constant-string* is a double-quoted character string. The string is passed to the assembly file with a `.ident` directive and placed by the assembler in the `.comment` section of the object file.*

---

* See the *Series 32000 GNX — Version 4 COFF Programmer's Guide* and the *Series 32000 GNX — Version 4 Assembler Reference Manual* for a description of the `.comment` section and the `.ident` directive.

Chapter 4

IMPLEMENTATION ISSUES

4.1 INTRODUCTION

This chapter describes compiler implementation aspects which may differ from other compilers and which may affect code portability.

Portability issues are recognized by the C standard as issues that may differ from one implementation to another. The following two sections discuss portability issues. Section 4.2 defines how the GNX—Version 4 C compiler behaves under the listed issues. Section 4.3 lists issues that cause an undefined behavior of the GNX—Version 4 C compiler.

4.2 IMPLEMENTATION ASPECTS

The following cases are aspects of this implementation.

4.2.1 Memory Representation

- The representation of the various C types in this compiler are

C TYPE	SERIES 32000 DATE TYPE
int	32-bit double-word
long	32-bit double-word
short	16-bit word
char	8-bit byte
float	32-bit single-precision floating-point
double	64-bit double-precision floating-point

- The set of values stored in a `char` object is signed.
- The padding and alignment of members of structures as described in Section 4.2.4.
- A field of a structure can generally straddle storage unit boundaries.
- While signed bitfields are implemented, it is not recommended to use them since their implementation is slow. Bitfields are not allowed to straddle a double-word boundary.

4.2.2 External Linkage

- There is no limit to the number of characters in external names.
- Case distinctions are significant in an identifier with external linkage.

4.2.3 Types and Conversions

- A right shift of a signed integral type is arithmetic, *i.e.*, the sign is maintained.
- When a negative floating-point number is converted to an integer, it is truncated to the nearest integer that is less than or equal to it in absolute value. The result is returned as a signed integer.
- When a double-precision entity is converted to a single-precision entity, it is converted to the nearest representation that will fit in a `float` with default rounding performed to the nearest value.
- The presence of a `float` operand in an operation not containing double operands causes a conversion of the other operand to `float` and the use of single-precision arithmetic. If double operands are present, conversion to double occurs.

4.2.4 Variable and Structure Alignment

The alignment of entities in a program is a trade-off issue. Most *Series 32000* CPUs are more efficient when dealing with entities aligned to a double-word boundary. This normally makes it necessary to have some amount of padding added to a program. This padding represents an overhead in storage space.

The GNX—Version 4 C compiler allows the user to tailor the alignment of structures/unions and their members and, independently, the alignment of other variables. Function parameters are always double-word aligned. This allows the calling of functions across modules without dealing with alignment issues.

Alignment of Variables

`Extern`, `static`, and `auto` variables are aligned in memory according to their size and the buswidth setting. Table 4-1 lists variable size, buswidth, and the alignment determined by these two parameters.

A buswidth setting of 1 means "align to 1 byte." Variables start on a byte boundary, in other words, there is no alignment and no padding. When allocating storage for variables, bytes are allocated sequentially with no padding between bytes.

Variables of size 1 are of the C type `char`, variables of size 2 are of the C type `short`, and variables of size 4 or greater are of the C types `int`, `long`, `float`, and `double` (size 8).

Table 4-1. Variable Alignment

BUS WIDTH	VARIABLE SIZE (BYTES)		
	1	2	>= 4
1	byte	byte	byte
2	byte	word	word
4	byte	word	double-word

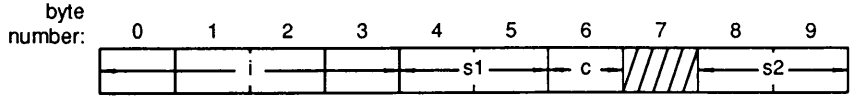
A buswidth setting of 2 means "align to an even byte." Variables that are larger than 1 byte start on a word boundary. This means that there may be padding of single bytes.

A buswidth setting of 4 means "align to a double-word boundary" (a byte whose address is divisible by four). Variables that are 2 bytes long start on a word boundary; variables that are 4 bytes or larger in size start on a double-word boundary. This means that there may be padding of up to three bytes.

Arrays are aligned as the alignment of their element type. Structures are aligned according to the alignment of the largest structure members. This is affected by the `-J (/ALIGN)` option. See "Structure/Union Alignment" and "Allocation of Bit-Fields" for more details.

Example: The arrangement of `int i; short s1; char c; short s2;`

with a buswidth of 2 or 4 is



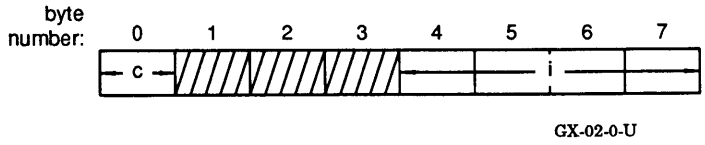
GX-01-0-U

Note that to align `s2` to a word boundary, padding space of one byte is needed after `c`. This padding does not exist with a buswidth of 1.

Example: The arrangement of

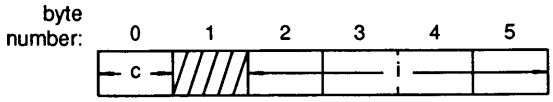
```
char c; int i;
```

with a buswidth of 4 is



GX-02-0-U

With a buswidth of 2, the arrangement is



GX-03-0-U

With a buswidth of 1, there is no padding.

It is important to note that the order in memory is the same as the declaration order only for `extern` and `static` variables. The optimizer may reorder `auto` variables in order to minimize padding space.

Fastest code is achieved by setting the default alignment to that of the data buswidth of the CPU (for all but the NS32008, the NS32CG16, the NS32FX16, the NS32CG160 and the NS32016). This can be accomplished by setting the BUS parameter in the target specification file, or by overwriting that file on the command line with the `-KB` (`/TARGET`) option.

Structure/Union Alignment

Structure members are aligned within the structure, relative to the beginning of the structure, in the same way that variables are aligned in memory. In order to maintain the alignment of the members relative to memory, the structure itself is aligned in memory according to the alignment of its largest members. This alignment may be controlled by putting `-J` (`/ALIGN`) on the command line.

In addition, the total size of a structure is such that it also ends on an alignment boundary of its largest member. This maintains the alignment of individual members in arrays of structures. This is illustrated in the `FILE struct` example at the end of this section.

For unions, there is no padding. The alignment of the union's largest members determine the alignment of the union itself.

Allocation of Bit-Fields

To understand the way bit-fields are handled, think of the situation where a field is fetched from memory. The number of bits fetched is determined by buswidth. For instance, if a bus is 2-bytes wide, then 2 bytes are fetched, even if only the first few bits are needed. For convenience, the number of bits fetched is called the "fetching unit."

Note that for the purpose of structure member alignment, the align switch value (1 byte, 2 bytes, or 4 bytes) is taken as a "virtual buswidth," even if it is different from the actual buswidth.

A complication exists when allocating bit-fields. The complication arises from the fact that different base types for bit-fields (`char` `short`, and `int`) are supported. The maximum length of a bit-field is the size of its base type; therefore, there may be times when a bit-field is larger than the buswidth. When the size of the base type is larger than the buswidth, the size of the fetching unit is considered to be the base-type size.

The precise rules for determining the start of the fetching unit are quite complicated. In general, it is determined by the current position in the allocation of structure members and by the base-type of the first bit-field in a group of consecutive bit-fields.

An attempt is made to pack consecutive bit-fields as much as possible, as long as the bit-fields remain in the same fetching unit. As soon as a field "spills over" into the next fetching unit, the alignment is set to the next memory unit (byte, word, or double-word, according to the align switch value and the base type of the field). A hole of padding bits remains, and the beginning of the spill-over field determines the start of a new fetching unit for following bit-fields. Using this method, bit-fields are packed as much as possible while still maintaining the alignment.

If, because of the bit-fields, the structure as a whole does not terminate on a byte boundary, padding bits are added to it to fill up to the end of the last byte it occupies. Additional padding bytes may be needed to fill to the alignment boundary of the largest structure member. This is seen in Figure 4-1. The bit-field does not quite reach the byte boundary; therefore, padding bits are added until the byte boundary is reached. Additional padding bytes are added to fill to the alignment boundary of the double-word structure member.

Example:

```
struct A {
  int i;
  unsigned bitfield : 4;
} a;
```

The arrangement of a's fields in memory will be:

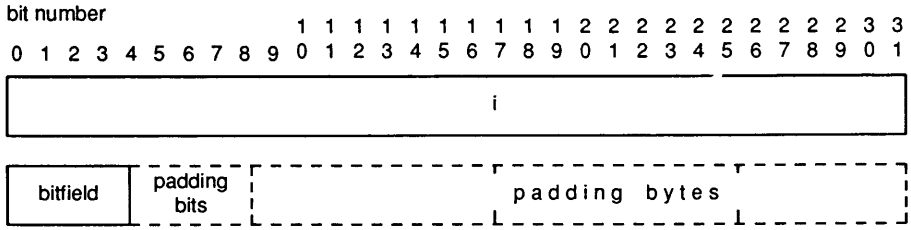


Figure 4-1. Bitfield Padding

Figure 4-2 is an example of the alignment on bit-fields given the different align switch settings. To summarize, the `-J` (`/ALIGN`) switch affects:

- the alignment and padding used for structure members and the alignment of variables of the structure type.
- the total storage allocated to a structure by determining if, and how many, padding bytes will be added after its last field.

Example:

```
struct X {
        char c,d,e;
        int  i: 24;
        };
```

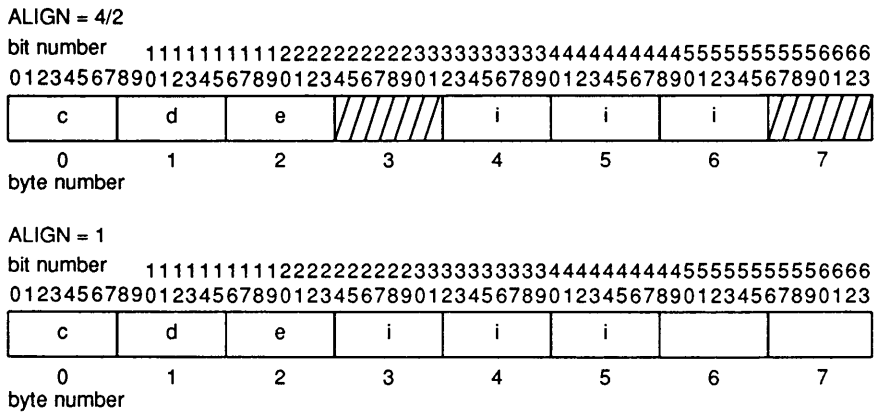


Figure 4-2. Alignment on Bitfields

CAUTION

The user must make sure that all parts of the program use the same alignment for the same structures; otherwise, problems result. The following example illustrates this point.

Suppose the example program includes `"foo.h"`. The file `"foo.h"` contains the following definitions:

```
typedef struct {
        int             counter;
        unsigned char   *pointer;
        char            flag1;
        char            flag2;
} XXX;

extern XXX array[10];
```

Note that `XXX` has two `char` members at its end. If align=4, any variable declared to be of type `XXX` will have two padding bytes added at its end in order to make it occupy an integral number of double-words. When align=1 or align=2, no padding is performed.

If a module using `"foo.h"` is compiled with align=4 and later linked with a module compiled with align=1 or align=2 that tries to use `array[n]` where `n > 0`, the result will be wrong. This is because the two modules disagree on the size of the elements in the array.

The solution to this problem is to make sure all modules are compiled using either the same alignment setting or a revised header file that has been made insensitive to the setting of the alignment switch. This is performed by including the necessary padding to enforce equal sizes and offsets. If the latter solution is chosen, `XXX` is revised to look like:

```
typedef struct {
        int             counter;
        unsigned char   *pointer;
        char            flag1;
        char            flag2;
        short           padding;
} XXX;
```

No padding is added by the compiler, and the size of the structure is the same for all switch settings.

4.2.5 Structure Returning Functions

In the GNX—Version 4 C compiler, structure returning functions have a hidden argument which is the address of an area the size of the returned structure. This area is allocated by the caller and its address is passed as a first argument to the structure returning function. Structure returning functions are, therefore, re-entrant and interruptible.

NOTE: At the optimizer's discretion, small structures (less than 5 bytes) may be passed and/or returned in a register.

4.2.6 Calling Sequence

The standard *Series 32000* calling conventions are used by the GNX—Version 4 C compiler for calls to external routines of all languages. It is, therefore, unnecessary to use the `fortran` keyword in C programs (if present, the keyword is ignored).

However, local or internal routines (functions which in C are preceded by the `static` keyword) are called by more efficient calling sequences.

The standard *Series 32000* calling conventions are described in Appendix A.

NOTE: Code using the *Series 32000* modularity features cannot be mixed with code not using those features. By default, the GNX—Version 4 tools assume no modularity.

4.2.7 Mixed-Language Programming

Mixed-language programs are frequently used for a couple of reasons. First, one language may be more convenient than another for certain tasks. Second, code sections already written in another language (*e.g.*, an already existing library function) can be reused simply by calling them.

A programmer who wishes to mix several programming languages needs to be aware of subtle differences between the compilation of the various languages. Appendix B describes the issues one needs to be aware of when writing mixed-language programs and compiling and linking such programs successfully.

4.2.8 Order of Evaluation

The evaluation order of expressions and actual parameters in the GNX—Version 4 C compiler differs from those of other compilers. Therefore, programs that rely on a specific order of evaluation may not run correctly when compiled. In particular, the following orders of evaluation are unspecified:

- The order in which expressions are evaluated.
- The order in which function arguments are evaluated.
- The order in which side effects take place. For instance, `a[i++] = i` may be evaluated as

```
a[i] = i;
i++;
```

or as

```
t = i;
i++;
a[t] = i;
```

4.2.9 Order of Allocation of Memory

The order of allocation of local variables in memory is compiler-dependent. After the optimizer of the GNX—Version 4 C compiler performs register allocation, it reorders the local variables left in memory. This reordering reduces memory space requirements and minimizes displacement length. User programs that rely on any order of allocation of local variables may not run correctly. See Chapter 6.

4.2.10 Register Variables

By default, register variables, as well as other local variables, are equal candidates for register allocation. When given complete freedom, the optimizer generally performs a better job of register allocation than when forced to follow the programmer's allocation suggestions. For programs which make assumptions about variables which reside in specific registers, an optimization flag (`-Ou` or `-O -Fu` on UNIX and `USER_REGISTERS` on VMS) is available to enforce the `pcc` allocation scheme for register variables of scalar types and of type double. See also Section 6.6.7.

4.2.11 Floating-Point Arithmetic

The floating-point arithmetic conversion rules of the GNX—Version 4 C compiler complies with the ANSI C standard and may differ from other C compilers.

In an operation not containing double operands, if one of two operands is of type `float`, the other operand is converted to type `float` and single-precision arithmetic is used. The result of the operation is of type `float`. Some other compilers perform such operations in double precision.

In old C compilers, the result of float-returning functions was actually returned in double format and placed in the F0-F1 register pair. When compiled by the GNX—Version 4 C compiler, such functions return the result in float format and place the result in the F0 register. Note that assembly programs that interface with float-returning functions may now incorrectly expect a double precision result.

Float parameters, however, are passed as double because the C language semantics do not require type identity between actual and formal parameters. Code is generated in the called function to convert these actual double values back to float if necessary.

Floating-point constants are of type `double`, unless they are typecast to `float` or are suffixed by the letter `f` or `F`. By preference, constants of type `float` should be used in float expressions to avoid the unnecessary casting of other operands to double precision. For example,

```
fmax += 17.5f;
```

is more efficient than

```
fmax += 17.5;
```

The following examples are of double constants and float constants.

Example:	*double constants*	*float constants*
	14.5e6	14.5e6f
	14.5	`(float)` 14.5

4.3 UNDEFINED BEHAVIOR

In the following cases, the behavior of the GNX—Version 4 C compiler is undefined:

- The value of a floating-point or integer constant is not representable.
- An arithmetic conversion produces a result that cannot be represented in the space provided.
- A volatile object is referred to by means of a pointer to a type without the volatile attribute.

- An arithmetic operation is invalid, such as division by 0, or produces a result that cannot be represented in the space provided, such as overflow or underflow.
- A member of a union object is accessed using a member of a different type.
- An object is assigned to an overlapping object.
- The value of a register variable has been changed between a `setjmp` call and a `longjmp` call.

Chapter 5

OPTIMIZATION TECHNIQUES

5.1 INTRODUCTION

The main difference between the GNX—Version 4 C Compiler and other compilers is the optimizer. Recompiling and optimizing with the GNX—Version 4 C Compiler will result in a 10 percent to 200 percent speedup for most programs, with the mean above 30 percent.

This chapter describes some of the advanced optimization techniques used by the GNX—Version 4 C Compiler to improve speed or save space. The most important techniques are:

- Value propagation
- Constant folding
- Redundant assignment elimination
- Partial redundancy elimination
- Common subexpression elimination
- Flow optimizations
- Loop unrolling
- Dead-code removal
- Loop-invariant code motion
- Strength reduction
- Induction variable elimination
- Register-allocation by coloring
- Peephole optimizations
- Memory-layout optimizations
- Fixed frame
- Runtime feedback optimization

The following sections describe these techniques in more detail. For coding suggestions and other practical guidelines on how to make best use of the optimizing aspects of the compiler, see Chapter 6.

5.2 THE OPTIMIZER

The optimizer, shared by all the GNX — Version 4 Compilers, is based on advanced optimization theory, developed over the past 15 years. Central to the optimizer is an innovative global-data-flow-analysis technique which simplifies the optimizer's implementation. It allows the optimizer to perform some unique optimizations in addition to standard optimizations found in other compilers. Optimizations are performed globally on the code of a whole procedure at a time and not just in a local context.

The optimizer can be regarded as a multi-step process. Each step performs its particular optimizations and provides new opportunities for the optimizations of the next step.

STEP ONE

The first step in the optimization process is to read in the source program one procedure at a time and to partition this procedure into basic blocks. A basic block is a straight line sequence of code with a branch only at the entry or exit. Some of the optimizations performed during this step are:

- **Value Propagation**
  Value propagation (or copy propagation) is the attempt to replace a variable with the most recent value that has been assigned to it. This optimization is primarily useful in the special case of constant propagation. It is important because it creates opportunities for other optimizations. Value propagation can be turned off by the `CODE_MOTION` optimization flag (`-Om` on UNIX systems).

- **Constant Folding**
  If an expression or condition consists of constants only, it is evaluated by the optimizer into one constant, thereby avoiding this computation at run-time. The optimizer, using algebraic properties such as the commutative, associative and distributive law, sometimes rearranges expressions to allow constant folding of part of an expression.

  The GNX—Version 4 C Compiler also folds floating-point constant expressions. This feature can be turned off using the `NOFLOAT_FOLD` option (`-Oc` on UNIX systems) of the optimizer.

- **Redundant Assignment Elimination**
  The optimizer detects and eliminates assignments to variables which are not used later in the program or which are assigned again before being used. This optimization can often be applied as a result of value propagation.

  Value propagation, constant folding, and redundant assignment elimination are illustrated in Figure 5-1.

The program sequence

```
a = 4;
if (a*8 < 0) b = 15;
else b = 20;
```

... *code which uses* b *but not* a ...

is translated by the GNX—Version 4 C Compiler front end into the following intermediate code

```
    a ← 4
    if (a*8 >= 0) goto L1
    b ← 15
    goto L2
L1: b ← 20
L2: ...
```

which is transformed by "value propagation" into

```
    a ← 4
    if (4*8 >= 0) goto L1
    b ← 15
    goto L2
L1: b ← 20
L2: ...
```

which after "constant folding" becomes

```
    a ← 4
    if (true) goto L1
    b ← 15
    goto L2
L1: b ← 20
L2: ...
```

"dead code removal" results in

```
    a ← 4
    goto L1
L1: b ← 20
L2: ...
```

which is transformed by another "flow optimization" into

```
    a ← 4
    b ← 20
    ...
```

Since there is no further use of a, a ← 4 is a "redundant assignment:"

```
    b ← 20
    ...
```

Figure 5-1. Relationship Between Various Optimizations

STEP TWO

The second step in the optimization process is the construction of the program's "flow graph." This is a graph in which each node represents a basic block. As mentioned in STEP ONE, a basic block is a linear segment of code with only one entry point and one exit point. If there is a path in the program that leads from one basic block to another, then an "arrow" is drawn in the graph to represent this path.

