GNX In-System Emulators

Version 3

Series 32000®

GNX — Version 3
C Optimizing Compiler
Reference Manual

Customer Order Number NSP-C-V3-M
NSC Publication Number 424010516-003B
May 1989

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
A	Sep 1988	First Release.
B	May 1989	miscellaneous bugfixes. fast fp emulation support added. GX32 support added.

PREFACE

This is a reference manual for National Semiconductor Corporation's GNX—Version 3 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 Version 3 of the 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.

No part of this document may be reproduced in any form or by any means without the prior written consent of National Semiconductor Corporation.

GENIX, NSX, ISE, ISE16, ISE32, SYS32, and TDS 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

Chapter 1 OVERVIEW

1.1 INTRODUCTION 1-1
1.2 AUDIENCE 1-2
1.3 FEATURES AND SUPPORTED LANGUAGE EXTENSIONS . . 1-2
1.3.1 Compiler Features 1-2
1.3.2 Supported C Language Extensions 1-3
1.4 DOCUMENTATION CONVENTIONS 1-3
1.4.1 General Conventions 1-3
1.4.2 Conventions in Syntax Descriptions 1-4
1.4.3 Example Conventions 1-4

Chapter 2 COMPILATION PROCESS

2.1 INTRODUCTION 2-1
2.2 C OPTIMIZING COMPILER STRUCTURE 2-1
2.3 COMMAND LINE OPTIONS 2-2
2.3.1 UNIX Compilation Options 2-2
2.3.2 VMS Compilation Qualifiers 2-9
2.4 TARGET MACHINE SPECIFICATION 2-13
2.5 THE INTERNAL COMPILER TABLES 2-14
2.6 FLOATING-POINT EMULATION 2-14
2.6.1 Floating-point Emulation — Native Host 2-15
2.6.2 Floating-point Emulation — Native Host, Cross Support 2-15
2.6.3 Floating-point Emulation — VAX/UNIX system 2-15
2.6.4 Floating-point Emulation — VAX/VMS System 2-16
2.7 ENVIRONMENT VARIABLES 2-16

Chapter 3 EXTENSIONS TO THE C LANGUAGE

3.1 INTRODUCTION 3-1
3.2 SINGLE-PRECISION FLOATING CONSTANTS 3-1
3.3 UNSIGNED CONSTANTS 3-1
3.4 ENUMERATED TYPE 3-2
3.5 STRUCTURE HANDLING 3-2
3.6 VOID DATA TYPE 3-2
3.7 BITFIELDS 3-2

3.8 VOLATILE AND CONST 3-2
3.8.1 Volatile 3-3
3.8.2 Const 3-4
3.9 ASM 3-5
3.10 IDENT 3-6

Chapter 4 IMPLEMENTATION ISSUES
4.1 INTRODUCTION 4-1
4.2 IMPLEMENTATION ASPECTS 4-1
4.2.1 Memory Representation 4-1
4.2.2 External Linkage 4-2
4.2.3 Types and Conversions 4-2
4.2.4 Variable and Structure Alignment 4-2
4.2.5 Structure Returning Functions 4-9
4.2.6 Calling Sequence 4-9
4.2.7 Mixed-Language Programming 4-9
4.2.8 Order of Evaluation 4-10
4.2.9 Order of Allocation of Memory 4-10
4.2.10 Register Variables 4-10
4.2.11 Floating-Point Arithmetic 4-11
4.3 UNDEFINED BEHAVIOR 4-11

Chapter 5 OPTIMIZATION TECHNIQUES
5.1 INTRODUCTION 5-1
5.2 THE OPTIMIZER 5-2
5.3 THE CODE GENERATOR 5-8
5.4 MEMORY LAYOUT OPTIMIZATIONS 5-9

Chapter 6 GUIDELINES ON USING THE OPTIMIZER
6.1 INTRODUCTION 6-1
6.2 OPTIMIZATION FLAGS 6-1
6.2.1 Optimization Options on the Command Line — UNIX Systems 6-1
6.2.2 Optimization Options on the Command Line — VMS Systems 6-3
6.2.3 Turning off Optimization Options 6-3
6.3 PORTING EXISTING C PROGRAMS 6-3
6.3.1 Undetected Program Errors 6-5
6.3.2 Compiling System Code 6-5
6.3.3 Timing assumptions 6-6
6.3.4 Low-Level Interface 6-6
6.3.5 Using Nonstandard Library Routines 6-7