Figure 5-2 illustrates a flow graph, representing an "if-then-else" sequence.

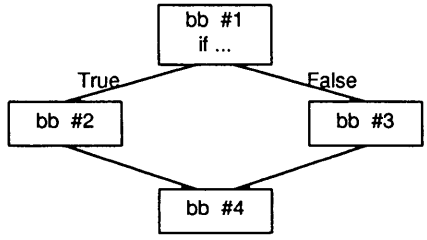


Figure 5-2. Flow Graph

During the construction of the flow graph, additional optimizations can be performed:

- **Flow Optimizations**
  Flow optimizations reduce the number of branches performed in the program. One example is to replace a branch whose target is another branch with a direct branch to the ultimate target. This often makes the second branch redundant. At other times code is reordered to eliminate unnecessary branches. Branches to "return" are replaced by the return-sequence itself.

- **Loop Unrolling**
  Loop unrolling duplicates the body of a loop. This reduces the number of times that the loop control code is executed. Loop unrolling improves performance by reducing the number of increment, comparison, and branch instructions.

  This technique is particularly useful for small loops whose iteration control constitutes a significant part of loop execution time. However, because loop unrolling does involve the duplication of the loop's body, more space is needed.

  When a loop is unrolled, it is replaced by code with the following structure:

  1. Pre-Loop Code - Checks whether to enter the unrolled loop body or to branch to the tail-code.
  2. Unrolled Loop Body - The loop body duplicated a number of times.
  3. Tail-Code - Performs the remaining iterations.

  Based on a loop's code size, the optimizer determines whether to perform loop unrolling, and if so how many times. An example of loop unrolling is shown in Figure 5-3.

Code sequence for initialization of an array :

```
foo(int j)
{
  int i;
  int a[100];

  for (i=j; i < 100; i++) {
       a[i] = 0;
    }
}
```

When unrolled 5 times, is equivalent to the following code sequence:

```
foo(int j)
{
  int i;
  int a[100];

  /* pre-loop code */
  i = j;
  if (i > 95) goto TAIL_CODE;

  /* unrolled loop body */
  for (; i < 95; i += 5) {
       a[i] = 0;
       a[i+1] = 0;
       a[i+2] = 0;
       a[i+3] = 0;
       a[i+4] = 0;
    }

  TAIL_CODE:
  /* tail code */
  for (; i < 100; i++)
       a[i] = 0;
}
```

Figure 5-3. Example of Loop Unrolling

- **Dead Code Removal**
  Flow optimizations are also designed to help the optimizer discover code which will never actually be executed. Removal of this code, called "dead code removal," results in smaller object programs.

STEP THREE

Step three of the optimization process is called "global-data-flow-analysis." It identifies

desirable global code transformations which speed program execution. Many of these concentrate on speeding up loop execution, since most programs spend 90 percent or more of their time in loops. Global-data-flow-analysis is the computation of a large number of properties for each expression in the procedure.

Unlike most optimizers, which employ unrelated and separate techniques, the optimizer centers around one innovative technique which involves the recognition of a situation called "partial redundancy." This technique is so powerful that many other optimizations turn out to be special cases. The central idea is that it is wasteful to compute an expression, say `a*b`, twice on the same path; it is often faster to save the result of the first computation and then replace the fully redundant second computation with the saved value. More common, however, is the case in which an expression is partially redundant; there is one path to an expression, which already contains a computation of that expression, but another path to that same expression does not.

The following optimizations are performed by a common technique:

- **Elimination of Fully Redundant Expressions**
  This optimization is often called "Common Subexpression Elimination." It is relatively simple to avoid the recomputation of fully redundant expressions. The optimizer saves the result of the first computation (usually in a register variable) and uses the saved value in place of the second computation. Performance-conscious programmers sometimes do this themselves, but many cases, such as array index and structure member calculations, are recognized only by the optimizer.

- **Partial Redundancy Elimination**
  A partially redundant expression can be eliminated in two steps. First, insert the expression on the paths in which it previously did not occur; this makes the expression fully redundant. Second, save the first computations and use the save value to replace the redundant computation. An example of this optimization is shown in Figure 5-4.

  Partial redundancy elimination sometimes results in slightly larger code, but execution is not harmed, since all inserted expressions are in parallel and only one is actually executed.

- **Loop Invariant Code Motion**
  If an expression occurs within a loop and its value does not change throughout that loop, it is called "loop invariant." Loop invariant expressions are also partially redundant. This can be understood by realizing that there are two paths into the loop body: one is through the loop entry (the first time the loop is executed), and the other is from the end of the loop, while the exit condition is false. Loop invariant computations are, therefore, removed from the loop in the same way: the expression is first inserted on the entry path to the loop, and then is saved on the entry path in a register, while the redundant computation in the loop is replaced by that register.

- **Strength Reduction**
  This optimization globally replaces complex operations with simpler ones. This is

In the following code, a*b is "partially redundant" (computed twice only if C is true):

```
if (C)
    x = a*b;
else
    b = b+10;
y = a*b;
```

It is first transformed into a "fully redundant" expression

```
if C = 1
    x ← a*b
else
    b ← b+10
    temp ← a*b
y ← a*b
```

Then, as in the simple case of "redundant expression elimination," this is reduced to

```
if C = 1
    temp ← a*b
    x ← temp
else
    b ← b+10
    temp ← a*b
y ← temp
```

Now, the expression a*b is computed only once on any path.

Figure 5-4. Example of Partial Redundancy Elimination

primarily useful for reducing complex array-subscript computations (involving multiplication into simpler additions).

Example:

```
static int a[15]; for (i=0; i<15; i+=1)
    a[i] = 1;
```

is transformed into:

```
for (i=0, p=a; i<15; i+=1, p+=4)
    *p = 1;
```

- **Induction Variable Elimination**
  Induction variables are variables which maintain a fixed relation to other variables. The use of such variables can often be replaced by a simple transformation. For instance, the example given for strength reduction can be reduced to the following:

```
for (p=a; p<a+60; p+=4)
   *p = 1;
```

STEP FOUR

The fourth optimization step performed by the optimizer, and possibly the most profitable, is the "register allocation" phase. Register allocation places variables in machine registers instead of main memory. References to a register are always much faster and use less code space than respective memory references.

The algorithm used by the optimizer is called the "coloring algorithm." First, global-flow-analysis is performed to determine the different live ranges of variables within the procedure. A live range is the program path along which a variable has a particular value. Generally, an assignment to a variable starts a new live range; this live range terminates with the last use of that assigned value.

The optimizer subsequently constructs a graph as follows: each node represents a live range; two nodes are connected if there exists a point in the program in which the two live ranges intersect. The allocation of registers to live ranges is now the same as coloring the nodes of the graph so that two connected nodes have different colors. This is a classic problem from graph theory, for which good solutions exist. If there are not enough registers, more frequently used variables have higher priority than less frequently used ones. Loop nesting is taken into account when calculating the frequency of use, meaning that variables used inside of loops have higher priority than those that are not.

Most optimizing compilers attempt register allocation only for true local variables, for which there is no danger of "aliasing." An alias occurs when there are two different ways to access a variable. This can happen when a global variable is passed as a reference parameter; the variable can be accessed through its global name, or through the parameter alias. A common case in C is when the address of a variable is assigned to a pointer.

The optimizer takes a more general approach by considering all variables with appropriate data types as candidates for register allocation, including global variables, variables whose addresses have been taken, array elements, and items pointed to by pointers. These special candidates cannot reside in registers across procedure calls and pointer references and, therefore, normally have lower priority than local variables. However, instead of completely disqualifying the special candidates in advance, the decision is made by the coloring algorithm.

Additional important optimizations performed by the register allocator are:

- **Use of Safe and Scratch Registers**
  The *Series 32000* machine registers are, by convention, divided into two groups: registers R0 through R2 and F0 through F3, the so-called "scratch" registers which can be used as temporaries but whose values may be changed by a procedure call, and the "safe" registers (R3 through R7 and F4 through F7) which are

guaranteed to retain their value across procedure calls. The register allocator spends a special effort to maximize the use of scratch registers, since it is not necessary to save these upon entry or restore them upon exit from the current procedure. The use of scratch registers, therefore, reduces the overhead of procedure calls.

- **Register Parameter Allocation**
  The register allocator attempts to detect routines, whose parameters can be passed in registers. This is possible for static routines only, since by definition all the calls to such routines are visible to the optimizer. Calls to other (externally callable) routines are subject to the standard *Series 32000* calling sequence. Passing parameters in registers is another way to reduce the overhead of procedure calls.

STEP FIVE
The last optimization step consolidates the results of all previous steps by writing out the optimized procedure in intermediate form for the separate code generator. Some reorganization takes place during this step. Local variables which have been allocated in registers are removed from the procedure's activation record (frame), which is reordered to minimize overall frame size.

5.3 THE CODE GENERATOR

The back end (code generator) attempts to match expression trees with optimal code sequences. It applies standard techniques to minimize the use of temporary registers, which are necessary for the computation of the subexpressions of a tree. The main strength of the code generator lies in the number of "peephole optimizations" it performs.

Peephole optimizations are machine-dependent code transformations that are performed by the code generator on small sequences of machine code just before emitting the code. Some of the most important peephole transformations are listed below:

- The code for maintaining the frame of routines which have no local variables, or whose variables are all allocated in registers, is removed.
- Case statements are optimized into binary search, linear search or table-indexed code (using the *Series 32000* `CASE` instruction), in order to obtain optimal code in each situation.
- The stack and frame areas are always aligned for minimal data fetches.
- Reduction of arithmetic identities, *i.e.*, `x*1 = x, x+0 = x`, etc.
- Use of the `ADDR` instruction instead of `ADD` of three operands.
- Some optimizations performed in the optimizer, such as the application of the distributive law of algebra, *i.e.*, `(10+i)*4 = 40+4*i`, provide additional opportunities to the code generator to fully exploit the *Series 32000*'s addressing modes.

- Use of ADDR instead of MOVZBD of small constant.
- Strength Reduction Optimizations. Use of MOVD instead of MOVF from memory to memory; use of index addressing mode instead of multiplication by 2, 4 or 8; use of combinations of ADDR instructions or shift and ADD sequences instead of multiplication by other constants up to 200.
- Fixed Frame Optimization. An important contribution of the code generator is its ability to precompute the stack requirements of a procedure in advance. This allows the generation of code which does not use (nor update) the FP (frame pointer), resulting in cheaper calling sequences.

  This optimization is most useful when the procedure contains many procedure calls because it is not necessary to execute code to adjust the stack after every call. Parameters are moved to the pre-allocated space instead of pushing them on to the stack using the top-of-stack addressing mode. Note that when using this optimization, the run-time stack pointer stays the same throughout the procedure, and all references to local variables are relative to it and not to the FP. Also note that since parameters are not pushed on to the stack, the evaluation order of parameters is not defined solely by their original order.

While most optimizations are beneficial for both speed and space, some optimizations favor one over the other. The default setting of the optimizer switch favors speed over space in trade-off situations. The following are the effects of favoring space over speed (by an optimization flag):

- Code is not aligned after branches.
- All returns within the code are replaced by a jump to a common return sequence.
- Certain space-expensive peephole transformations are not performed.

5.4 MEMORY LAYOUT OPTIMIZATIONS

The following memory layout optimizations are performed by the GNX—Version 4 C Compiler:

- Frame variables that are allocated in registers are removed from the frame.
- Internal, static routines whose parameters are passed in registers have smaller frames.
- The stack alignment is always maintained. Stack parameters are passed in aligned positions.
- Frame variables are allocated in aligned positions. The optimizer reorders these variables to save overall frame space.
- Code is aligned after every unconditional jump.

5.5 RUNTIME FEEDBACK

The optimizer has normally no way to determine the actual runtime behavior of a program. What looks like a loop may in reality never be executed. The GNX—Version 4 C Compiler has an option to create a statistic record of a program's execution path. This execution profile can then be used in a subsequent optimization pass of the same program, to improve the optimizer's heuristic algorithms. This technique, call runtime feedback optimization, effects mainly the following optimizations:

- Loop Unrolling
- Register Allocation

For more details on runtime feedback optimization see Section 7.4.

Chapter 6

GUIDELINES ON USING THE OPTIMIZER

6.1 INTRODUCTION

The following sections are provided as guidelines on using the GNX—Version 4 C Compiler. Experienced programmers should understand this compiler's optimization techniques in order to:

- Learn how to port programs to the GNX—Version 4 C Compiler.
- Understand how to recognize and avoid nonportable code.
- Avoid using programming tricks that rely on the way ordinary compilers generate code.
- Avoid performing "hand optimizations" that the optimizer does anyway.
- Avoid writing code that may prevent certain optimizations.
- Understand how to select the different command line optimization flags to achieve optimal performance.

Please read Chapter 5 for a complete description of the optimization techniques.

6.2 OPTIMIZATION FLAGS

Optimization options available to the user are listed in Table 6-1. Default options are marked by (*).

Table 6-1. Optimization Options

UNIX	VMS	DESCRIPTION
o	NOOPT	does not invoke the optimizer phase.
B	RUNTIME_FEEDBACK	performs runtime feedback optimization
* b	NORUNTIME_FEEDBACK	does not perform runtime feedback optimization
c	NOFLOAT_FOLD	does not compute floating-point constant expressions at compile time.
* C	FLOAT_FOLD	performs floating-point constant folding.
* F	FIXED_FRAME	uses fixed frame references, avoids use of the FP register or the *Series 32000* ENTER/EXIT instruction.
f	NOFIXED_FRAME	compiles for debugging: uses slower FP and TOS addressing modes.
I	NOVOLATILE	applies all optimizations to all variables (including global variables).
i	VOLATILE	compiles system code: assumes that all global and static memory variables and pointer dereferences are volatile.
L	STANDARD_LIBRARIES	assumes use of standard run-time library
l	NO STANDARD_LIBRARIES	assumes that all routines have corrupting side effects.
* M	CODE_MOTION	performs global code motion optimizations.
m	NOCODE_MOTION	does not perform global code motion optimizations.
N	LOOP_UNROLLING	performs loop-unrolling optimizations.
* n	NOLOOP_UNROLLING	does not perform loop-unrolling optimizations.
U	NOUSER_REGISTERS	ignores user register declarations.
u	USER_REGISTERS	allocates user-declared register variables in registers as done by `pcc`.
* R	REGISTER_ALLOCATION	performs the register allocation pass of the optimizer.
r	NOREGISTER_ALLOCATION	does not perform the register allocation pass of the optimizer.
* S	SPEED_OVER_SPACE	optimizes for speed only.
s	NOSPEED_OVER_SPACE	does not waste space in favor of speed.
1-9		maximal memory/swap-space available is 1 through 9 Mbytes (default: 4 Mbytes).

6.2.1 Optimization Options on the Command Line — UNIX Systems

The `-O` option enables the optimizer. Specifying `-O` on the command line results in the fastest possible code without undue increase in code size. (`-ObCFIUMnLRS`). In special cases, such as when compiling operating system code, there may be a need to further refine the optimization phase by specifying optimization flags. Individual optimization flags can be specified either by using the `-F` option or by simply appending them to `-O`. Table 6-2 lists reasons why a particular default option might be changed.

Even when the optimizer pass is omitted, some local optimizations are performed by the code generator. Note that specifying the compiler debug option (`-g`) on the command line automatically turns off the optimizer fixed frame flag (`-OF`), unless otherwise specified by the user.

Also note that using the compiler target option (`-KB1`) favors space over speed by saving alignment holes normally produced when the bus width is the default (`-KB4`).

6.2.2 Optimization Options on the Command Line — VMS Systems

The fastest possible code, without undue increase in code size, is generated by specifying `/OPTIMIZE` on the command line. This is equivalent to entering:

```
/OPTIMIZE=(FIXED_FRAME, CODE_MOTION, REGISTER_ALLOCATION, FLOAT_FOLD,
SPEED_OVER_SPACE, NOVOLATILE, STANDARD_LIBRARIES, NOUSER_REGISTERS,
NOLOOP_UNROLLING,NORUNTIME_FEEDBACK)
```

In special cases, such as when compiling operating system code, there may be a need to further refine the optimization phase by specifying optimization flags. Table 6-2 lists reasons why a particular default option might be changed.

Even when the optimizer pass is omitted, some local optimizations are performed by the code generator. Therefore, specifying `/NOOPTIMIZE` (which is the default for this qualifier) is equivalent to entering:

```
/OPTIMIZE=(   NOOPT, NOFIXED_FRAME, NOCODE_MOTION, NOREGISTER_ALLOCATION,
              NOFLOAT_FOLD, SPEED_OVER_SPACE, NOVOLATILE,
              NOSTANDARD_LIBRARIES, USER_REGISTERS,NOLOOP_UNROLLING,
              NORUNTIME_FEEDBACK)
```

Note that specifying the compiler debug option (`/DEBUG`) on the command line automatically turns off the optimizer fixed frame option (`FIXED_FRAME`), unless otherwise specified by the user.

Also note that using the compiler option `/TARGET=(BUSWIDTH=1)` favors space over speed by saving alignment holes normally produced when the bus width is the default (`BUSWIDTH=4`).

6.2.3 Changing Default Optimization Options

There is normally no reason to turn off any of the optimization options; the default produces the best results, see Table 6-2. Refer to Chapters 2 and 5 for more on optimization options.

Table 6-2. Changing Default Optimization Options

OPTION	REASON FOR CHANGING OPTION	SEE ALSO
NOFIXED_FRAME (-Of)	to debug the program or to compile nonportable programs that assume knowledge of the run-time stack.	6.3.4, 6.4
VOLATILE (-Oi)	to compile system programs, such as device drivers, which contain variables that change or are referenced spontaneously.	6.3.2
NO_STANDARD_LIBRARIES (-Ol)	to compile programs which re-implement standard functions, in a way which does not agree with the optimizers assumptions (*i.e.*, have side effects).	6.3.5
NOFLOAT_FOLD (-Oc)	to compile programs whose correct execution depends on the order in which floating-point expressions are evaluated.	6.3.6
NOCODE_MOTION (-Om)	to compile programs which contain huge functions, which are a drain on the system's resources and are time consuming to optimize.	
LOOP_UNROLLING (-ON)	to compile program segments containing tight loops which are most often executed.	6.6.9
USER_REGISTERS (-Ou)	to compile programs which rely on the register allocation scheme of pcc.	6.6.7
NOREGISTER_ALLOCATION (-Or)	to run programs that cease to work when performing register allocation.	6.6.7
NOSPEED_OVER_SPACE (-Os)	to compile programs which must fit as tightly as possible in memory.	6.6.9
NOOPT (-Oo or use -F*flags* without giving -O)	when the optimizer phase is not required and another flag needs to be turned off as well.	6.6.10
NORUNTIME_FEEDBACK (-Ob or use -F*flags* without giving -O)	when run-time feedback is required to achieve better optimization results based on the typical behavior of the program.	6.6.11

6.3 PORTING EXISTING C PROGRAMS

Almost every program which runs when compiled by other C Compilers, will compile and run on the GNX—Version 4 C Compiler without any changes in the sources. However, there might be a few programs which will cease to work in the same manner as before, when compiled by the GNX—Version 4 C Compiler. There might be other programs, which seem to work when compiled without the optimizer, but which cease to work when optimized. The following sections describe some of the reasons for this phenomenon.

6.3.1 Undetected Program Errors

The single most common reason for a nonfunctioning program is an undetected program error, which becomes apparent only when compiling under a different compiler or only when optimizing. Many of these errors result from the fact that the program author relied on the way the compiler compiled, and thereby created a program which is clearly nonportable.

The following partial list points out some of the most common problems:

- **Uninitialized local variables.**
  Since the memory and register allocation algorithms of the GNX—Version 4 C Compiler are very different from those of other compilers, a local variable may wind up in a completely different place. For example, a programmer may fail to initialize a local variable, with the assumption that, upon program start, it would certainly contain zero. This may become false as a result of the register allocation phase of the GNX—Version 4 C Compiler.

- **Relying on memory allocation**
  One cannot assume that if two variables are declared in a certain order, they will actually be allocated in that order. A program that uses address calculations to proceed from one declared variable to another declared variable might not work.

- **Failing to declare a function**
  A `char` returning function will return a value in the lower-order byte of R0, without affecting the other bytes. A failure to declare that function where it is used, might result in an error. For instance, assuming that `get_code()` is defined to return a `char`, then

```
main() {
   int   i;
   if ((i = get_code()) == 17)   do_something();
}
```

  might never execute `do_something` even if `get_code` returns `17` since the whole register is compared to `17`, not just the low-order byte.

  A similar problem exists for functions which return `short` or `float`, or those which return a structure.

6.3.2 Compiling System Code

System code is distinguished from general "high-level" code, by the fact that it is machine-dependent, often contains real-time aspects and interspersed `asm` statements, and is often driven by asynchronous events, such as interrupts. Examples of system code are interrupt routines, device handlers and kernel code. From the optimizer's point of view, ordinary looking global variables can actually be semaphores or memory-mapped I/O, that can be affected by external events, which are not under the optimizer's control. Even so, it is still possible to optimize such code, by taking some precaution, and by activating some special optimization flags. Some of these aspects are discussed in the following sections.

- **Volatile variables**
  Volatile variables are variables, which might be used or changed by asynchronous events, such as I/O or interrupts. The `/OPTIMIZE=VOLATILE` (`-Oi` under the UNIX operating system) qualifier treats all global variables, static variables, and pointer dereferences as volatile, which means that they are not subject to any optimizations. As a result, the number and nature of memory references to them will not change. Remember that individual identifiers can be declared as volatile by using volatile type qualifiers. The following examples demonstrate the consequences of volatile variables and pointer dereferences.

  Examples: 1. `x = 17; x = 18;`

  If `x` is volatile, both of the two assignments to `x` are executed even though the first one seems redundant.

  2. `x = 9;`
     `y = x + 1;`

     If `x` is volatile, this program segment is not optimized to
     `y = 10;`

  3. `*p = b + c;`

     if `*p` is volatile, then this results in

```
movd    b, REG
addd    c, REG
movd    REG, 0(p)
```

     and not

```
movd    b, 0(p)
addd    c, 0(p)
```

  The difference stems from the fact that the second sequence, though faster, makes two references to `0(p)` when the programmer may have wanted only one.

6.3.3 Timing Assumptions

Optimizing a program changes the timing of various constructs. In particular, delay-loops might now run faster than before.

6.3.4 Low-Level Interface

- **Relying on register order**
  A program that relies on the fact that a given register variable resides in a specific register must be compiled with the `/OPTIMIZE=USER_REGISTERS` flag (`-Ou` on UNIX systems) turned on (see Section 6.6.7).

- **Relying on frame structure**
  A program, that relies on a specific frame structure, must be compiled with the `FIXED_FRAME` flag turned off (`-Of` on UNIX systems). This includes, in particular, programs that use the standard `alloca()` function (which allocates space on the user's frame).

  Referring to variables on the frame of a different function (such as the caller of this function) by complex pointer arithmetic may also cease to work. See Appendix A for more details.

- **Using `asm` statements**
  The code inserted by `asm` statements may cease to work because the surrounding code produced by the GNX—Version 4 C Compiler will normally be different from another compiler's code. See Section 6.6.6.

6.3.5 Using Nonstandard Library Routines

The GNX—Version 4 C Compiler assumes by default that all the C standard mathematical library routines listed in Table 6-3 are available as a standard run-time library. These library routines have absolutely no access to global variables. Therefore, calls to these routines are specially recognized and marked as calls which do not disturb optimizations of the global variables of the program. The global library variable `errno` is treated as volatile, so no references to it will be optimized. This is normally a safe assumption since it is unusual for a program to redefine (and thereby hide) these standard routines. The functions `abs`, `fabs`, and `ffabs` actually compile into in-line code and do not generate a procedure call at all.

In addition, a set of intrinsic routines known internally to the computer are supported. See Chapter 8.

The compiler generates a warning message whenever it compiles a program which does redefine one of these routines. In this case the user must decide whether the redefined routine's behavior is consistent with the previously mentioned assumption of the optimizer. If it is not, the user has the choice of renaming the redefined routine (so that calls to it are not specially recognized), or of using the `NOSTANDARD_LIBRARY` flag (`-Ol` on UNIX), which turns off the recognition of all library routines.

Table 6-3. Recognized Library Routines

abs	acos	facos	asin	fasin	atan	fatan
atan2	fatan2	cabs	fcabs	ceil	fceil	cos
fcos	cosh	fcosh	erf	ferf	erfc	ferfc
exp	fexp	fabs	ffabs	fmod		
ffmod	fmodf	ffmodf	frexp	gamma	hypot	fhypot
j0	j1	jn	ldexp	log	flog	log10
flog10	modf	pow	fpow	sin	fsin	sinh
fsinh	sqrt	fsqrt	tan	ftan	tanh	ftanh
y0	y1	yn				

6.3.6 Reliance on Naive Algebraic Relations

Since the optimizer performs floating-point constant folding, *i.e.*, it rearranges expressions to evaluate constant subexpressions at compile time, some naive algebraic expressions are folded away.

Example:

```
do {
        a = a*2;
}
while ((a + 1.0) - 1.0 == a);
```

is optimized to

```
do {
        a = a*2;
}
while (1);
```

which was not the programmer's intention.

To maintain the program and keep the programmer's original intention, the programmer should use the `NOFLOAT_FOLD` (`-Oc` on UNIX systems) optimization flag to suppress the folding optimization.

6.4 DEBUGGING OF OPTIMIZED CODE

Most of the time, the user should not need to debug an optimized program. The majority of all bugs can be found before optimization is turned on. However, there are some rare bugs which make their appearance only when the optimizer is introduced, bugs that are difficult to find without a debugger.

The problem is that code motion optimizations and register allocation obsolete most of the symbolic debugging information generated by the compiler. The following "rules of thumb" can be employed when using symbolic debug information together with the optimizer:

- Line number information is correct, but the code performed at the specified lines may be different from non-optimized code as a result of various code motion optimizations, such as moving loop invariant expressions out of loops.
- Symbolic information for global variables is normally correct, since global variables are rarely put in registers. In particular, if a global variable is not referenced within the current procedure, the value in memory is valid and the symbolic information is correct.
- Symbolic information for parameters is correct except in the following two cases:
  1. When a parameter is allocated a register and there is an assignment to that parameter, the symbolic information is incorrect.
  2. When a parameter of a local procedure is passed in a register as a result of an optimization, the symbolic information is incorrect. In this case, the symbolic information of all other parameters is incorrect because their offset within the procedure's frame is changed.
- Symbolic information of local variables is likely to be incorrect because most of the local variables are put in registers; the rest of the local variables are reordered into new frame locations, or "optimized out".
- Note that if symbolic information is requested, then slightly different code is generated. This happens because the optimizing flag `FIXED_FRAME` (`-OF` on UNIX systems) is automatically disabled when the `/DEBUG` (`-g` qualifier on UNIX systems) is used. Specifically, the `ENTER` instruction is always generated at the entry of procedures, and frame variables are referenced by FP-relative rather than SP relative addressing mode. Without disabling this flag, symbolic debugging is almost impossible.

It is helpful to have an assembly listing of the program in question which has been compiled with the `/ASM` (`-S` on UNIX systems) and the `/ANNOTATE` (`-n` on UNIX systems) qualifiers. Such a listing contains comments from the optimizer regarding its actions (see Section 6.5).

6.5 IMPROVED ANNOTATION

The GNX C compiler has a unique annotation feature which helps in the debugging of an optimized code.

Upon invocation with the `-n` and `-S` flag (`/ANNOTATE` and `/ASM` on VMS), the compiler emits the source lines into the assembly code as comments (see Section 2.3). In addition the GNX optimizer emits annotated comments explaining its actions.

Example: The following code accumulates the first `n` elements of array `a` into the global accumulator `acc`. `n` resides in register r4.

```
for(i = 0; i < n; i++)
    acc += a[i];
```

The optimizer may generate the following annotated code

```
#--- for(i = 0; i < n; i++)
#   # temp initialized to &a
        movqd       $(1),r2
        movd        $(0)+_a,r1
#   # load (moved up) acc to r3
        movd        _acc,r3
.LL2:
#--- acc += a[i];
        addd        0(r1),r3
        addqd       $(1),r2
#   # temp = temp + 4 (temp incremented)
        addqd       $(4),r1
        cmpd        r2,r4
        blt         .LL2
#   # store (moved down) r3 to acc
        movd        r3,_acc
```

The actions taken by the optimizer can be inferred from the comments:

"`load (moved up) acc to r3`" (the value of the variable `acc` is first loaded into the register r3)

"`temp initialized to &a (temp initialized)`" (while performing strength reduction optimization, a compiler

pointer is initialized by the first element's address)

`"temp = temp + 4 (temp incremented)"` (the temporary pointer allocated by the optimizer is updated to point to the next element at the end of each iteration)

`"store (moved down) r3 to acc"` (the value of `acc` is updated)

6.6 ADDITIONAL GUIDELINES FOR IMPROVING CODE QUALITY

Using some of the following programming guidelines results in programs that take advantage of the GNX—Version 4 C Compiler optimizations.

6.6.1 Static Functions

It is not only good software engineering practice, but also good optimization practice to declare all functions not called from outside the file as "static." This allows the optimizer to use a more efficient internal calling sequence upon calls to such routines. This internal calling sequence uses the BSR instruction instead of the JSR or CXP instruction and also passes parameters in registers rather than on the stack.

If a program consists of a single file and this is discovered by the GNX—Version 4 C Compiler (by indicating compilation and linking in one step), then all functions within that file are automatically considered static by the compiler, resulting in the same advantages.

6.6.2 Integer Variables

Many operators, including index calculations, are defined in C to operate on integers, and imply a conversion when given non-integer operands. Therefore, to avoid frequent run-time conversions from `char` or `short` to `int`, integer variables, particularly variables which serve as array-indices, should be defined as type `int` and not `short` or `char`.

6.6.3 Local Variables

Local variables should be used as much as possible, particularly when they are employed as loop counters or array indices, as they have a better chance of being placed in registers.

6.6.4 Floating-Point Computations

In programs which do not require double-precision floating-point computations, a significant run-time improvement can be achieved by paying attention to the following points:

- define all functions as returning float type, not double
- define all constants to be 'float' using the `f` suffix or cast expressions explicitly to float
- use the single precision version of the standard floating-point routines such as `fabs()` instead of `abs()`, `fsin()` instead of `sin()`, etc.

6.6.5 Pointer Usage

The following terms are used throughout this section.

- **potential definition**
  A statement potentially defines a memory location if the execution of the statement may change the contents of that memory location.

Example:

1. A call to a function potentially defines all global variables, because their value may change during the execution of that function.
2. Imagine the following code fragment.

```
extern int *p, *q;
        .
        .
*p = 8;
        .
        .
```

   The assignment statement potentially defines the memory location `*q`, because `q` may point to the same memory location as `p`. The location `*p` is defined, *i.e.*, given a new value, by the assignment. About location `*q`, it can only be said that it *may* be changed, hence the potential definition.

- **potential use**
  A statement "potentially uses" a memory location if it *may* reference (read from) that memory location.
- **address taken variable**
  A variable is considered "address taken" if the address operator (&) is applied to it within the file or if the variable is a global variable that is visible by other files.
- **volatile/non-volatile registers**
  By convention, the registers are divided into volatile registers (registers R0 through

R2 and F0 through F3) and non-volatile registers (registers R3 through R7 and F4 through F7). Volatile registers may be changed by a procedure call, whereas non-volatile registers are guaranteed to retain their value across procedure calls. Therefore, all non-volatile registers used within a procedure have to be saved at the entry and restored at the exit of that procedure.

The optimizer does not keep track of the contents of pointers; therefore, it cannot tell, for any given location in the program, where each pointer is pointing.

Since a pointer can point at any memory location, the optimizer makes the following assumptions concerning pointer usage:

1. Every assignment to a "pointer dereference," the location pointed to by a pointer, potentially defines all other pointer dereferences and all address-taken variables.
2. Every use of a pointer dereference (*i.e*, a value read through a pointer) potentially uses all other pointer dereferences and all address-taken variables. This is because any accessible memory location is potentially read.
3. Every function call potentially defines and potentially uses all pointer dereferences, all address taken-variables, and all global variables. This is so because that function's code may, using pointers, read and/or write any accessible memory location. Of course, any global variable may be used and/or changed.

It is advisable to keep these assumptions in mind when using pointers. In particular, using arrays is preferable to using pointers. The following example illustrates this point. Assume `a` is an array of `char` and `p` is a pointer to `char`. The two program segments perform the same function.

Example: *program segment 1*

```
for (i = 0 ; i != 10 ; i++) {
   a[i] = global_var; a[i+1] = global_var + 1;
}
```

program segment 2

```
for (p = &a[0] ; p != &a[10] ; p++) {
   *p = global_var; *(p+1) = global_var + 1;
}
```

In program segment 1, `global_var` can be put in a register. In program segment 2, however, `p` may point to `global_var`. The first statement (`*p = global_var`) potentially defines `global_var`; therefore, it cannot be put in a register.

Another aspect of this same issue is that of *common subexpressions*. The optimizer normally recognizes multiple uses of the same expression and saves that expression in a temporary variable (usually a register). This cannot be performed when worst case

assumptions are made about potential definition of expressions (as described in the previous section). Expressions that contain pointer dereferences or global variables are vulnerable; therefore, if many uses of the same expression span across procedure calls, it is advisable to save them in local variables. In the following example:

```
foo1(p -> x);
foo2(p -> x);
```

The expression `p -> x` cannot be recognized by the optimizer as a common subexpression because `foo1()` may change its value. The following hand optimization may help:

```
t = p -> x; /* t is local, therefore */
foo1(t);    /* not potentially defined by foo1() */
foo2(t);    /* so its value is still valid for foo2()
*/
```

The programmer is using his knowledge that `p -> x` is not changed by `foo1()` to make this optimization. The optimizer cannot do the same because it assumes the worst case.

6.6.6 Asm Statements

Extreme care should be taken if using `asm` statements. If using `asm` statements, remember the following:

- The optimizer is not aware of the contents of an `asm` statement. Therefore, it assumes that an `asm` statement potentially defines and potentially uses all of the variables (including local variables). This means that no common subexpressions can be recognized across an `asm` statement.
- In order to allow an `asm` statement to use a specific register (*e.g.*, `asm ("save [r0,r1,r2]");`), the optimizer de-allocates all the registers.
- The compiler usually generates code which is different from the code generated by other compilers. This applies particularly to allocation of local variables and parameters of static procedures.
- The code surrounding the `asm` statement may change as a result of changes in other parts of the procedure.
- An `asm` statement that contains a branch instruction or a branch target (label) may cause the optimizer to generate wrong code.

For the above mentioned reasons, it is strongly advised to look at the generated assembly code before and after inserting `asm` statements into a program.

6.6.7 Register Allocation

The C language is unique in that it allows the programmer to specify (or rather recommend) that some variables be allocated to machine registers. The optimizer normally ignores these recommendations, since it turns out that in most cases the optimizer's own register allocation algorithms are as good as or superior to the programmer's recommendations. There are several reasons for this fact:

- The user can use a register for one variable only. The optimizer however allocates a register along live ranges of variables, making it possible for several variables with non-conflicting live ranges to use the same register.
- The user can declare as a register, only local variables whose addresses are not taken; whereas, the optimizer allocates global variables as well as variables whose addresses are taken (where possible).
- The user can allocate variables in safe registers only. Therefore, every register which is used has to be saved/restored at the entry/exit of the procedure. The optimizer allocates variables that do not live across procedure calls in unsafe registers. Therefore, these registers need not be saved/restored.
- Because of code motion optimizations, the number of references of variables may be changed. Therefore, the choice of register variables may not be optimal. In the following example:

```
int  j;
register int i;
i = j;
if (i == 3 || i == 4 || i == 5)
```

undesired effects result if optimized with the `/USER_REGISTERS` flag (`-Ou` on UNIX systems) The reason is that `j` is copy propagated and replaces all occurrences of `i`. Therefore, `i` occupies a register for nothing, while `j` may end up in memory (because either the ordinary register allocation of the optimizer is not invoked or there are no registers left for `j`).

6.6.8 setjmp()

Calls to `setjmp()` are specially recognized by the compiler. Procedures that contain calls to `setjmp()` are only partially optimized because procedure calls may end up in a call to `longjmp()`. Code motion optimizations are performed only within linear code sequences (those sequences not containing branches or branch targets). Register allocation is limited to optimizer generated temporary variables, register declared variables, and variables whose live ranges do not contain function calls.

6.6.9 Optimizing for Space

The default behavior of the GNX—Version 4 C Compiler optimizes for optimal speed without undue increase in code size. There are several things that can be done to improve code density:

- Optimize with the `NOSPEED_OVER_SPACE` flag on (`-Os` on UNIX systems).
- Squeeze the data area by using smaller alignment between variables, *i.e.*, `/TARGET=(BUS=1)` on VMS systems or `-KB1` on UNIX systems.
- Do not use loop-unrolling optimization.

  The optimizer has certain heuristics based on size considerations, whether to perform loop-unrolling. These heuristics also take into account the relevant on-chip caches. Nevertheless, if code density is important, it is advisable not to use the loop-unrolling optimization.
- Squeeze all structure definitions by using the `/ALIGN=1` switch (`-J1` on UNIX systems). See Section 4.2.4.

6.6.10 Using /NOOPT (-Oo) option

The `/OPTIMIZE=NOOPT` (`-Oo`) compiler option is used when the optimizer phase is not required and another flag needs to be turned off as well. For instance, `/OPTIMIZE=(NOOPT,FIXED_FRAME)` (`-OoF`) turns on fixed frame without running the optimizer, while `/OPTIMIZE=(NOOPT,FIXED_FRAME)` (`-Of`) turns off fixed frame but runs the optimizer.

With `/OPTIMIZE=NOOPT` (`-Oo`) by itself, the optimizer is not run, but the code-generator performs all its optimizations (see Section 5.3).

6.6.11 Runtime Feedback Optimization

In the runtime feedback optimization the optimizer makes use of runtime information collected during a previous runs of the program in order to better predict the program's run-time behavior. As a result, optimizations done by the compiler will be based on assumptions closer to the real behavior of the program. Runtime feedback optimization effects mainly the following optimizations:

- Loop Unrolling
- Register Allocation

Runtime optimization is invoked by using the `-OB` (`/OPT=RUNTIME_FEEDBACK`) compiler option. The PIT file in the current directory (or the one specified in the PITFILE environment variable global symbol) is assumed to contain the data for the runtime feedback option.

NOTE: The runtime information used for the runtime feedback optimization should represent the real run-time behavior of the program. Otherwise, the optimizer relies on false assumptions. This may actually cause degradation in code quality.

For more details on runtime feedback optimization see Section 7.4.

6.7 COMPILATION TIME REQUIREMENTS

Using the optimizer slows down the compilation process. It is therefore recommended to use the optimizer only on final production versions of a program. The amounts of resources (time and memory) vary strongly from program to program and actually depend on the size of the routines in the compiled program file. The larger a routine, the more time and memory needed to optimize it. This behavior is more or less quadratic, the optimizer needs about four times the resources to optimize a routine of 1000 lines than to optimize a routine of 500 lines.

If time or memory requirements are unacceptable and routines cannot be reduced to reasonable (500 lines) size, it is possible to turn off some optimizations, using the `NOCODE_MOTION` (`-Om` on UNIX systems) and/or the `NOREGISTER_ALLOCATION` (`-Or` on UNIX systems) flags.