6.4 DEBUGGING OF OPTIMIZED CODE 6-8
6.5 ADDITIONAL GUIDELINES FOR IMPROVING CODE QUALITY 6-9
6.5.1 Static Functions 6-9
6.5.2 Integer Variables 6-10
6.5.3 Local Variables 6-10
6.5.4 Floating-Point Computations 6-10
6.5.5 Pointer Usage 6-11
6.5.6 Asm Statements 6-13
6.5.7 Register Allocation 6-13
6.5.8 setjmp() 6-14
6.5.9 Optimizing for Space 6-14
6.6 COMPILATION TIME REQUIREMENTS 6-14

Appendix A SERIES 32000 STANDARD CALLING CONVENTIONS

A.1 INTRODUCTION A-1
A.2 CALLING CONVENTION ELEMENTS A-1

Appendix B MIXED-LANGUAGE PROGRAMMING

B.1 INTRODUCTION B-1
B.1.1 Writing Mixed-Language Programs B-1
B.1.2 Compiling Mixed-Language Programs B-6
B.1.3 Compilation on UNIX Operating Systems B-7
B.1.4 Compilation on VMS Operating Systems B-8
B.2 COMPILING THE MIXED-LANGUAGE EXAMPLE B-8
B.2.1 Compiling the Example on a UNIX System B-9
B.2.2 Compiling the Example on a VMS System B-9
B.3 PROGRAM MODULE LISTINGS B-9

Appendix C ERROR MESSAGES

C.1 INTRODUCTION C-2
C.1.1 Warnings C-2
C.1.2 Errors C-2
C.2 Error Messages C-2

Appendix D COMPILER OPTIONS

D.1 INTRODUCTION D-2

FIGURES

Figure 4-1. Bitfield Padding 4-6
Figure 4-2. Alignment on Bitfields 4-7

Figure 5-1.	Relationship Between Various Optimizations	5-3
Figure 5-2.	Flow Graph	5-4
Figure 5-3.	Example of Partial Redundancy Elimination	5-6
Figure B-1.	Cross-Language Pairs	B-2

TABLES

Table 2-1.	Filename Conventions	2-4
Table 2-2.	Target Selection Parameters	2-14
Table 2-3.	Internal Compiler Tables	2-15
Table 4-1.	Variable Alignment	4-3
Table 6-1.	Optimization Options	6-2
Table 6-2.	Turning off Optimization Options	6-4
Table 6-3.	Recognized Library Routines	6-7
Table B-1.	Compilers and their Associated Libraries	B-7
Table D-1.	UNIX Operating System Options	D-2
Table D-2.	VMS Operating System Options	D-4
Table D-3.	Options Passed to the Preprocessor — UNIX Systems	D-5
Table D-4.	Options Passed to the Preprocessor — VMS Systems	D-5
Table D-5.	Options Recognized and Passed to the Linker	D-6

INDEX

Chapter 1

OVERVIEW

1.1 INTRODUCTION

This manual describes National Semiconductor's GNX—Version 3 C optimizing compiler. This compiler is one of a family of compatible optimizing compilers for the *Series 32000* architecture.* The GNX—Version 3 C compiler replaces and obsoletes the previous GNX—Version 2 C compiler. It implements the C language as described in the *C Programming Language* by Kernighan and Ritchie. It is fully compatible with the System V C compiler, a compiler derived from the portable C compiler (pcc). It is also enhanced by several features from the draft-proposed ANSI C standard (referred to throughout this manual as standard C). The GNX—Version 3 C optimizing compiler is available as a cross-support compiler on VMS™ and UNIX® operating systems as well as a native compiler on *Series 32000* operating systems derived from UNIX System V, Release 3. A list of related reference documentation (*e.g.*, *Series 32000 GNX — Version 3 Assembler Reference Manual*) are located in the *Series 32000 GNX — Version 3 Commands and Operations Manual*.

This manual is organized as follows:

- Introduction (Chapter 1)
- Compilation Process (Chapter 2)
- Extensions to Standard C (Chapter 3)
- Implementation Issues (Chapter 4)
- Optimization Techniques (Chapter 5)
- Guidelines on Using the Optimizer (Chapter 6)
- *Series 32000* Calling Standard Conventions (Appendix A)
- Mixed-Language Programming (Appendix B)
- Error Messages (Appendix C)
- Compiler Options (Appendix D)

---

* At this writing, the family consists of a C optimizing compiler, Pascal optimizing compiler, FORTRAN 77 optimizing compiler, and Modula-2 optimizing compiler.