On UNIX host systems, an optimization flag (`-O`*number*) is available to set an upper limit on the memory requirements of the optimizer to a certain number of megabytes. This can be useful on host systems without virtual memory or with a limited swap-space configuration. If necessary, the optimizer then skips certain optimizations on huge routines only, in order to stay under the chosen limit. In such cases, an appropriate message is given. This flag is only necessary when compiling modules with extremely large procedures (over 500 lines in a single procedure), a case when the optimizer may need a larger swap space than the one currently available. For instance,

```
-O2
```

limits the optimizer to 2 Mbytes of swap space.

An alternate method for setting an upper limit on memory requirements, on native systems, is to use the environment variable AVAIL_SWAP, which sets the maximum swap space requirement of the optimizer in megabyte units. This environment variable should be set to the number of megabytes to be used. The user can choose from 1 Mbyte to 16 Mbytes. If the user's choice is outside of these parameters, the default value of 4 Mbytes is chosen. For instance,

```
setenv AVAIL_SWAP 2
```

makes 2 Mbytes of swap space the default. This can be overridden using the previously described `-O`*number* option.

Chapter 7

PROFILE INFORMATION

7.1 INTRODUCTION

Profile information is statistical data about the run-time behavior of a program. Such information can be gathered by compiling the program using the `-B` option on UNIX (`/GATHER` on VMS), and executing the compiled program with typical inputs. Each execution of the compiled program results in the accumulation of profile information in a special file. Profile information is used by the optimizer and the tool `sprof`.

The optimizer can use profile information to achieve better code optimization. Code can be recompiled using the compiler option `-OB` on UNIX (`/OPT=RUNTIME_FEEDBACK` on VMS).

`sprof` processes profile information and displays it as an annotated source file. You can use `sprof`'s output to:

- Pinpoint the most often executed sections of program code in order to determine areas for concentrated hand optimization.
- Test the expected relative frequency of execution of different code sections.
- Provide indication of test coverage.
- Discover bugs by spotting unexpected execution of code lines.

7.2 GATHERING PROFILE INFORMATION

7.2.1 The Profile Information

Profile information is gathered for each basic block (see Section 5.2). A precise trace of every basic block execution is recorded. From this information the execution rate of each line can be deduced.

7.2.2 Code Compilation

When compiling a program on which we want to gather profile information additional code is generated by the compiler. Also a profile information tables (PIT) file is created. When the program is executed, the additional code that was generated accumulates profile information internally and adds it to the PIT file before the program exits.

The extra code is generated by the compiler and the PIT file is created by the tool `pgen` after the linking phase.

The following operations are performed:

1. Allocation of buffer space in the .bss section (uninitialized data). The buffer space is used for basic block execution counters, which keep track of the number of times each basic block is executed.
2. Insertion of extra code at the beginning of each basic block. This code increments the proper basic block execution counter each time a block is executed.
3. Generation of additional symbolics. Additional symbolics are generated in order to map the basic blocks in the executable file to source lines.
4. Linking of the program with the object file `pfb_exit.o` (`pfb_exit.obj` on VMS and `db_pfb_exit.o` for cross compilation on a *Series 32000*/UNIX system). This file includes a customized version of the standard library `_exit` routine, which accumulates the internally accumulated profiling information into the PIT file at the end of each execution.
5. Creation of the PIT file. The tool `pgen` is invoked to create and initialize the PIT file.

See section 7.2.4 for more information on the PIT file and section 7.2.3 for more information on `pgen`.

NOTE: Profile information cannot be gathered on an optimized program. The optimizer is not to be invoked together with the profile information gathering option.

7.2.3 Pgen

The tool `pgen` reads the executable file and generates the profile information table (PIT) file that is used by `sprof` and the compiler (see Section 7.2.4).

On UNIX systems `pgen` is automatically invoked by the driver when called with the `-B` flag (Section 7.2.6). On VMS systems `pgen` must be invoked separately after linking the program compiled with the `/GATHER` qualifier (Section 7.2.7).

7.2.4 The PIT File

The PIT (Profile Information Tables) file contains accumulated profile information.

The PIT file is created by `pgen` (see Section 7.2.3) and modified by the additional code in the `pfb_exit` object file (Section 7.2.5) just before execution is completed.

The PIT file is used by `sprof` and by the profile feedback option. `PIT` is the default name for the file in which profile information accumulates. In order to override this default name, the environment variable (logical name on VMS) PITFILE is used. If PITFILE is set during profile gathering, information is accumulated to a file bearing this name.

7.2.5 The `pfb_exit.o` (`pfb_exit.obj`) File

This file must be linked with the profiled program in order to enable profile-information accumulation. The `pfb_exit.o` (`pfb_exit.obj` or `pfb_exit.obj` on VMS) file includes a customized version of the `_exit` routine of the C runtime-library, which accumulates the profile information into the PIT file at the end of program execution.

In the UNIX environment, linkage with `pfb_exit.o` is performed automatically by the compiler when compiling with the `-B` option (see Section 7.2.6). In the VMS environment, linking with `pfb_exit.obj` must be done by the user (see Section 7.2.7).

NOTES: 1. For cross compilation on a *Series 32000*/UNIX system, the file is named `db_pfb_exit.o`.

2. In native environment, `pfb_exit.o` uses the standard I/O library of `libc` for writing the PIT file.
In the cross environment virtual I/O facilities of the cross C library, which are based on debugger and monitor support, are used.

3. For modular compilation (`-X` option on UNIX, `/MODULAR` on VMS) a special version of the pfb_exit.o exists. On native UNIX environment it is called `Xdb_pfb_exit.o`, on cross configuration UNIX environment `Xpfb_exit.o`, on VMS environment `Xpfb_exit.obj`.

7.2.6 Compilation in the UNIX Environment

The syntax for compilation to gather profile information on the UNIX environment is:

```
nmcc -B[<pitfile>] filename  (cross-support configuration)
cc -B[<pitfile>] filename  (native configuration)
```

`<pitfile>` is the name of the PIT file to be created. If `<pitfile>` is omitted, the default name `PIT` is given. Note that there should be no space between `-B` and `<pitfile>`.

The compiler driver automatically calls all the necessary subprograms when invoked with the `-B` option. This includes linking with the special `pfb_exit.o` file (Section 7.2.5) and calling `pgen` (Section 7.2.3) after linking.

7.2.7 Compilation in the VMS Environment

The syntax for compilation to gather profile information on the VMS environment consists of three steps:

1. Compilation. Use the /GATHER compiler qualifier in the syntax:

```
nmcc /GATHER my_module.c
```

2. Link the program with `pfb_exit.obj`. Use the syntax:

```
nmeld gnxdir:crt0.obj, gnxdir:pfb_exit.obj,-
      my_module.obj, ... ,gnxdir:libc.a
```

The `pfb_exit.obj` file must appear before the standard libraries.

For modular compilation (`/MODULAR`) linkage is done with an object file named `Xpfb_exit.obj`.

3. Run the tool `pgen` to create the PIT file. Use the syntax

   ```
   pgen <executable_name> [<PIT_file_name>]
   ```

   If `<PIT_file_name>` is not specified, it will be named `PIT` by default.

7.2.8 Code Execution

The profiled program can be executed repeatedly with any desired inputs. Profile information from each execution accumulates in the PIT file.

7.2.9 Disabling Profile Information Accumulation

Profiling information will not be accumulated under any of the following conditions:

— No PIT file exists

— Read or create permission for the PIT file is denied

— The PIT file is not in the expected format

— The PIT file and executable file are incompatible

— The cumulative number of executions of a certain basic block in the profiled program exceeds the maximum count limit, which equals the maximum unsigned long integer minus one (decimal 4294967295).

These conditions cause only the accumulation of profiling information to be disabled, and do not affect the normal operation or semantics of the profiled program.

The PIT file can be removed or renamed temporarily in order to disable the accumulation of profiling information.

7.2.10 Redefining Standard `libc` Symbols

The following standard `libc` symbols (routines and variables) are used by the profile gathering code:

```
_cleanup
close
errno
etext
_exit
```

```
fclose
fgets
fopen
fprintf
fputs
getenv
_iob
mktemp
rename   (link and unlink in native SYS-V systems)
rindex   (strrchr in native SYS-V systems)
sprintf
sscanf
sys_errlist
sys_nerr
unlink
```

Redefining any of these symbols can cause unexpected results.

7.2.11 Execution Time Considerations

The additional code produced in each basic block for gathering profile information may slow down a CPU-bound program by a factor of 20%-30% (without taking I/O into account).

On cross systems where the loading of a program and I/O operations are on slow serial lines, use of `sprof` may slow down execution significantly. This is because I/O operation will be performed during the accumulation of the profiling information (just before exiting).

7.2.12 Space Considerations

The additional code and space needed the PIT file adds approximately 20%-30% to the original code and uninitialized data size.

Additional symbolic information is also produced. However this symbolic information occupies only disk space and is not loaded into memory (since it is not a component of real code or data).

7.3 SPROF - THE GNX SOURCE PROFILER

The `sprof` profiler provides high-level language information about the runtime behavior of a program. The profile consists of an annotated listing of the source file. A number is printed at the beginning of each line to indicate the number of times that line was executed. `sprof` is supplied as part of the GNX—Version 4 C compiler package, and runs on all cross and native GNX supported environments.

Unlike other profilers, `sprof`'s provides information on the basic block level (see Section 5.2). This means that `sprof` does not sample the program periodically, but instead gathers a precise trace of every basic block execution. From this information the execution rate of each line can be deduced.

`sprof` does not provide either timing information about a program or function caller/callee statistics. Rather, `sprof` gives an exact count of source-line executions. A standard profiler, such as the UNIX profiler `prof` (supported by the compiler in native environment), can be used to collect timing or caller/callee information.

7.3.1 Example

```
   main()                                                    /*line 1*/
   {                                                         /*line 2*/
       int i;                                                /*line 3*/
                                                             /*line 4*/
 1     for (i = -6; i <= 7; i++)                             /*line 5*/
14        if (i >= 0)                                        /*line 6*/
 8           printf("factorial(%d) = %d0,i,fact(i));  /*line 7*/
 1   }                                                       /*line 8*/

   int fact(n)                                               /*line 10*/
   int n;                                                    /*line 11*/
   {                                                         /*line 12*/
36     if (n == 0)                                           /*line 13*/
 8        return 1;                                         /*line 14*/
       else                                                  /*line 15*/
28        return (n * fact(n - 1));                         /*line 16*/
 0   }                                                       /*line 17*/
```

Figure 7-1. Example of **sprof** Output

The number "1" annotating the first basic block on line 5 indicates that the main program was executed once. The "14" and "8" annotations of lines 6 and 7 indicate that the body of the loop in `main` was executed 14 times, of which only 8 resulted in a call to the function `fact`.

It can be deduced that the function `fact` did not returned implicitly (i.e. without using an explicit `return`), as shown by the "0" annotation of line 17. In contrast, the function `main` did return implicitly, as can be seen from the "1" annotation besides the closing brace of function `main` (line 8).

The runtime behavior of the function `fact` is also illustrated. `fact` was called 36 times. Only 8 of these invocations were from `main`, therefore the remaining 28 calls were recursive. This is further shown by the annotation of "28" on line 16.

7.3.2 Running SPROF

After program execution and accumulation of profile information in the PIT file, `sprof` can be called to process and present profile information. Various options are available to control the output (see Section 7.3.3).

In order to process profile information the following files must exist:

- The source file(s)
- The executable file
- The PIT file

Figure 7-2 illustrates the flow of data through `sprof`. `nnm` is the standard GNX utility for displaying the symbol table of a COFF object file (see the *GNX Commands and Operations Manual*).

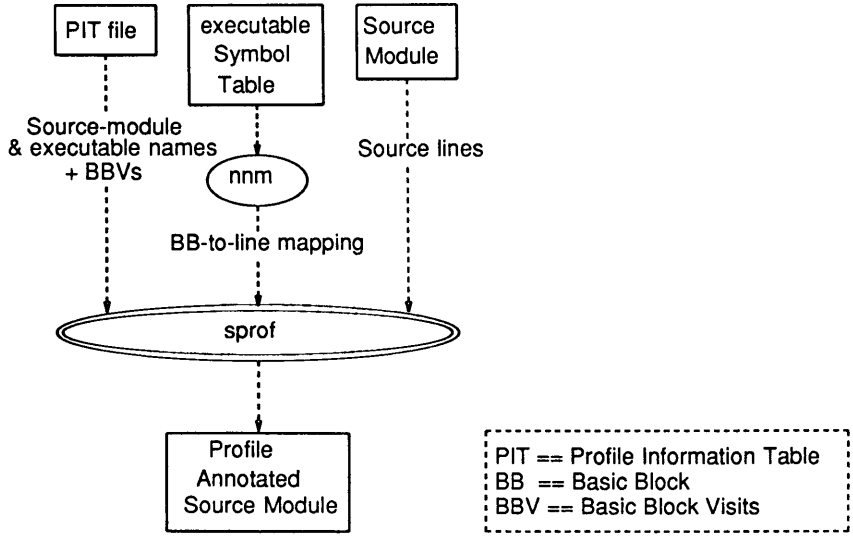


Figure 7-2. `sprof` Data Flow Description

7.3.3 SPROF Invocation

The tool `sprof` is supplied as a standard part of the GNX—Version 4 C compiler package both on UNIX and VMS.

SYNTAX (UNIX):

```
sprof [-d source_dir] [-e exec_file ]
      [-p pit_file] [-o output_file]
      [-f[fmc<margin_width>]] [source_file ...]
```

SYNTAX (VMS):

```
SPROF [/DIRECTORY=source_dir] [/EXECUTABLE=iexec_file ]
      [/PITFILE=pit_file] [/OUTPUT=output_file]
      [/FORMAT=([format_option [, ...] ]
      [source_file ...]
```

Where:

source_dir The directory where the source file is located. The default is the current directory.

exec_file The name of the executable file. The default is the executable name as found in the PIT file.

pit_file The name of the PIT file to be used. The default name is `PIT` or the value of the environment variable PITFILE.

output_file The name of the output file to be generated. The default is standard output (SYS$OUTPUT on VMS).

source_file The name of the source file to be profiled. The default is all source files in the executable file that were compiled with the `-B` option.

The -f options (*format_option* on VMS) are:

`f` (`FORMFEED` on VMS)
output a FORM-FEED character between output source files

`m` (`MARK_MARGIN` on VMS)
mark the MARGIN of the source with vertical bar "|" characters.

`c` (`COMPACT` on VMS)
print counts COMPACT (the count for sequential basic blocks is only printed if different from the previous basic block). The default is to print counts for every basic block.

`<margin-width>` (`WIDTH=`*margin_width* on VMS)
The width reserved for printing profiling counts. The default is 8. A negative value will left-justify the counts.

CAUTION

Compatibility of the PIT file and the program source file is determined only by the creation date of the PIT file. Therefore, a source file with the same name, and an older date than the PIT file, but with contents different from the program source file, will not be recognized by `sprof` as being incompatible with the PIT file. Such a situation will result in an incorrect `sprof` output.

7.3.4 Counts and Basic Blocks

`sprof` prints basic block counts according to the following rules:

1. Print a count only for lines which start a basic block. (No count will be printed for lines which consist of declarations or calls to a cpp macro defined to nothing.)
2. Print a count only for the first basic block of a group that is mapped to the same line.

Example:

```
100   if (a < 5) j = 3;
```

The count in the profiled line is the number of times the first basic block in this line (i.e `if (a<5)`) was executed. However, the number of times the condition was true can not be deduced from this output. This information is provided if the code is written in the form

```
100   if (a < 5)
 27       j = 3;
```

7.4 RUNTIME FEEDBACK OPTIMIZATION

The runtime feedback optimization option is used to enable the compiler to tune optimizations based on statistical information on typical run-time behavior.

The optimization algorithms used by the optimizer are based on assumptions and heuristics. However, run-time behavior may be different. In such a case, the compiler can achieve better optimization by operating under a different set of assumptions as suggested by the profile information.

For example, the following optimizations can be improved by using the profile feedback option:

- Register allocation - Usually the optimizer heuristics used to determine register allocation are based on loop nesting and conditional execution. By using the profile information, register allocation is based on an better estimate of the use of variables.
- Loop unrolling - Loop unrolling optimization enlarges code size. Therefore it is worthwhile to optimize only those loops which are entered many times. The profile information provides improved estimation of where this optimization should be performed.

For more details on these optimizations see Section 5.2.

The runtime feedback mechanism is divided into two phases, described in the following two sections.

7.4.1 Profile Information Gathering

This phase involves compilation and program execution. It is described in detail in Section 7.2. The profile information is collected in the PIT file (see Section 7.2.4).

NOTE: The profile information gathered in the PIT file must represent the true run-time behavior of the program. Otherwise false assumptions are made by the optimizer and recompilation can cause degradation in program performance.

7.4.2 Runtime Feedback Compilation

The runtime feedback compilation is invoked using the `-OB` (`/RUNTIME_FEEDBACK` on VMS) compiler option. The PIT file in the current directory, or the one specified in the PITFILE environment variable (logical name on VMS) is assumed to contain the data for the runtime feedback option.

Chapter 8

INTRINSIC FUNCTIONS

8.1 INTRODUCTION

The GNX C compiler generally uses in its code selection the most efficient instructions from the *Series 32000* instruction set. There are, however, certain instructions which have no natural matching C construct or that are not fully utilized by the C language. Such instructions are in particular the Application Specific Instruction Set (ASIS) of the NS32CG16, NS32CG160, NS32FX16 and NS32GX320 microprocessors (see the *Series 32000/EP Embedded Processor Databook*) and certain standard *Series 32000* instructions. The GNX C compiler provides access to these instructions by means of intrinsic functions. The NS32CG160, NS32FX16 and NS32CG16 CPUs share the same core instructions. These microprocessors will be referred to as the CG-Core throughout this chapter. A familiarity with ASIS instructions is assumed.

In order to use intrinsic functions, include the appropriate GNX header file `cg16.h`, `gx320.h` or `ns32000.h`, prior to any call to the function. The function call must contain a parameter list compatible with the formal parameter list of the prototype. Otherwise it is considered a redeclaration of the function, and a proper warning message is issued by the compiler.

Redefining an intrinsic function, i.e., defining a different function with the same name as an intrinsic function, results in an error. However, it is possible to use intrinsic function names for different functions by specifying the `-Fl` or `-Ol` option (NO_STANDARD_LIBRARIES on VMS). (See Section 6.3.5).

Special compilation options generate various run-time checks for flagging improper values of parameters in calls to intrinsic functions. The compilation options and the run-time checks performed are described in Chapter 2, Section 2.5.

8.1.1 Using Intrinsic Functions

Intrinsic functions are known internally to the compiler. The syntax used for invoking intrinsic functions is the same as for regular C functions. However, no function call will appear in the code generated by the compiler. Instead, an instruction sequence containing the application specific instruction will be produced. The optimizer will attempt to allocate parameters in the registers as needed by the instruction.

NOTE: Unlike regular functions, taking the address of an intrinsic routine is not permitted. Any attempt to do so will results in an error message.

There is one intrinsic function for each supported special instruction. Generally the function's name is the instruction assembly mnemonic, with a leading underscore. This conforms to the ANSI C convention of global identifiers with leading underscores being reserved for implementation. The parameters and the result type of each function are described in full ANSI C prototype format in special GNX header files. Currently there are three such header files:

- `ns32000.h`
  For the general Series 32000 instructions.
- `cg16.h`
  For the CG-Core Application Specific instructions.
- `gx320.h`
  For the NS32GX320 Application Specific instructions.

A complete description of each function is given in Sections 8.3, 8.4 and 8.2, respectively. Examples of using the functions are provided in Appendix E.

8.2 General Series 32000 Intrinsic Functions

This section describes the special *Series 32000* instructions accessible by use of intrinsic functions. Supported instructions are divided into:

Single bits
Instructions that refer to a single bit in memory. They enable efficient setting (`sbit`), clearing (`cbit`), inverting (`ibit`) and testing (`tbit`) of a single bit. Single bit instructions find the first set bit in a given byte, word or double-word (`ffs`), and calculate an absolute bit address (`cvtp`).

Bit-Fields
Instructions that manipulate a consecutive group of bits in memory. They include `ext`, which extracts a bit-field, and `ins`, which inserts a bit-field.

Absolute value
Integer absolute value (`abs`) and floating-point absolute value (`fabs` and `ffabs`).