1.2 AUDIENCE

This manual is intended for use as a reference manual by 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. To aid the user who does not have such knowledge, a chapter on optimization techniques used by the optimizer is provided. This chapter supplies the background information necessary to understand and use the compiler optimization flags. A second optimization chapter provides guidelines to help the programmer avoid problems that could occur when using the optimizer.

This manual assumes that the reader has a working knowledge of the C language. Recommended C reference and tutorial books include:

> Harbison, Samuel and Steele, Guy. *C, A Reference Manual*, Second Edition, Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1984.
>
> Kernighan, Brian and Ritchie, Dennis. *The C Programming Language,* First Edition. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.
>
> Kochan, Stephen. *Programming in C.* Rochelle Park, New Jersey: Hayden Book Company, 1983.
>
> Purdam, Jack. *C Programming Guide.* Indianapolis, Indiana: Que Corporation, 1983.
>
> Plum, Thomas. *Learning to Program in C.* Philadelphia, Pennsylvania: Plum Hall Inc., 1983.

1.3 FEATURES AND SUPPORTED LANGUAGE EXTENSIONS

1.3.1 Compiler Features

The following are the main features of the C optimizing compiler:

- `pcc` compatible.
- Accepts standard C language and many extensions (Section 1.3.2 lists supported C language extensions).
- Optimizations can be tuned to either improve speed or save space.
- Optimization level is controlled by the user.
- The compiler generates code tuned to the specific target system.
- Full support of mixed-language programming.
- Full support of read-only constants.
- Full support of volatile variables.
- User controlled alignment of variables and structure members.

- Improved structure handling over that defined in *The C Programming Language* book by Kernighan and Ritchie.
- Assembly output can be annotated with source lines.

1.3.2 Supported C Language Extensions

The compiler implements the full C language as defined by the "C Reference Manual," Appendix A of the *C Programming Language* book, by Kernighan and Ritchie. In addition, the following extensions are supported:

- New ANSI C keyword/datatypes:
  - `const` for defining read-only entities
  - `volatile` for sensitive variables
- ANSI C support for the void data type. This includes its use as a cast and "pointer to void" as the universal pointer.
- Structures may be assigned, passed as arguments and returned from functions.
- Structure/union member names need not be globally unique.
- Structure and union size are not limited.
- Unsigned constants (to save run-time conversions).
- Single-precision floating constants (to save run-time conversions).
- Enumeration datatype, compatible with int type (ANSI).
- 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");
```

Chapter 2

COMPILATION PROCESS

2.1 INTRODUCTION

The GNX—Version 3 C Compiler is a modular language processor consisting of five separate programs. This chapter describes the five programs, the GNX—Version 3 C Compiler invocation, options available to the user, and file generation.

2.2 COMPILER STRUCTURE

The GNX—Version 3 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 standard 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 3 Compilers, *i.e.*, the FORTRAN 77 Compiler, the Pascal Compiler, the C Compiler, and the Modula-2 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.

2.3 COMMAND LINE OPTIONS

The GNX—Version 3 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.* 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 3 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.

---

* For details on how to use GTS, see the appropriate *Series 32000 GNX — Version 3 Commands and Operations* manual.

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, FORTRAN 77, and Modula-2 source files are also recognized, and compile appropriately if your system includes the National Semiconductor GNX — Version 3 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, typically produced by previous runs of the GNX—Version 3 C Compiler, and pass directly to the linker. The object files link into one executable file with the default name `a.out` (or `a32.out` in a cross-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
.m, .mod .M, .MOD .def .DEF	Modula-2 source file Modula-2 source with cpp directives Modula-2 definition module source file Modula-2 definition module 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.*

-O (PERFORMS OPTIMIZATIONS)

-F*flags* (SPECIFIES OPTIMIZATION FLAGS)

-O*flags* (PERFORMS OPTIMIZATIONS ACCORDING TO FLAGS)

The -O option directs the GNX—Version 3 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

* The GNX—Version 3 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.

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*flags* (GENERATES RUN-TIME CHECKS)
This is only useful when compiling Pascal, FORTRAN 77, and Modula-2 programs.

-g (PREPARES SYMBOLIC DEBUGGING INFORMATION)
The `-g` option instructs the GNX—Version 3 C Compiler to prepare symbolic debugging information for symbolic debuggers, such as `idbg32`. See the discussion on debugging of optimized code in Section 6.4.

-i*dir* (SPECIFIES DIRECTORY TO IMPORT FROM)
This option is useful only when compiling GNX — Version 3 Modula-2 programs.

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

-c (COMPILES BUT DOES NOT LINK)
The `-c` option directs the GNX—Version 3 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 3 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 (IMBEDS 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`. This option renames the executable file, whose default name is `a.out` (or `a32.out` in a cross-support environment). 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 3 C Optimizing Compiler normally prints warnings regarding inconsistencies in the input program. The `-w` option suppresses these 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 draft-proposed 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.

-A*n* (CONFORMS TO EDITION *n* OF REPORT)
This is only useful when compiling Modula-2 programs. *n* can be either 2 or 3.

-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 allows the user to change the default size of certain internal tables of the compiler. The default sizes are sufficient for most applications, but can be overridden using this option. See Section 2.5 for more details.

-v (VERBOSE)
This option lists the subprograms of the GNX—Version 3 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 3 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 use of external references). For further information see the *Series 32000 GNX — Version 3 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 3 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)

The following option can be used as an "escape" to pass additional options (not recognized by the GNX—Version 3 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 imbedded 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 3 C Compiler is as follows:

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

The normal operation of the GNX—Version 3 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 3 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:

/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.

/OPTIMIZE [=(*flags*)]
: This qualifier directs the GNX—Version 3 C Compiler to perform global optimizations. A detailed discussion of the GNX—Version 3 C Compiler optimization techniques is located in Chapter 5 and Chapter 6. Default is `/NOOPTIMIZE`.

/DEBUG
: The `/DEBUG` qualifier instructs the GNX—Version 3 C Compiler to prepare symbolic debugging information for symbolic debuggers, such as `dbg32`. See the discussion on debugging of optimized code in Section 6.4. Default is `/NODEBUG`.

/PROFILE
: The `/PROFILE` qualifier prepares profiling information for a program profiler, such as `prof`. Default is `/NOPROFILE`.

/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`.

/ANNOTATE

This qualifier directs the compiler to put GNX—Version 3 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`.

/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 3 C Compiler's alignment scheme.

/WARNING

The GNX—Version 3 C Compiler prints warnings regarding inconsistencies found in the input program. The `/NOWARNING` qualifier suppresses these warning diagnostics. Default is `/WARNING`.

/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 qualifier allows the user to change the default size of certain internal tables of the compiler. The default sizes are sufficient for most applications, but can be overridden using this qualifier. See Section 2.5 for more details.

/VERBOSE

This qualifier lists the parts of the GNX—Version 3 C Compiler as they are called by the driver program. Default is `/NOVERBOSE`.

/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 3 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.

/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 3 Language Tools Technical Notes* and the *Series 32000 Programmer's Reference Manual*. Default is /NOMODULAR.

/ERROR[=*filename*]

The /ERROR qualifier instructs the GNX—Version 3 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.ERR FILE1.C
```