All definitions given in this section are supplied as part of the GNX C compiler package in the file `ns32000.h`.

8.2.1 Single Bit Instructions

PROTOTYPE

```
typedef int _xbit(int offset,
                  int *base);

_xbit _cbit;    /* clear bit  */
_xbit _ibit;    /* invert bit */
_xbit _sbit;    /* set bit    */
_xbit _tbit;    /* test bit   */
```

DESCRIPTION

The `cbit`, `ibit` and `sbit` functions operate on the bit at `offset` bits from `base`. The return value is the original value of the specified bit.

`_cbit` clears the bit to zero.

`_ibit` inverts the bit — zero becomes one and vice versa.

`_sbit` sets the bit to one.

The `tbit` function returns the value of the bit residing at `offset` from `base`.

The appropriate instruction version (byte word or double word) is selected by the compiler according to the type of parameter passed for `offset`. Calling the functions as a procedure, such as

```
(void)_ibit(...
```

will prevent the compiler from producing the code reading the original value of the bit.

8.2.2 _ffs (Find First Set)

PROTOTYPE

```
typedef int _ffsb(unsigned char base,      /* find first set byte
                  unsigned char *offset);

typedef int _ffsw(unsigned short base,     /* find first set word
                  unsigned char *offset);

typedef int _ffsd(unsigned int base,       /* find first set double-wo
                  unsigned char *offset);
```

DESCRIPTION

The `ffs` routines search for the first set bit in `base`. The search starts at the offset specified in the int pointed by `offset`, and proceeds in ascending order to the first set bit or to the last bit in base.

The routines return the value of the PSR F flag. If a set bit is found, the value of the int pointed by `offset` is changed to the bit number of the bit found, and the F flag in the PSR is cleared. If no set bit is found, the int pointed by `offset` is set to zero and the F flag in the PSR is set to 1.

Calling the functions as a procedure, such as

```
(void)_ffsw(...
```

will prevent the compiler from producing the code reading the PSR F flag.

NOTE: The value of the int pointed by `offset` must be within tha range 0 to 7 (in `ffsb` routine), 0 to 15 (in `ffsw` routine) and 0 to 31 (in `ffsd` routine). Compilation with the `-aa` option (`/CHECK=PARAMETER` on VMS) generates code to check in run-time for this limitation.

8.2.3 _exti (Extract bit-field)

PROTOTYPE

```
void _extb(int offset,
           void *base,
           char *dest,
           int length);

void _extw(int offset,
           void *base,
           short *dest,
           int length);

void _extd(int offset,
           void *base,
           int *dest,
           int length);
```

DESCRIPTION

The `ext` routines copy the bit-field specified by `base`, `offset` and `length` to the `dest` operand. The field is right justified in `dest`. High-order bits are zero-filled if the field is shorter than `dest` or discarded if the field is longer than `dest`.

The field starts at bit position

```
offset % 8
```

within the memory byte

```
base + (offset / 8)
```

NOTES: 1. `length` must be a constant. Otherwise the routine is not inlined and an emulation function is called.

2. `length` must be in the range 1 through 32. Compilation with the `-aa` option (`/CHECK=PARAMETER` on VMS) will generate code to check in run-time for this limitation.

CAUTION

Although a bit-field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits. A bit-field must be composed of bits from no more than four contiguous bytes.

If the `offset` operand is a constant expression with a value in the range 0 to 7, and `length` is a constant expression, the compiler will use the short version of the instruction (`exts`).

8.2.4 _ins (Insert Bit-field)

PROTOTYPE

```
void _ins(int offset,
          unsigned int src,
          int *base,
          int length);
```

DESCRIPTION

The `_ins` routine inserts the `src` operand into the bit-field specified by `base`, `offset` and `length`. The `src` operand is right-justified in the field. High-order bits are zero-filled if `src` is shorter than the field or discarded if `src` is longer than the field.

The field starts at bit position

```
offset % 8
```

within the memory byte

```
base + (offset / 8)
```

`length` specifies the number of bits in the field, and must be in the range 1 through 32. Compilation with the `-aa` option (`/CHECK=PARAMETER` on VMS) will generate code to check in run-time for this limitation.

NOTES: 1. `length` must be a constant. Otherwise the routine is not inlined and an emulation function is called.

2. `length` must be in the range 1 through 32. Compilation with the `-aa` option (`/CHECK=PARAMETER` on VMS) will generate code to check in run-time for this limitation.

CAUTION

Although a bit-field may contain up to 32 bits, an alignment restriction appears for fields containing more than 25 bits. A bit-field must be composed of bits from no more than four contiguous bytes.

If the `offset` operand is a constant expression with a value in the range 0 to 7, and `length` is a constant expression, the compiler will use the short version of the instruction (`inss`).

The appropriate version for `_ins` (`insb`, `insw` or `insd`, for byte, word or double-word respectively) is selected by the compiler according to the type of the parameter passed for `src`.

8.2.5 _cvtp (Convert to Bit Pointer)

PROTOTYPE

```
int _cvtp(int offset,
          void *base);
```

DESCRIPTION

The `cvtp` function returns the absolute bit address of the memory bit specified by `base` and `offset`. The bit address specifies the number of bits from the first bit in the memory space (bit 0 of the byte at address 0) to the specified bit.

8.2.6 abs (Absolute Value)

PROTOTYPE

```
int abs(int val);          /* integer absolute value */
double fabs(double val);   /* double absolute value  */
float ffabs(float val);    /* float  absolute value  */
```

DESCRIPTION

The `abs` routines return the absolute value of their parameter `val`. `abs` compiles into the integer ABS instruction, `fabs` compiles into the floating-point ABSL instruction and `ffabs` compiles into the floating-point ABSF instruction. These routines differ from the other intrinsic routines: Their names do not begin with an underscore, and the floating-point versions' names are not the same as the assembly mnemonic.

The reason for these differences is that the routines exist in the mathematical library `libm.a`, and are declared in the header file `math.h`.

8.3 CG-Core Intrinsic Functions

The CG-Core microprocessors complements the full instruction set of the *Series 32000* processor with special graphics-oriented instructions. These include Bit-aligned Block Transfer (BITBLT) functions, line drawing, pattern replications and data compression-expansion. In addition, an interface to an external BITBLT processing unit (BPU) enables very fast BITBLT operations.

The CG-Core graphic instructions supported by intrinsic functions are:

`bband` Bit-aligned Block Transfer, 4-direction bitwise AND operation for CRT applications.

`bbfor` Bit-aligned Block Transfer, 2-direction FAST OR operation optimized for printers.

`bbor` Bit-aligned Block Transfer, general 4-direction OR operation.

`bbstod` Bit-aligned Block Transfer, 4-direction replace (Source-to-Destination) operation.

`bbxor` Bit-aligned Block Transfer, 4-direction XOR operation.

`bitwt` Bit-aligned Word Transfer, for small BITBLT OR operations.

`extblt` Drives a 4-direction, 16-function external BITBLT processing unit. The BPUs supported are the DP8510 or DP8511 for the NS32CG16 CPU, or the on-chip BPU for the NS32CG160.

`movmp` Move Multiple Pattern, for pattern fill, horizontal line drawing, memory clear.

`sbitps` Set Bit Perpendicular String, for image expansion and horizontal/vertical/diagonal line drawing.

`sbits` Set Bit String, for image expansion and horizontal line drawing.

`tbits` Test Bit String, for image data compression.

In addition, the general *Series 32000* instruction set includes instructions that manipulate single bits or bit-fields in memory. These are useful for graphic operations and are supported by intrinsic functions, as described in Section 8.4.

TERMS AND CONVENTIONS

bit alignment The ability to access any bit in memory. The *Series 32000* instruction set and the CG-Core instructions enable efficient direct access to bit-aligned data.

bit-block A rectangular sub-area of an image. A bit-block consists of *y* lines of *x* bits, where the spacing between lines is warp bits.

bit ordering Bit ordering is from the least significant to the most significant bit. Bit positions increase left-to-right in the image.

image Images are defined as binary bit-maps, where a set bit (one) represents a black dot, and a clear bit (zero) represents a white dot.

left-to-right A graphic operation that traverses memory in increasing address order (see **memory ordering** below).

memory ordering The displayed image's memory increases in a left-to-right, top-to-bottom raster direction. The first byte represents the image's top-left corner. The *Series 32000* architecture is "little-endian" in referring to a word (16 bits) or double-word (32 bits) — the least significant byte is stored at the lowest address.

right-to-left A graphic operation that traverses memory in decreasing address order (see **memory ordering** above).

shift The shifts in this chapter are *Series 32000* logical left bit shifts. With the imaging convention this results in image bits moving from left to right.

warp The horizontal width of the image. Also known as raster or pitch.

All definitions given in this section are supplied as part of the GNX C compiler package in file `cg16.h`.

8.3.1 _extblt (External Bit Aligned Block Transfer)

PROTOTYPE

```
void  _extblt(char *src_addr,  /* Extblt without preloading */
              char *dest_addr,
              int adj_width,
              int height,
              int horiz_incr,
              int adj_src_wrap,
              int adj_dest_wrap);

void  _extbltp(char *src_addr,    /* Extblt with preloading */
              char *dest_addr,
              int adj_width,
              int height,
              int horiz_incr,
              int adj_src_wrap,
              int adj_dest_wrap);
```

PARAMETERS

`src_addr` The base byte-address of the source data, the value should be even.

`dest_addr` The base byte-address of the destination, the value should be even.

`adj_width` Adjusted width of the image on which the operation is performed. The adjustment is the width in words of destination data multiplied by `horiz_incr`:
`(width * horiz_incr)`

`height` The number of lines on which the operation is performed.

`horiz_incr` The horizontal step in bytes for copying. Its value should be +2 or -2.

`adj_src_warp` The adjusted wrap of the source.

The adjustment is to the actual source warp in bytes minus the width in bytes (*not* the adjusted width) less two:

```
(source warp - (width in bytes - 2))
```

`adj_dest_warp` The adjusted wrap of the destination.

The adjustment is the actual destination warp in bytes less the width in bytes (*not* the adjusted width) less two:

```
(destination warp - (width in bytes - 2))
```

DESCRIPTION

The two `extblt` functions drive an external BITBLT processing unit (BPU). The BPUs supported are the DP8510 or DP8511 for the NS32CG16 CPU, or the on-chip BPU for the NS32CG160. The CPU supplies addresses and bus cycles while the BPU operates on the data. For more details on the EXTBLT instruction refer to the NS32CG16 or the NS32CG160 Printer/Display Processor Programmer's Reference Supplement.

NOTE: Compilation with the `-aa` option (`/CHECK=PARAMETER` on VMS) will generate code to check in run-time for the following:

- "src_addr" and "dest_addr" values are even.
- "horiz_incr" is +2 or -2.
- "width" value is a multiple of "horiz_incr" and has the same sign.

8.3.2 BITBLT instructions

PROTOTYPE

```
typedef void  _bbfunc(char *src_addr,
                      char *dest_addr,
                      unsigned char shift_val,
                      unsigned int height,
                      unsigned int mask1,
                      unsigned int mask2,
                      int adj_src_warp,
                      int adj_dest_warp,
                      unsigned short width);

_bbfunc _bbfor,       /* plain, fast or */
        _bbor_s,      /* bbor with inverted source */
        _bbor_da,     /* bbor with decreasing addresses */
        _bbor_sda,    /* bbor with inverted source and
                       * decreasing addresses */

        _bband,       /* plain bband */
        _bband_s,     /* bband with inverted source */
        _bband_da,    /* bband with decreasing addresses */
        _bband_sda,   /* bband with inverted source and
                       * decreasing addresses */

        _bbxor,       /* plain bbxor */
        _bbxor_s,     /* bbxor with inverted source */
        _bbxor_da,    /* bbxor with decreasing addresses */
        _bbxor_sda,   /* bbxor with inverted source and
                       * decreasing addresses */

        _bbstod,      /* plain bbstod */
        _bbstod_s,    /* bbstod with inverted source */
        _bbstod_da,   /* bbstod with decreasing addresses */
        _bbstod_sda;  /* bbstod with inverted source and
                       * decreasing addresses */
```

DESCRIPTION

The BITBLT instructions perform a full two-operand, bit-aligned block transfer. Sixteen intrinsic BITBLT functions are supplied as an interface to these instructions. They are divided into four groups, according to the operator between the source and destination bits: `_bbor` for an OR operator, `_bband` for an AND operator, `_bbxor` for a XOR operator, and `_bbstod` for a source to destination copy, which overwrites the destination bits.

Within each group there are four variants, resulting from the combination of:

- The direction of the transfer — increasing or decreasing addresses. Increasing addresses indicate a left-to-right operation; decreasing addresses indicate a right-to-left operation.
- The reading of the source data — with or without inversion. The inversion is a logical negation of each source bit.

A separate intrinsic function is supplied for each combination of variants. These are coded as suffixes to the function names. The suffixes are `_s` for inverted source, `_da` for decreasing addresses, and `_sda` for inverted source and decreasing addresses. No suffix signifies a simple version of the function (i.e. no source inversion, increasing addresses). There are therefore 16 different BITBLT functions, as shown in the table below:

Operator	plain	inverted source	decreasing addresses	inverted source and decreasing addresses
AND	`_bband`	`_bband_s`	`_bband_da`	`_bband_sda`
OR	`_bbfor`	`_bbor_s`	`_bbor_da`	`_bbor_sda`
XOR	`_bbxor`	`_bbxor_s`	`_bbxor_da`	`_bbxor_sda`
STOD	`_bbstod`	`_bbstod_s`	`_bbstod_da`	`_bbstod_sda`

NOTE: No plain `bbor` function is supplied since the CG-Core `bbfor` instruction has the same functionality with faster performance.

A bottom-to-top BITBLT operation can be performed by giving negative source and destination warp values, and beginning from the bottom line of the image. Thus, combined with the `_da` option, the BITBLT functions can manipulate blocks of data beginning in any of its four corners.

All functions have the prototype given above. The parameters are the same as the instruction operands with the following meanings:

- `src_addr` is the base byte-address of the source bit-block
- `dest_addr` is the base byte-address of the destination.
- `height` is the vertical size of the bit-block, which specifies the number of lines to be transferred.
- `width` is the horizontal size of the transferred bit block. It is the number of whole words on one line containing the bit-block.
- `mask1` and `mask2` are bit masks "protecting" those bits at the left and right of the source word block that do not belong to the bit-block from affecting the bits in the destination block. A bit set to one means that this bit should not be protected. A clear bit means that this bit should not affect the destination bit.

  The upper 16 bits of the `mask` parameters must be clear, otherwise behavior is undefined. Compilation with the `-aa` compiler option (`/CHECK=PARAMETER` on VMS) will generate code to check in run-time for this condition.

- `shift_val` contains the difference between the bit offsets of the source and destination bit-blocks, relative to the word block:

  ```
  shift_val = destination bit offset - source bit offset
  ```

  `shift_val` must be positive. `src_addr` and `dest_addr` may need to be adjusted to ensure this.

- `adj_src_warp` describes the adjusted source warp. For left-to-right operations the source warp must be adjusted to:

  ```
  adj_src_warp = source warp - 2*(width - 1)
  ```

  For right-to-left operations the source warp must be adjusted to:

  ```
  adj_src_warp = source warp + 2*(width - 1)
  ```

- `adj_dest_warp` describes the adjusted destination warp. For left-to-right operations the destination warp must be adjusted to:

  ```
  adj_dest_warp = destination warp - 2*(width - 1)
  ```

  For right-to-left operations the destination warp must be adjusted to:

  ```
  adj_dest_warp = destination warp + 2*(width - 1)
  ```

For more details about the BITBLT instructions refer to the appropriate CG-core Processor Programmer's Reference Supplement.

8.3.3 _bitwt (Bit Aligned Word Transfer)

PROTOTYPE

```
void  _bitwt(unsigned short **src_addr,
             unsigned  **dest_addr,
             int shift_val);
```

DESCRIPTION

The `_bitwt` function performs a bit-aligned transfer of a short int. The 16 bits of data at `**src_addr` are read, zero extended to an unsigned int and shifted to the left by the number of bit positions specified in `shift_val`. The 32 bits of data at `**dest_addr` are read, ORed with the shifted source and the result is written into `**dest_addr`. Then `*src_addr` and `*dest_addr` are incremented by two to point to the next shorts to be operated on (`(byte *)` incrementing).

NOTE: `shift_val` must be in the range of 0 to 15. Compilation with the `-aa` option (`/CHECK=PARAMETER` on VMS) will generate code to check in run-time for this limitation.

The `_bitwt` function is useful for implementing a high-speed "source OR destination" BITBLT operation, when the source data is aligned such that it does not need masking The implementation consists of a simple loop containing the `_bitwt` function and `add` instructions that adjust the source and destination warps.

8.3.4 _movmp (Move Multiple Pattern)

PROTOTYPE

```
void  _movmpb(void *dest_addr,
              int incr,
              unsigned count,
              unsigned char pattern);

void  _movmpw(void *dest_addr,
              int incr,
              unsigned count,
              unsigned short pattern);

void  _movmpd(void *dest_addr,
              int incr,
              unsigned count,
              unsigned int pattern);
```

DESCRIPTION

The _movmpb function copies count times the byte specified by the pattern function to dest_addr. Each count is spaced incr bytes from the previous one. Only the low order byte of pattern will be copied, if a pattern parameter larger than a byte is passed.

Similarly, the _movmpw function copies the word pattern. The incr is in units of bytes. If a double-word pattern parameter is passed, only its low order word will be copied. If a byte pattern parameter is passed, it will be zero-extended.

_movmpd copies the double-word pattern. If a byte or word pattern parameter is passed, it will be zero-extended.

This function is useful for quickly clearing large memory blocks. For example, in printer applications a page image can be cleared prior t o drawing the next page. The _movmp function can also be used for drawing horizontal lines and for creating simple patterns.

8.3.5 _sbits (Set Bit String)

PROTOTYPE

```
int _sbits(void *dest_addr,
           int bit_offset,
           unsigned run_length,
           unsigned *lookup_table);
```

DESCRIPTION

The `_sbits` function operates on `run_length` consecutive bits, starting with the bit at `bit_offset` from the byte at `*dest_addr`. `run_length` must be in the range 0 to 25. An OR operation is performed between these bits and a double-word from `lookup_table`. The double-word used is

```
lookup_table[run_length + 32*(bit_offset % 8)]
```

The function returns zero if `run_length` is 25 or less, and returns one if `run_length` is 26 or greater. Calling the functions as a procedure, such as

```
(void)_sbits(...
```

will prevent the compiler from producing the code for the return value.

The `lookup_table` parameter is used to lookup any table. The `Sbits` macro is an interface to the most common "black" lookup table (all bits in the run will be set). This lookup table is provided in the `libcg16.a` archive. The prototype for the `Sbits` macro is:

```
#define _Sbits(addr,offset,length)  \
        _sbits((addr),(offset),(length),_sbits_tbl)

extern char  _sbits_tbl[];
```

8.3.6 _sbitps (Set Bit Perpendicular String)

PROTOTYPE

```
void  _sbitps(void *src_addr,
              int *bit_offset,
              int run_length,
              int dest_warp);
```

DESCRIPTION

The `_sbitps` function sets `run_length` bits, starting at the bit at `*bit_offset` from the byte at `*src_addr`. The set bits are separated by `dest_warp` bits.

The function can be used to draw vertical lines by passing `dest_warp` equal to the image warp. Forty-five degree lines are drawn by `dest_warp` equal to the image warp plus or minus one. Other applications include expansion and/or rotation of images (in conjunction with the `_tbits` function) and filling.

8.3.7 _tbits (Test Bit String)

PROTOTYPE

```
int _tbits0(void *src_addr,          /* count a series of zeros */
            int *bit_offset,
            int *run_length,
            int max_run_length,
            int max_bit_offset,
            unsigned int *Fflag);

int _tbits1(void *src_addr,          /* count a series of ones */
            int *bit_offset,
            int *run_length,
            int max_run_length,
            int max_bit_offset,
            unsigned int *Fflag);
```

DESCRIPTION

The `_tbits0` function counts the number of consecutive clear bits, starting from the bit at `*bit_offset` from the byte at `src_addr`. Counting will terminate either at the first set bit, or if `max_run_length` or `max_bit_offset` bits were tested before a set bit was found. The number of bits scanned will be placed in `*run_length`, and `*bit_offset` will be the offset upon termination.

Similarly, `_tbits1` counts the number of consecutive set bits.

Both functions return the value of the PSR L flag. Calling the functions as a procedure, such as

```
(void)_tbits1(...
```

will prevent the compiler from producing the code for the return value.

The value of the PSR F flag is placed in the Fflag value. Calling the functions with the macro `IGNORE_PARAM` (defined in the `cg16.h`) as the actual Fflag parameter will prevent the compiler from producing the code for assigning the value of the PSR F flag.

Table 8-1. Effect of tbits on PSR L and F flags

CONDITION	L	F	NOTES
*run_length < max_run_length bit found	1	F	F reflects last bit tested
*run_length >= max_run_length bit not found	1	F	F reflects last bit tested
*bit_offset >= max_bit_offset bit not found	0	F	F reflects last bit tested
*bit_offset >= max_bit_offset on entry	0	0/1	0 for _tbits0 1 for _tbits1

8.4 NS32GX320 Intrinsic Functions

The NS32GX320 high-performance 32-bit microprocessor combines the full instruction set of the *Series 32000* family with the following on-chip integrated features: instruction and data caches, a 2-channel DMA controller, a 15-level interrupt control unit (ICU) and three 16-bit timers. In addition, Digital Signal Processing is supported by four special instructions:

`mulwd`	Multiply Word to Double
`cmuld`	Complex Multiply Double
`cmacd`	Complex Multiply and Accumulate Double
`mactd`	Multiply and Accumulate Twice Double

These instructions are accessible from the GNX C compiler using the special NS32GX320 intrinsic routines.

All prototype definitions given in this section are supplied as part of the GNX C package in the file `gx320.h`.

8.4.1 NS32GX320 typedefs

WCOMPLEX and DCOMPLEX are typedefs, defined for cmuld and cmacd. They designate data types for fixed-point complex arithmetic.

```
typedef struct WCOMPLEX {
        short re;
        short im;
} WCOMPLEX;

typedef struct DCOMPLEX {
        long re;
        long im;
} DCOMPLEX;
```

SHORT2 is a typedef, defined for mactd.

```
typedef struct SHORT2 { /* for mactd */
        short s1;
        short s2;
} SHORT2;
```

8.4.2 _mulwd (Multiply Word to Double)

PROTOTYPE

```
int   _mulwd(short src1, int* src2);
```

DESCRIPTION

The `_mulwd` function returns the integer multiplication of `src1` operand, by the lower 16 bits of `*src2`.

8.4.3 _cmuld (Complex Multiply Double)

PROTOTYPE

```
void _cmuld(WCOMPLEX src1,
            WCOMPLEX src2,
            DCOMPLEX *result);
```

DESCRIPTION

The `_cmuld` function assigns the complex multiplication of the source parameters (`*src1` and `src2`) to `*result`. For example

```
result->re = src1.re*src2.re - src1.im*src2.im
result->re = src1.re*src2.im + src1.im*src2.re
```

8.4.4 _cmacd (Complex Multiply and Accumulate Double)

PROTOTYPE

```
void _cmacd(DCOMPLEX *accum,
            WCOMPLEX src1,
            WCOMPLEX src2);
```

DESCRIPTION

The `_cmacd` function accumulates the complex multiplication of the source parameters (`src1` and `src2`) into `*result`. For example

```
accum->re = accum->re + src1.re*src2.re - src1.im*src2.im
accum->re = accum->im + src1.re*src2.im + src1.im*src2.re
```

8.4.5 _mactd (Multiply and Accumulate Twice Double)

PROTOTYPE

```
void _mactd(int *accum,
            SHORT2 src1,
            SHORT2 src2);
```

DESCRIPTION

The `_mactd` function accumulates the result of the two multiplications of `src1.s1*src2.s1` and `src1.s2*src2.s2` into `*result`. For example

```
accum = accum + src1.s1*src2.s1 + src1.s2*src2.s2
```

Appendix A

SERIES 32000 STANDARD CALLING CONVENTIONS

A.1 INTRODUCTION

The main goal of standard calling conventions is to enable the routines of one program to communicate with different modules, even when written in multiple programming languages. The *Series 32000* standard calling conventions support various special language features (such as the ability to pass a variable number of arguments, which is allowed in C), by using the different calling mechanisms of the *Series 32000* architecture. These conventions are employed only to call "externally visible" routines. Calls to internal routines may employ even faster calling sequences by passing arguments in registers, for instance.

Basically, the calling sequence pushes arguments on top of the stack, executes a call instruction, and then pops the stack, using the fewest possible instructions to execute at the maximum speed. The following sections discuss the various aspects of the *Series 32000* standard calling conventions.

A.2 CALLING CONVENTION ELEMENTS

Elements of the standard calling sequence are as follows:

- **The Argument Stack**
  Arguments are pushed on the stack from right to left; therefore, the leftmost argument is pushed last. Consequently, the leftmost arguments are always at the same offset from the frame pointer, regardless of how many arguments are actually passed. This allows functions with a variable number of arguments to be used.

  NOTE: This does not imply that the actual parameters are always evaluated from right to left. Programs cannot rely on the order of parameter evaluation.

  The run-time stack must be aligned to a full double-word boundary. Argument lists always use a whole number of double-words; pointer and integer values use a double-word (by extension, if necessary), floating-point values use eight bytes and are represented as `long` values;

  structures/unions use a multiple of double-words.

NOTE: Stack alignment is maintained by all GNX — Version 4 compilers through aligned allocation and de-allocation of local variables. Interrupt routines and other assembly-written interface routines are advised to maintain this double-word alignment.

The caller routine must pop the arguments off the stack upon return from the called routine.

NOTE: The compiler uses a more efficient organization of the stack frame if the `-OF` (`FIXED_FRAME` in VMS) optimization is enabled. In that case, programs should not rely on the organization of the stack frame.

- **Saving Registers**

  General registers R0, R1, and R2 and floating registers F0, F1, F2, and F3 are temporary or scratch registers whose values may be changed by a called routine. Also included in this list of scratch registers is the long register L1 of the NS32181/NS32381/NS32580 FPU. It is not necessary to save these registers on procedure entry or restore them before exit. If the other registers (R3 through R7, F4 through F7, and L3 through L7 of the NS32181/NS32381/NS32580) are used, their values should be saved (onto the stack or in other memory locations) by the called routine immediately upon procedure entry and restored just before executing the return instruction. This should be performed because the caller routine may rely on the values in these registers not changing.

  NOTE: Interrupt and trap service routines are required to save/restore all registers that they use. If the service routine calls another routine it must save scratch registers as well.

- **Returned Value**

  An integer or a pointer value that returns from a function, returns in (part of) register R0.

  Floating-point values return in floating point registers: A `float` value is returned in register F0. A `double` value is returned in register pair F0-F1.

  If a function returns a structure or union, the calling function passes an additional argument at the beginning of the argument list. This argument points to where the called function returns the structure. The called function copies the structure into the specified location just before returning from the function. Note that functions that return such types must be correctly declared as such, even if the return value is ignored. For details see Chapter 4.

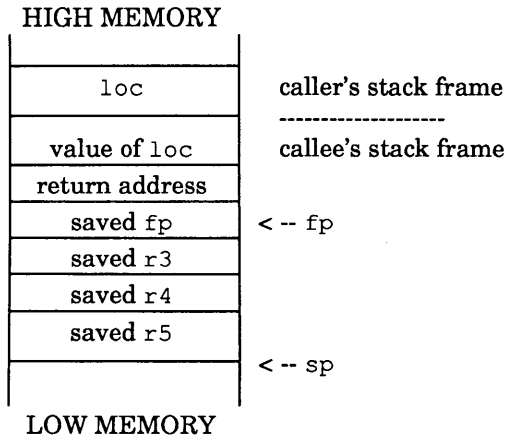
Example:

```
int iglob;
m()
{
     int loc;
     a = ifunc(loc);
}

ifunc(p1)
int p1;
{
     int i, j, k;
     j = 0;
     for (i = 1; i <= p1; i++)
         j = j + f(i);
     return(j);
}
```

The compiler may generate the following code:

```
_m:
        enter   [],4            # Allocate local variable
        movd    -4(fp),tos      # Push argument
        bsr     _ifunc
        adjspb  $(-4)           # Pop argument off stack
        movd    r0,_iglob       # Save return value
        exit    []
        ret     $(0)

_ifunc:
        enter   [r3,r4,r5],0    # Save safe registers
        movd    8(fp),r5        # Load argument to temp register
        movqd   $(0),r4         # Initialize j
        cmpqd   $(1),r5
        bgt     .LL1
        movqd   $(1),r3         # Initialize i
.LL2:
        movd    r3,tos          # Push argument
        bsr     _f
        adjspb  $(-4)           # Pop argument off stack
        addd    r0,r4           # Add return value to j
        addqd   $(1),r3         # Increment i
        cmpd    r3,r5
        ble     .LL2
.LL1:
        movd    r4,r0           # Return value
        exit    [r3,r4,r5]      # Restore safe registers
        ret     $(0)
```

After the enter instruction is executed by `ifunc()`, the stack will look like this:



Appendix B

MIXED-LANGUAGE PROGRAMMING

B.1 INTRODUCTION

Mixed-language programs are frequently used for a couple of reasons. First, one language may be more convenient than another for certain tasks. Second, code sections, already written in another language (*e.g.*, an already existing library function), can be reused by simply calling them.

A programmer who wishes to mix several programming languages needs to be aware of subtle differences between the compilation of the various languages. The following sections describe the issues the user needs to be aware of when writing mixed-language programs and then compiling and linking such programs successfully.

B.1.1 Writing Mixed-Language Programs

The mixed-language programmer should be aware of the following topics:

- **Name Sharing** – Potential conflicts including permitted name-lengths, legal characters in identifiers, compiler case sensitivity, and high-level to assembly-level name transformations.
- **Calling Conventions** – The way parameters are passed to functions, which registers must be saved, and how values are returned from functions. See Appendix A for details.
- **Declaration Conventions** – The demands that different languages impose when referring to an outside symbol (be it a function or a variable) that is not defined locally in the referring source file. Note that this is also true of references to an outside symbol that is not in the same language as that of the referring source file.

To help the programmer avoid these potential problems, a set of rules for writing mixed-language programs has been devised. Each rule consists of a short mnemonic name (for easy reference), the audience of interest for the rule, and a brief description of the rule.

Figure B-1 summarizes all of the rules in the context of each possible cross-language pair.

	C	Pascal	FORTRAN 77	*Series 32000* Assembly
Series 32000 Assembly	'_' prefix	'_' prefix include ext case sensitivity	'_' prefix '_' suffix ref args case sensitivity	
FORTRAN 77	'_' suffix ref args case sensitivity	'_' suffix include ext ref args		'_' prefix '_' suffix ref args case sensitivity
Pascal	include ext case sensitivity		'_' suffix include ext ref args	'_' prefix include ext case sensitivity
C		include ext case sensitivity	'_' suffix ref args case sensitivity	'_' prefix

Figure B-1. Cross-Language Pairs

RULE 1 **case sensitivity**

This rule is of interest to every programmer who mixes programming languages.

C and *Series 32000* assembly are case sensitive while FORTRAN 77 and Pascal are not. Programmers who share identifiers between these two groups of languages must take this into account. To avoid problems with case sensitivity, the programmer can:

1. Take care to use case-identical identifiers in all sources and compile FORTRAN 77 and Pascal sources using the case-sensitive option (`/CASE_SENSITIVE` on VMS, `-d` on UNIX).
2. Use only lower-case letters for identifiers which are shared with FORTRAN 77 or Pascal since the FORTRAN 77 and Pascal compilers fold all identifiers to lower-case if not given the case-sensitive option.

RULE 2 **'_' prefix**

This rule is of interest to those who mix high-level languages with assembly code.

All compilers map high-level identifier names into assembly symbols by prepending these names with an underscore. This ensures that user-defined names are never identical to assembly reserved words. For example, a high-level symbol `NAME`, which can be a function name, a procedure name, or a global variable name, generates the assembly symbol `_NAME`.

Assembly written code which refers to a name defined in any high-level language should, therefore, prepend an underscore to the high-level name. Stated from a high-level language user viewpoint, assembly symbols are not accessible from high-level code unless they start with an underscore.

RULE 3 **'_' suffix**

This rule is of interest to those who mix FORTRAN 77 with C, Pascal, or assembly code.

The FORTRAN 77 compiler appends an underscore to each high-level identifier name (in addition to the action described in RULE 2). The reason for an appended underscore is to avoid clashes with standard-library functions that are considered part of the language, *e.g.*, the FORTRAN 77 WRITE instruction. For example, a FORTRAN 77 identifier `NAME` is mapped into the assembly symbol `_NAME_`.

Therefore, a C, Pascal, or assembly program that refers to an FORTRAN 77 identifier name should append an underscore to that name. Stated from an FORTRAN 77 user viewpoint, it is impossible to refer to an existing C, Pascal, or assembly symbol from FORTRAN 77 unless the symbol terminates with an underscore.

RULE 4 **ref args**

This rule is of interest to those who mix FORTRAN 77 with other languages.

Any language which passes an argument to a FORTRAN 77 routine must pass its address. This is because a FORTRAN 77 argument is always passed by reference, *i.e*, a routine written in FORTRAN 77 always expects addresses as arguments.

Routines not written in FORTRAN 77 cannot be called from an FORTRAN 77 program if the called routines expect any of their arguments to be passed by value. Only routines which expect all their arguments to be passed by reference can be called from FORTRAN 77.

The Pascal program must declare all FORTRAN 77 routine arguments using `var`. C programs must prepend the address operator & to FORTRAN 77 routine arguments in the call. The C or Pascal programmer

who wants to pass an unaddressable expression (such as a constant) to a FORTRAN 77 routine, must assign the expression to a variable and pass the variable, by reference, as the argument.

RULE 5 **include ext**

This rule is of interest to Pascal programmers who want to share variables between different source files which may or may not be written in Pascal.

Pascal sources which share global variables or routines must make these variables known to separately compiled modules. This is done by the `import` and `export` attributes, or by inclusion of a `.h` file which contains the variables or routines. ERROR in line number 290 incorrect number of fields line is: .if!C'Pascal'See The *Series 32000 GNX–Version 4 Pascal Optimizing Compiler Reference Manual. ERROR line292contains a . No matching*

In addition to these rules, a few points should be noted. First, GNX — Version 4 FORTRAN 77 allows identifiers longer than the six character maximum of traditional FORTRAN compilers. Second, the family of GNX — Version 4 Compilers allows the use of underscores in identifiers. Both of these enhancements simplify name sharing.

Importing Routines and Variables

The general conventions of all languages must be kept in mixed-language programs. In particular, externals must be declared in those program sections which import them. The following are examples of declarations of external (imported) functions/procedures and external (imported) variables in each language. The examples are in the form:

caller language: *external (imported) functions / procedures*
or
external (imported) variables

Example:

C:
```
extern int  func_();
```
or
```
extern int  var_name_;
```

Note that the strict reference C model (ANSI C standard) is assumed. If the model is relaxed, then the external declarations are not mandatory.

FORTRAN 77:
```
INTEGER  func
```
or
```
COMMON  /var_name/ local_name
```

Pascal `.h` file:
```
function func_ : integer ; external;
```

```
                    procedure proc_ ; external;
                      or
                    #include "var_def.h"
```

where the file `var_def.h` contains:

```
  var var_name_ : integer;
```

Pascal `.h` file: import/export

```
import function func_ : integer;
var export i: integer;
```

Series 32000: assembly

```
.globl _func_
  or
.globl _var_name_
```

B.1.2 Compiling Mixed-Language Programs

After writing different program parts in different languages, keeping in mind the rules previously mentioned, the mixed-language programmer must still link and load these parts to make them run successfully. The following three points should be mentioned in conjunction with the successful linking and loading of programs.

- External library (standard or nonstandard) routines must be bound with the user-written code that calls them.
- Initialization code which arranges to pass program parameters to the main program and then calls the main program, sometimes has to be bound with user-written code.
- The entry point of the code, *i.e.*, the location where the program starts executing, should be determined.

Libraries:
The following table (Table B-1) lists libraries associated with each compiler. When programming with mixed languages, the libraries associated with the languages used must be bound with the program during the link phase of compilation.

Initialization code and Entry-points:
Normally, the entry point of the final executable file is called `start`. The code that follows this entry-point is an initialization code that prepares the run-time environment and arranges parameters to be passed to the user-written main program. The initialization object file which is linked by default is called `crt0.o`. The `crt0.o` file always calls `_main`.

The assembly-symbol that starts the user main program is `_main` (the underscore is prepended by the C compiler) in the case of C programs and `_MAIN__` in Pascal or FORTRAN 77 programs.

Table B-1. Compilers and their Associated Libraries

COMPILER (DRIVER) NAME	LIBRARIES
cc (cross nmcc)	libc
f77 (cross nf77)	libF77, libI77, libm, libc
pc (cross nmpc)	libpas, libm, libc

Note that the last two compilers completely ignore the user's main program name. Therefore, in C, the user-written code is called directly from `crt0.o`. In Pascal and FORTRAN 77, `_main` is located in the respective standard library which performs additional initializations before calling the user entry-point `_MAIN__`.

B.1.3 Compilation on UNIX Operating Systems

National Semiconductor's GNX tools (assembler, linker, *etc.*) on UNIX systems relieve a user's concern about external libraries, initialization code, and entry-points. This is due to the coherence and consistency of the GNX — Version 4 Compilers and their integration through the use of a common driver.

When using a GNX — Version 4 Compiler on a UNIX system, the user does not directly call the compiler front end, optimizer, code generator, assembler or linker. Instead, the calls are indirectly made through the driver program.

The driver program accepts a variable number of filename arguments and options and knows how to identify language-specific options. The driver also identifies the languages in which its filename arguments are written by the names of these arguments. Therefore, the driver can arrange to compile and bind the programs with the needed libraries in order to run the program successfully.

The driver program used by C, Pascal and FORTRAN 77 programmers is exactly the same program on UNIX systems, named differently for each language. The respective driver names are `cc`, `pc` and `f77` (`nmcc`, `nmpc` and `nf77` for cross-support).

The driver program looks at its own name in order to determine the libraries that are to be bound with the program. In addition, the driver links additional libraries according to the name extensions of any of its filename arguments. For instance, `cc` also links `libm` and `libpas` when one of the filename arguments is a Pascal source (recognized by the `.p`, `.pas`, `.P` or `.PAS` extensions).

The -v (/VERBOSE on VMS) option of the driver verbosely outputs all driver actions. With this option the interested user can track problems that might arise (such as undefined symbols from the linker).

As mentioned in the previous section, different languages use different initialization codes that reside in language-specific standard libraries. It is necessary that the correct language initialization code be linked with a mixed-language program. The driver program helps do this, but it needs to know in which language the main program is written.

To ensure that the correct initialization code is linked with a mixed-language program, the user should call the driver that corresponds to the language of the main program module within the mixed-language program.

For example, suppose there are four source modules written in four different languages (c_utils.c written in C, f_utils.f written in FORTRAN 77, p_utils.p written in Pascal, and s_utils.s written in assembly language), and there is a fifth module that has already been compiled separately (obj.o, an object module). Assuming there is a main program written in FORTRAN 77, the f77 driver should be used.

```
f77 main.f c_utils.c f_utils.f p_utils.p s_utils.s obj.o
```

If the main program is written in C, cc is used, and so on.

B.1.4 Compilation on VMS Operating Systems

Under the GNX tools on VMS systems, the linking phase is separate from the compilation phase; therefore, it demands separate actions from the user.

The interested user should refer to the language tools manuals (assembler, linker, *etc.*) for a complete description of how to use them on VMS systems.

B.2 COMPILING THE MIXED-LANGUAGE EXAMPLE

The example listed in Section B.3 consists of a number of program modules written in languages different from the main program which is written in C.

B.2.1 Compiling the Example on a UNIX System

To compile the program modules on a UNIX system, type the command:

```
nmcc c_main.c\

     c_fun.c f77_fun.f pas_fun.p asm_fun.s
```

This assumes that all the program modules are in the same directory. If the program compiles and links successfully, the result is an executable file that, when run, prints

the line "`Passed OK !!!`".

B.2.2 Compiling the Example on a VMS System

To compile the modules on a VMS system, type the following commands:

```
nmcc c_main.c
nmcc c_fun.c
nf77 f77_fun.f
nmpc pas_fun.p
nasm asm_fun.s
```

After successful linking, the result is an executable file that, when run, prints the line "`Passed OK !!!`".

B.3 PROGRAM MODULE LISTINGS

The different program modules are listed in this section.

c_main.c

```
/*-----------------------------------------------------------------------
 *   Example of a C program which communicates with C, Pascal,
 *   Fortran 77, and Assembly external functions, via direct
 *   calls as well as via a global variable.
 *   Parameter passing by reference is accomplished by passing the
 *   addresses of the characters variables 'a', 'b', 'c', 'd' and 'e'.
 *-----------------------------------------------------------------------*/

char str_[] = "Passed OK !!!0;       global ('exported') string

main() {
    char a, b, c, d, e;
    int  three = 3;         FORTRAN must get its parameters by reference
                            So we put this constant into a variable ...


    if (c_func   (&a, 0)       &&       in C arrays start with 0
        pas_func (&b, 2)       &&       in Pascal they start at 1
        f77_func_(&c, &three) &&        in f77, at 1
        asm_func (&e, 4))               in assembly, at 0

            printf("%c%c%c%c%c%s", a, b, c, d, e, str_+5);
               Should print "Passed OK !!!"
}

dummy()
{
}
```

c_fun.c

```
/*
 *  Declaration of the public character string 'str[]' and definition
 *  of the C function 'c_func()'.
 *  Note the appending of an underscore to the external symbol 'str_'
 *  which is shared with FORTRAN 77.
 */
extern char str_[];

int c_func(c_ptr, index)
char     *c_ptr;
int      index;
{
    *c_ptr = str_[index];
    return 1;
}
```

f77_fun.f

```
C
C   The FORTRAN 77 function:
C
C   All parameters are passed by reference
C   The COMMON statement aliases the external array 'str' as 'text'
C
      LOGICAL FUNCTION f77_func(c, index)
      CHARACTER  c
      INTEGER    index
      COMMON /str/ text
      CHARACTER text(15)
      c = text(index)
      f77_func = .TRUE.
      RETURN
      END
```

pas_fun.p

```
(*
 *  The Pascal function 'pas_func()'
 *)

(* 'str[]' character-array declaration *)
#include "str_pas.h";

(* make this function visible to outsiders ('export') *)
function pas_func(var c: char; index: integer): boolean; external;

function pas_func();
begin
    c := str_[index];
    pas_func := TRUE;
end;
```

str_pas.h

```
(* 'str[]' character-array declaration for Pascal *)
var
    str_ : packed array [1..15] of char;
```

```
#
#    The 32000 Assembly Language Function 'asm_func'
#
#       The function includes an artificial use of r7, to demonstrate the
#       need to save it upon entry and restore upon exit, as opposed to
#       r0, r1 and r2; f0, f1, f2 and f3 which can be used freely without
#       saving or restoring.  This is according to the Series 32000
#       standard calling convention.
#       The function return value is placed in r0, also according to the
#       standard calling convention.
#
       .globl    _str_       # Import the global str[] array.
       .globl    _asm_func   # Export (make visible) the assembly function.
       .align 4

_asm_func:
       enter     [r7],0                          # Set frame, show saving of r7
       movb      _str_+0(12(fp)),0(8(fp))        # argument_1 ← str[argument_2]
       movqd     $(1),r7                         # artificial use of r7
       movd      r7,r0                           # return_value ← TRUE
       exit      [r7]                            # Unwind frame, restore r7
       ret       $(0)                            # Return to caller
```

Appendix C

ERROR DIAGNOSTICS

C.1 INTRODUCTION

The GNX C compiler has a superior error handling mechanism. In most cases, the compiler continues to compile when an error is found. An error message is displayed, providing information on the type of error, the source filename, the line number location of the error. Generally, the compiler attempts to minimize the effects of errors on compilation.

C.2 ERROR MESSAGES

Errors are divided into six categories:

1. Limitation Errors
2. System Errors
3. Severe Errors
4. Syntax Errors
5. Caution Errors
6. Warnings

C.2.1 Error Messages Format

The general syntax of an error message is

filename, line *m*:*c*) [*category*]: *message*

Where:

filename	is the source file name.
m	is the line number location of the error.
c	is a lower case letter used to mark the error position on the source line.
category	is the error category.

The error message is followed by an echo of the source line. The errors are marked with the appropriate lower case letter corresponding to the error message as displayed in the syntax.

Example:

```
"stam.c", line 3: a) [severe]: "j" undefined
                  b) [severe]: illegal indirection
                  c) [syntax]: ')' may be missing before ';'
for ( j = 1; *i != 0; ;
------a---------b-----c-----------------------------------------------
```

C.2.2 System Errors

System errors are related to the operating system or the environment in which the compiler runs. For example:

```
[system]: Can't open file filename
[system]: Ran out of memory
```

C.2.3 Limitation Errors

Limitation errors are caused by exceeding compiler limitations. Generally a limitation error causes the suspension of code generation. In these cases the limitation message includes "no object file produced". However, in some cases limitation messages are warnings, and code generation continues.

The following is a complete list of limitation errors:

1. `[Limitation]: array size too large; no object file produced`

   This error message is produced if an array size exceeds the maximal number which can be represented in 29 bits `(536870911)`.

2. `[Limitation]: structure too large; no object file produced`

   This error message is produced if a structure size exceeds the maximal number which can be represented in 29 bits `(536870911)`.

3. `[Limitation]: array dimension too large; no object file produced`

   This error message is produced if a number greater than the maximal number which can be represented in 29 bits `(536870911)` is used for an array's dimension.

4. `[Limitation]: cumulative size of structure members is too large; no object file produced`

This error message is produced for structures for which the cumulative size of the structure's members exceeds the maximal number which can be represented in 29 bits (536870911).

5. `[Limitation]: not enough space on frame for compiler-produced temporaries; no object file produced`

   This error message is produced when the cumulative size of the local variables and the temporary variables created by the compiler for computations exceeds the maximal number which can be represented in 29 bits (536870911).

6. `[Limitation]: cumulative size of local variables is too large; no object file produced`

   This error message is produced when the cumulative size of the local variables and the temporary variables created by the compiler for computations exceeds the maximal number which can be represented in 29 bits (536870911).

7. `[Limitation]: size too large for symbolic information, may confuse the debugger`

   This error message is produced for objects (structures/unions or arrays) who's size is greater than the maximal number which can be represented in 16 bits (65535). Code continues to be generated, however the operation of the debugger may be affected.

8. `[Limitation]: array dimension too large for symbolic information; may confuse the debugger`

   This error message is produced if a number greater than the maximal number which can be represented in 16 bits (65535) is used for an array's dimension. Code continues to be generated, however the operation of the debugger may be affected.

C.2.4 Syntax Errors

When the compiler detects a syntactic error, an attempt is made to fix the error by internally changing the input to a syntactically legal phrase. This is done in order to continue compilation and produce a maximum number of useful diagnostics; it should not be used as a means to correct your program.

An error is fixed by the deletion, insertion or replacement of a token, or by the skipping of a language phrase. The appropriate action is selected by the compiler based on the context of the source code, semantic information and a set of heuristics. Once a successful change in the source code is introduced, a syntax error is reported to indicate the change. If the change does not result in a legal syntactic phrase, the compiler skips to a point where the text is synchronized with the language syntax.

NOTE: In some cases the change introduced by the compiler may not be appropriate for your source code.

Examples:

1. Token deletion

```
"filename", line 1: a) [syntax]: Unexpected ','
int i,,j;
------a-----------------------------------------------------
```

The unexpected token ',' was deleted internally in attempt to continue compilation.

2. Token insertion

```
"filename", line 3: a) [syntax]: ';' may be missing before '}'
i= i }
-----a-------------------------------------------------------
```

The token ';' was inserted before the token '}' in an attempt to continue compilation.

3. Token replacement

```
"filename", line 4: a) [syntax]: ']' unexpected, ';' may be
                         more appropriate
i ++]
----a---------------------------------------------------------
```

The unexpected ']' token was replaced with a ';' in an attempt to continue compilation.

4. Skipping a language phrase

```
"stam.c", line 6: a) [syntax]: Unexpected "p", more errors may
                     follow... skipping until ';' on line 13
xxxx ;p
------a-----------------------------------------------------
```

Indicates the beginning of the skipped phrase.

```
"stam.c", line 13: a) [syntax]: ... skipped until here (from
                      line 6)
foo(){};
-------a-------------------------------------------------------
```

Indicates the end of the skipped phrase.

NOTE: Additional errors occasionally result from the fact that part of the program was skipped by the compiler.

C.2.5 Severe Errors

Severe errors are caused when the compiler detects semantic violations of the language rules, where the programmer's intention is not clear to the compiler. In this case, an error message is displayed and code generation is terminated.
For example:

```
"stam.c", line 10: a) [severe]: traditional and prototype
                      parameters cannot be mixed
foo(int,a){};
--------a-----------------------------------------------------
```

C.2.6 Caution Errors

Caution errors are issued for erroneous language constructs which the compiler either "thinks" may be on purpose or "guesses" the programmers intention. An error message is displayed and generation of code continues. If the programmer is satisfied with the compiler's action, the program produced may be run.
For example, if `i` was declared as `volatile int i;`:

```
"stam.c", line 2: a) [caution]: volatile pointer mismatch
int *p = &i;
----------a----------------------------------------------
```

C.2.7 Warnings

Warning messages are issued for input which conforms to the language, but is deemed to be inappropriate by the compiler in the context found. An error message is displayed and generation of code continues.
For example:

```
"notreached.c", line 8: a) [warning]: statement not reached
              i++;
----------------a-----------------------------------------
```

Warning messages can be disabled by the `-w` compiler option on UNIX systems (`/NOWARNING` on VMS).

Appendix D

COMPILER OPTIONS

D.1 INTRODUCTION

This appendix contains tables for quick reference to the GNX—Version 4 C compiler options. These tables list:

- Options to the compiler on UNIX systems
- Options to the compiler on VMS systems
- Options to the compiler that pass to the C preprocessor on UNIX systems
- Options to the compiler that pass to the C preprocessor on VMS systems
- Options to the compiler that pass to the linker

(Options that pass to the linker are relevant only for UNIX systems.)

Table D-1. UNIX Operating System Options
Sheet 1 of 2

OPTION	FUNCTION
-A	Allocate variables as standard.
-a*flags*	Generate runtime checks.
-B	Add code for profile information gathering.
-c	Suppress loading, force production of object file in *file*.o.
-d	This option is useful only when compiling Pascal and FORTRAN 77 programs.
-F*flags*	Set optimization flags but do not call optimizer.
-f	Use floating-point emulation library.
-g	Prepare symbolic debug information for debugger.
-J*width*	Force alignment boundary within structs to *width*.
-KC*cpu* -KF*fpu* -KB*bus*	Set target CPU. Set target FPU. Set target buswidth.
-l*lib*	Use *lib* as a program library.
-m	Use m4 as the preprocessor for FORTRAN 77 and assembly files.
-n	Put C source lines as comments into assembly output file.
-N[*flags*]*nnn*	This option is useful when compiling FORTRAN 77 programs.

Table D-1. UNIX Operating System Options
Sheet 2 of 2

OPTION	FUNCTION
-O*flags*	Perform optimizations according to *flags*.
-X	Generate code that conforms to the *Series 32000* architectural feature of modularity.
-p	Prepare profiling information for profiling.
-Q	Compile only, verify for syntax errors.
-R	Put all literal strings in read-only memory.
-S	Do not assemble, leave assembly in *file*.s.
-T	This option is only useful when compiling FORTRAN 77 programs.
-v	Verbose: list the subprograms as actually called by the driver.
-vn	List the subprograms to be called, but do not actually execute them.
-W*x*,*options*	Pass *options* to compiler phase *x*. *x* can be `p` (C preprocessor), `a` (assembler), or `l` (linker).
-w	Suppress warnings.
-w66	This option is only useful when compiling FORTRAN programs.
-Z*c*	Use an alternate library.

Table D-2. VMS Operating System Options
Sheet 1 of 2

OPTION	FUNCTION
/[NO]OBJECT[=*filename*]	[do not] Generate an object file during the compilation process.
/[NO]OPTIMIZE[=(*flags*[...])]	[do not] Perform optimizations [according to *flags*].
/CHECK[=(*option*[,...])]	Generate run-time checks.
/[NO]DEBUG	[do not] Prepare symbolic debug information for debugger.
/[NO]GATHER	[do not] Add code for profile information gathering.
/[NO]PROFILE	[do not] Prepare profiling information for profiling.
/[NO]ASM[=*filename*]	[do not] Generate an assembler file during the compilation process.
/[NO]ANNOTATE	[do not] Put C source lines as comments into assembly output file.
/[NO]ROM_STRINGS	Put all literal strings in read-only memory.
/ALIGN[=*width*]	Force alignment boundary within structs to *width*.
/[NO]WARNING	[do not] Output warning diagnostics.
/[NO]STANDARD	Allocate variables as standard.

Table D-2. VMS Operating System Options
Sheet 2 of 2

OPTION	FUNCTION
/[NO]PRE_PROCESSOR	[do not]Run the source code through the cpp preprocessor.
/[NO]VERBOSE	[do not] List the compiler subprograms called by the driver.
/[NO]VN	[do not] List the subprograms to be called, but do not actually call them.
/TARGET=(CPU=*cpu*) /TARGET=(FPU=*fpu*) /TARGET=(BUSWIDTH=*bus*)	Set target CPU. Set target FPU. Set target buswidth.
/[NO]MODULAR	Generate code that conforms to the *Series 32000* architectural feature of modularity.
/[NO]ERROR[=*filename*]	[do not] Generate an error log file during the compilation process.

Table D-3. Options Passed to the Preprocessor — UNIX Systems

OPTION	FUNCTION
-C	Prevent the macro preprocessor from removing comments.
-D*name=def* -D*name*	Define *name* to have the value *def.* Define *name* to have the value 1.
-E	Run only the preprocessor, send the result to *stdout.*
-I*dir*	Look for include files in *dir* after looking in the current directory.
-M	Generate makefile dependencies (cpp option).
-P	Run only the preprocessor, send the result to a preprocessed source. file.
-U*name*	Undefine *name.*

Table D-4. Options Passed to the Preprocessor — VMS Systems

OPTION	FUNCTION
/[NO]COMMENT	[do not] Prevent the preprocessor from removing comments.
/DEFINE=(*name*[=*def*][,...])	Define *name* to the preprocessor.
/[NO]EXPAND[=*filename*]	[do not] Generate a source file after preprocessing.
/INCLUDE=(*include_dir*[,...])	Look for include files in *include_dir* after looking for them in the current directory.
/UNDEFINE=(*name*[,...])	Undefine *name* to the preprocessor.

Table D-5. Options Recognized and Passed to the Linker

OPTION	FUNCTION
-e *epname*	Define *epname* as entry point.
-o *out*	Name the compilation output file *out*.
-r	Retain relocation.
-s	Strip.
-u *symname*	Undefine *symname* in symbol table.
-V	Print linker version information.
-x	Do not preserve local symbols in the symbol table.
-i	Intialize variables in runtime.

Appendix E

EMBEDDED PROGRAMMING HINTS

E.1 INTRODUCTION

The GNX C compiler provides certain features which allow for programming of embedded applications in C. These features help solve the following issues:

- full control over memory allocation - including RAM, ROM, stack space, trap and interrupt vectors, peripheral memory-mapped control registers.
- startup actions performed at system reset - including initializing stack pointers, configuration registers, peripheral control registers, and timers.
- initialization of RAM data variables - usually by copying from ROM or by zeroing.
- interrupt/trap handling

This appendix provides suggestions and examples for using the C compiler in embedded applications.

E.2 VOLATILE AND CONST

The `const` and `volatile` type qualifiers can be used in embedded applications to indicate ROM entities and memory mapped entities, respectively. A general overview of the semantics and use of these qualifiers is explained below. For further detail see Section E.2.5 and the ANSI C standard.

E.2.1 Const Type Qualifier

The value of an object (any lvalue expression) whose type includes the `const` qualifier cannot be modified. The `const` qualifier can be used for several purposes:

1. Constant strings can be made a part of the program code and placed into ROM.
2. Protecting variables from being changed. If during run-time an attempt is made to change a `const` variable, a trap will occur.

A non-volatile global or static object declared as `const`, will be allocated in read-only memory (the `.text` area) if it is initialized.

For example:

```
const int i = 137;        /* i is defined as const */
i = 17;                   /* this is illegal !!    */
i += 12;                  /* this is illegal !!    */
```

The `const` syntax allows for the declaration of both 'constant pointers' and 'pointers to constants'. For example:

```
const char * pcc;          /* pcc is defined as pointer to    */
                           /* const char                      */
char * const cpc;          /* cpc is defined as const pointer */
                           /* to char                         */
const char * const cpcc;   /* cpcc is defined as const        */
                           /* pointer to const char           */
```

The types `pointer to const object` and `const pointer to object`, as in the above example, have different meanings. The value of a `pointer to const object` can be modified; however the value of the pointed object can **not** be modified. In contrast, the value of a `const pointer to an object` can **not** be modified; however the value of the pointed object can be modified.

For example:

```
const char * pcc;    /* pcc is defined as pointer to */
                     /* const char                   */
.  .  .
.  .  .
.  .  .
pcc ++;                /* this is O.K. */
*pcc = 17;             /* this is an error */
```

E.2.2 Volatile Type qualifier

The value of an object (any lvalue expression) whose type includes the `volatile` qualifier can be used or changed by asynchronous events (such as I/O or interrupts). Such an object should not be subject to any optimization that will change or delay references to it.

By using the `volatile` qualifier, you can specify volatile objects. Therefore, full optimization is carried out on all other objects, including global variables and pointer dereferences.

For example, in the following code

```
volatile int i;
int j;
.  .  .
.  .  .
foo() {
    .  .  .
    .  .  .
    for (i=1 ; i<j; i++) {
        .  .  .
        .  .  .
    }
}
```

the compiler can put `j` in a register. But for `i` this optimization is not permitted.

The `volatile` syntax allows for the declaration of both 'volatile pointers' and 'pointers to volatiles'.

For example:

```
char * pc;              /* pc is defined as pointer to char  */
volatile char * pvc;    /* pvc is defined as pointer to */
                        /* volatile char                */
char * volatile vpc;    /* vpc is defined volatile pointer */
                        /* to char                      */
volatile char * volatile vpvc;
                        /* vpvc is defined as volatile */
                        /* pointer to volatile char    */
```

The types `pointer to volatile object` and `volatile pointer to object`, as in the above example, have different meanings. References to a `pointer to volatile object` can be optimized; however references to the pointed object can not be optimized. In contrast, references to a `volatile pointer to an object` can not be optimized; however references to the pointed object can be optimized.

Assignment of `pointer to volatile object` to `pointer to object` is permitted only if an explicit cast is used. Not using an explicit cast causes an error message.

For example:

```
pvc = pc;              /* O.K. */
pc = vpc;              /* illegal */
pc = (char *) vpc;     /* O.K. */
```

E.2.3 Memory Allocation

Memory allocation is performed by the operating system in native programming environments such as UNIX. However, embedded applications require the ability to control memory allocation. This is achieved by specifying in the linker directive file:

- the memory ranges of various program sections.
- the division of program sections into ROM and RAM.
- the sections to be copied from ROM to RAM at program startup.

A complete description of the linker directive file is provided in Chapter 3 of the *GNX Linker User's Guide*. Figure E-1 is an example of a simple linker definition file for defining two areas of memory.

```
MEMORY {
    ROM : origin=0x1000  length=0x2000
    RAM : origin=0x10000 length=0x80000
}

SECTIONS {
    .text INTO(ROM) : { *(.text) }
    .data INTO(RAM) : { *(.data) }
          . . .
}
```

Figure E-1. Example of Linker Directive File

E.2.4 Initialized C Variables

The C programming language allows compile-time initialization of global and static variables. In addition, uninitialized global and static variable are defined by the C language to have a zero value at program startup.

In native environment, initialization is handled by the compiler and the operating system. In cross environment, when loading the program with the GNX debugger, the debugger performs these initializations. However in embedded applications, all initialized data resides in ROM and must be explicitly copied to RAM at program startup. The GNX linker directive file and the GNX run-time library are used to automatically

initialize RAM variables.

Refer to the *GNX Linker User's Guide* for further details.

E.2.5 Programming Memory Mapped Devices

When writing code for the registers of memory mapped peripheral, correctly and efficient accessing these entities can be problematic. However, the GNX C compiler allows optimization of such code.

The `volatile` qualifier should be used to specify the memory mapped entities. This allows the optimizer to perform optimizations without changing or delaying references to these entities.

An example of the correct way to code memory mapped entities is:

```
#define ctrl_reg *((volatile short *)0xffe8)

foo()
{
   return ctrl_reg;
}
```

This will result in:

```
movxwd @(65512),r0
ret    0
```

NOTE: Do not define a global pointer variable, such as `volatile short *ctrl_reg = (short *) CTRL_REG;` for memory mapped entities. Dereferencing such a pointer, as in `*ctrl_reg`, will result in less efficient code.

E.3 ASM STATEMENTS

The `asm` keyword (see Section 3.3.2) provides for the unlimited insertion of assembly language statements into any position in the code. It is recommended to use this feature only for actions not codeable in standard C, such as processor register manipulation and `svc` calls on *Series 32000* development boards.

Extreme care should be taken especially when `asm` is used in conjunction with the optimizer. See Section 6.6.6 for details.

The following example is a routine that will change the CPU status to supervisor mode on a NS32CG16ED board:

```
change_to_supervisor()
{
    asm("movd  $9,r0");            /* svc call number 9 */
    asm("movd  $0x55555555,r1");   /* security code     */
    asm("svc");
}
```

The following example assumes that the `stack_pointer` symbol is defined in the linker directive file (see the *GNX Linker User's Guide* for more details) as

```
stack_pointer = 0x003ffff0.
```

The following code will load the stack pointer register with the above value:

```
asm("addr   stack_pointer, r0");
asm("lprd   sp, r0");
```

E.4 EXAMPLES OF PROGRAMMING WITH INTRINSIC FUNCTIONS

This section describes programming with intrinsic functions. More details can be found in Chapter 8.

E.4.1 NS32CG16 bit instructions

An example of a graphic application based on certain special NS32CG16 core bit operations is illustrated in this section. The image is represented by a bit-map 80 bits wide and 21 lines high. The picture is drawn by printing the bit-map in an ascii format in the following way:

1. A set bit in the bit-map is represented by the character '*'.
2. A clear bit is represented by a space .

It is important to include the proper header file, with intrinsic routines declarations, in your application. In this example, `cg16.h` is included.

The following definitions are used throughout the example:

```
#define PAGE_WIDTH_IN_BYTES    10
#define PAGE_WIDTH_IN_BITS     (PAGE_WIDTH_IN_BYTES  * 8)
#define PAGE_HEIGHT            21
#define Page(y,x)    (page + (y)*PAGE_WIDTH_IN_BYTES + (x))
```

The bitmap is kept in the following char array:

```
char page[PAGE_WIDTH_IN_BYTES * PAGE_HEIGHT];
```

A 'V' is drawn in the upper-left part of the image using the `_sbitps` intrinsic function:

```
draw_a_v(){
  int offset = 3;

   _sbitps(Page(2,0),&offset,7,PAGE_WIDTH_IN_BITS + 1);
   _sbitps(Page(2,0),&offset,8,-PAGE_WIDTH_IN_BITS + 1);
}
```

The resulting image is shown in Figure E-2

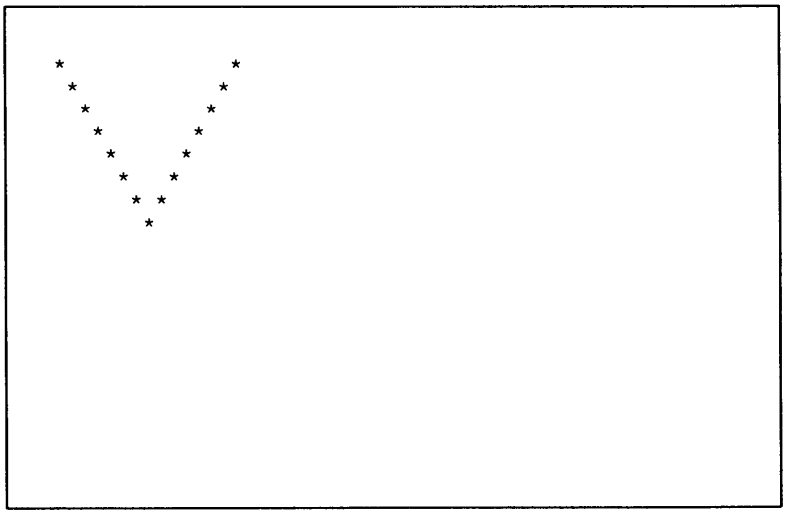


Figure E-2. The Image

A reversed image of 'V' figure is drawn by using the `_bbor_s` intrinsic function:

```
/*
 * Copy a bit-block (reversed) containing the V, to
 * the lower right corner of itself.
 * Mask off the areas to the right and left of the image,
 * to protect the image of the V drawn above.
 */
 _bbor_s(Page(1,0),Page(9,2),3,10,0x0fffe,0x0000f,
        PAGE_WIDTH_IN_BYTES-2,PAGE_WIDTH_IN_BYTES-2,2);
```

The resulting image is shown in Figure E-3.

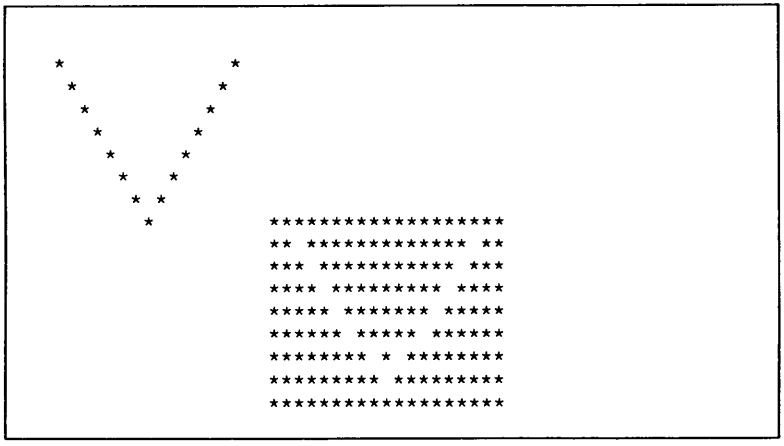


Figure E-3. The Image with the Reversed Shape

To print the image in the format of the example, the intrinsic routine `_tbit()` (a general 32000 instruction) is used:

```
/*-------------------------------------------------------
 *              dump_as_ascii()
 *
 * Displays the bitmap "page" as an ascii picture.
 * Each bit is displayed as a character:
 *          '*' if it is ON, ' ' (space) if it is OFF.
 *
 *-------------------------------------------------------*/
#include <ns32000.h>
dump_as_ascii()
{
  int line, offset;
  char *pict;

  for (line = 0, pict = page;
            line < PAGE_HEIGHT;
               line++, pict += PAGE_WIDTH_IN_BYTES)
  {
/*
 * Use of tbit to test each bit on the current line.
 */
      for (offset = 0; offset < PAGE_WIDTH_IN_BITS; offset++)
          putchar( _tbit(offset, pict) ? '*' : ' ');

      putchar('\n');
  }
}
```

The code is compiled using the following syntax:

UNIX environment

```
nmcc  -KCG16 filename
```

VMS environment

```
NMCC /TARGET=(CPU=CG16) filename
```

E.4.2 NSGX320 specific instructions

An example of a part of an implementation of a digital FIR (Finite Impulse Response) filter is illustrated in this section.

The following definitions of the types `WCOMPLEX` and `DCOMPLEX` (used in the example) are found in `gx320.h` (see Section 8.4.1):

```
typedef struct WCOMPLEX {
      short re;
      short im;
} WCOMPLEX;

typedef struct DCOMPLEX {
      long re;
      long im;
} DCOMPLEX;
```

This example shows a common operation in various DSP (Digital Signal Processing) applications. The function performs a complex multiply and accumulate operation on two complex vectors of length ten, and then scales down the complex result from 32 bit to 16 bits. The C code is:

```
#include <gx320.h>

#define HALF 16384
#define SHIFT 15
WCOMPLEX b[10];
WCOMPLEX a[10];
WCOMPLEX c;
DCOMPLEX init_result = {0,0};
for_fir()
{
    DCOMPLEX result;
    int i;

    /* initialize result to 0 */
    result = init_result;

    for(i = 0 ; i < 10; i++)
        _cmacd(&result, a[i], b[i]);

    /* now scale down from 32 bits back to 16 bits */
    c.re = (result.re + HALF) >> SHIFT ;
    c.im = (result.im + HALF) >> SHIFT ;
}
```

The following code was produced for the for_fir() function, when compiled with -KCGX320 and -O:

```
_for_fir:
movd    _init_result,r0
movd    _init_result+(4),r1
cmacd   _a,_b
cmacd   _a+(4),_b+(4)
cmacd   _a+(8),_b+(8)
cmacd   _a+(12),_b+(12)
cmacd   _a+(16),_b+(16)
cmacd   _a+(20),_b+(20)
cmacd   _a+(24),_b+(24)
cmacd   _a+(28),_b+(28)
cmacd   _a+(32),_b+(32)
cmacd   _a+(36),_b+(36)
movqd   $(0),tos
movqd   $(0),tos
movd    r0,0(sp)
movd    r1,4(sp)
movd    0(sp),r0
movd    4(sp),r1
addd    $(16384),r0
addd    $(16384),r1
ashd    $(-15),r0
ashd    $(-15),r1
cmpd    tos,tos
movw    r0,_c
movw    r1,_c+(2)
ret     0
```

NOTE: The `nop` instructions were used because `r0` and `r1` can not be accessed for two instructions after the `cmacd` instruction.

E.5 PROGRAMMING TRAP/INTERRUPT ROUTINES

The example used in this section is a clock display for the time of day. The routine `clock_handler` handles a clock interrupt, which occurs `TICKS_PER_SECOND` times per second. The time display is updated every second.

Since the routine does not use floating-point registers, `save_regs=int_regs` is specified in the `pragma` directive (i.e. only integer registers are saved). The saved registers include all those used by the routine. Scratch registers are also saved because the handler calls the routine `update_time_display`.

The C code for the clock interrupt handler is:

```
#pragma interrupt(clock_int_routine,save_regs=int_regs);

void clock_int_routine(void)
{
   static int counter;
   static int hours;
   static int minutes;
   static int seconds;

   counter++;
   if (counter == TICKS_PER_SECOND)
     {
        seconds++;
        counter = 0;
        if (seconds == 60)
          {
            minutes++;
            seconds = 0;
            if (minutes == 60)
              {
                hours++;
                minutes = 0;
                if (hours == 24)
                   hours = 0;
              }
          }
         update_time_display(hours,minutes,seconds);
     }
}
```

Certain CPUs of the *Series 32000/EP* microprocessor family can be set to work in either direct or indirect exception mode.

When direct exception mode is enabled the address of the trap handler (residing in the interrupt dispatch table) is interpreted by the CPU as a pointer. The clock interrupt entry in the interrupt dispatch table should be set to the address of `_clock_int_routine`. The following line is inserted to the clock interrupt entry in the initialization of the interrupt dispatch table

```
.double @_clock_int_routine
```

When direct exception mode is disabled (or non existent), the address of the trap handler (residing in the interrupt dispatch table) is interpreted by the CPU as an external procedure descriptor (i.e. mod + offset). The clock interrupt entry in the interrupt dispatch table should be set to the descriptor of `_clock_int_routine`. The following line should inserted to the clock interrupt entry in the initialization of the interrupt dispatch table

```
.xpd _clock_int_routine
```

In addition the interrupt handler should be associated to a module table entry. It is recommended to do so by adding the following `asm` statement to the C source file before the interrupt handled definition

```
asm(".module");
```

For more details on modular and direct exception mode see the *Series 32000* instruction set and the GNX Assembler manual.

INDEX

G

H

I

J

P

Q

R

S

T

U

V

W

X

Z

National Semiconductor

READER'S COMMENT FORM

In the interest of improving our documentation, National Semiconductor invites your comments on this manual.

Please restrict your comments to the documentation. Technical Support may be contacted at:

(800) 759-0105 - US and Canada
((0)8141) 103-330 - Germany only

Please rate this document according to the following categories. Include your comments below.

	EXCELLENT	GOOD	ADEQUATE	FAIR	POOR
Readability (style)	☐	☐	☐	☐	☐
Technical Accuracy	☐	☐	☐	☐	☐
Fulfills Needs	☐	☐	☐	☐	☐
Organization	☐	☐	☐	☐	☐
Presentation (format)	☐	☐	☐	☐	☐
Depth of Coverage	☐	☐	☐	☐	☐
Overall Quality	☐	☐	☐	☐	☐

NAME _____ DATE _____

TITLE _____

COMPANY NAME/DEPARTMENT _____

ADDRESS _____

CITY _____ STATE _____ ZIP _____

Do you require a response? ☐ Yes ☐ No PHONE _____

Comments:

_____

_____

_____

_____

_____

GNX — Version 4.0 C Optimizing Compiler Reference Manual

FOLD, STAPLE, AND MAIL

424010516-004

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

National Semiconductor Corporation
Technical Publications Dept., M/S E295
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-9968

National Semiconductor

SALES OFFICES

ALABAMA
Huntsville
(205) 837-8960
(205) 721-9367

ARIZONA
Tempe
(602) 966-4563

B.C.
Burnaby
(604) 435-8107

CALIFORNIA
Encino
(818) 888-2602
Inglewood
(213) 645-4226
Roseville
(916) 786-5577
San Diego
(619) 587-0666
Santa Clara
(408) 562-5900
Tustin
(714) 259-8880
Woodland Hills
(818) 888-2602

COLORADO
Boulder
(303) 440-3400
Colorado Springs
(303) 578-3319
Englewood
(303) 790-8090

CONNECTICUT
Fairfield
(203) 371-0181
Hamden
(203) 288-1560

FLORIDA
Boca Raton
(305) 997-8133
Orlando
(305) 629-1720
St. Petersburg
(813) 577-1380

GEORGIA
Atlanta
(404) 396-4048
Norcross
(404) 441-2740

ILLINOIS
Schaumburg
(312) 397-8777

INDIANA
Carmel
(317) 843-7160
Fort Wayne
(219) 484-0722

IOWA
Cedar Rapids
(319) 395-0090

KANSAS
Overland Park
(913) 451-8374

MARYLAND
Hanover
(301) 796-8900

MASSACHUSETTS
Burlington
(617) 273-3170
Waltham
(617) 890-4000

MICHIGAN
W. Bloomfield
(313) 855-0166

MINNESOTA
Bloomington
(612) 835-3322
(612) 854-8200

NEW JERSEY
Paramus
(201) 599-0955

NEW MEXICO
Albuquerque
(505) 884-5601

NEW YORK
Endicott
(607) 757-0200
Fairport
(716) 425-1358
(716) 223-7700
Melville
(516) 351-1000
Wappinger Falls
(914) 298-0680

NORTH CAROLINA
Cary
(919) 481-4311

OHIO
Dayton
(513) 435-6886
Highland Heights
(216) 442-1555
(216) 461-0191

ONTARIO
Mississauga
(416) 678-2920
Nepean
(404) 441-2740
(613) 596-0411
Woodbridge
(416) 746-7120

OREGON
Portland
(503) 639-5442

PENNSYLVANIA
Horsham
(215) 675-6111
Willow Grove
(215) 657-2711

PUERTO RICO
Rio Piedas
(809) 758-9211

QUEBEC
Dollard Des Ormeaux
(514) 683-0683
Lachine
(514) 636-8525

TEXAS
Austin
(512) 346-3990
Houston
(713) 771-3547
Richardson
(214) 234-3811

UTAH
Salt Lake City
(801) 322-4747

WASHINGTON
Bellevue
(206) 453-9944

WISCONSIN
Brookfield
(414) 782-1818
Milwaukee
(414) 527-3800

INTERNATIONAL OFFICES

Electronica NSC de Mexico SA
Juventino Rosas No. 118-2
Col Guadalupe Inn
Mexico, 01020 D.F. Mexico
Tel: 52-5-524-9402

National Semicondutores Do Brasil Ltda.
Av. Brig. Faria Lima, 1409
6 Andor Salas 62/64
01451 Sao Paulo, SP, Brasil
Tel: (55/11) 212-5066
Telex: 391-1131931 NSBR BR

National Semiconductor GmbH
Industriestrasse 10
D-8080 Furstenfeldbruck
West Germany
Tel: 49-08141-103-0
Telex: 527 649

National Semiconductor (UK) Ltd.
301 Harpur Centre
Horne Lane
Bedford MK40 ITR
United Kingdom
Tel: (02 34) 27 00 27
Telex: 826 209

National Semiconductor Benelux
Vorstlaan 100
B-1170 Brussels
Belgium
Tel: (02) 6725360
Telex: 61007

National Semiconductor (UK) Ltd.
1, Bianco Lunos Alle
DK-1868 Fredriksberg C
Denmark
Tel: (01) 213211
Telex: 15179

National Semiconductor
Expansion 10000
28, rue de la Redoute
F-92260 Fontenay-aux-Roses
France
Tel: (01) 46 60 81 40
Telex: 250956

National Semiconductor S.p.A.
Strada 7, Palazzo R/3
20089 Rozzano
Milanofiori
Italy
Tel: (02) 8242046/7/8/9

National Semiconductor AB
Box 2016
Stensatravagen 13
S-12702 Skarholmen
Sweden
Tel: (08) 970190
Telex: 10731

National Semiconductor
Calle Agustin de Foxa, 27
28036 Madrid
Spain
Tel: (01) 733-2958
Telex: 46133

National Semiconductor Switzerland
Alte Winterthurerstrasse 53
Postfach 567
Ch-8304 Wallisellen-Zurich
Switzerland
Tel: (01) 830-2727
Telex: 59000

National Semiconductor
Kauppakartanonkatu 7
SF-00930 Helsinki
Finland
Tel: (0) 33 80 33
Telex: 126116

National Semiconductor Japan Ltd.
Sanseido Bldg. 5F
4-15 Nishi Shinjuku
Shinjuku-ku
Tokyo 160 Japan
Tel: 3-299-7001
Fax: 3-299-7000

National Semiconductor Hong Kong Ltd.
Southeast Asia Marketing
Austin Tower, 4th Floor
22-26A Austin Avenue
Tsimshatsui, Kowloon, H.K.
Tel: 852 3-7243645
Cable: NSSEAMKTG
Telex: 52996 NSSEA HX

National Semiconductor (Australia) PTY, Ltd.
1st Floor, 441 St. Kilda Rd.
Melbourne, 3004
Victory, Australia
Tel: (03) 267-5000
Fax: 61-3-2677458

National Semiconductor (PTE), Ltd.
200 Cantonment Road 13-01
Southpoint
Singapore 0208
Tel: 2252226
Telex: RS 33877

National Semiconductor (Far East) Ltd.
Taiwan Branch
P.O. Box 68-332 Taipei
7th Floor, Nan Shan Life Bldg.
302 Min Chuan East Road,
Taipei, Taiwan R.O.C.
Tel: (86) 02-501-7227
Telex: 22837 NSTW
Cable: NSTW TAIPEI

National Semiconductor (Far East) Ltd.
Korea Office
Room 612,
Korea Fed. of Small Bus. Bldg.
16-2, Yoido-Dong,
Youngdeungpo-Ku
Seoul, Korea
Tel: (02) 784-8051/3 - 785-0696-8
Telex: K24942 NSRKLO

National Semiconductor Corporation printed in U.S.A.