creates an error log file named FILE1.ERR.

/PRE_PROCESSOR

This qualifier causes the source file to be passed to the GNX C preprocessor before the normal processing by the GNX—Version 3 C language parser. Default is /NOPRE_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
```

/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 `/COMMENT`.

/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 `INCLUDE`, 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 `/TARGET` (`-K`) 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`. 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]381	2
[NS32]cg16	[NS32]580	4
[NS32]032	emulation	
[NS32]332	nofpu	
[NS32]532		
[NS32]gx32		

Example: The following example specifies an NS32332 CPU, an NS32081 FPU, and a buswidth of 4 bytes.

UNIX

```
cc -KC332 -KF081 -KB4 temp.c
```

or `nmcc -KC332 -KF081 -KB4 temp.c` (cross-support)

VMS

```
NMCC /TARGET=(CPU=332,FPU=081,BUS=4) TEMP.C
```

2.5 THE INTERNAL COMPILER TABLES

The compiler has a default size for each of its internal static tables. This size is sufficient for most applications; however, some programs can cause overflow of one or more of the tables. If this happens, the compiler can be run with the /TABLE_SIZE qualifier (-N on UNIX systems) to increase the size of the appropriate table.

The compiler gives an error message specifying in which table the overflow occurred and which option to use to overcome the problem. Table 2-3 shows the options.

Table 2-3. Internal Compiler Tables

UNIX FLAG	VMS FLAG	DEFAULT SIZE
c	CONTROL_NESTING	100
i	IR_BUFFER	2048
n	IDENTIFIERS	1200
t	INTERNAL_TREE	5100

The tables can be expected to overflow when:

- CONTROL_NESTING (c)
  the program includes deeply-nested compound statements (if/while/do/switch).
- IR_BUFFER (i) and INTERNAL_TREE (t)
  there are very complex expressions or compile-time initialization of very large aggregates.
- IDENTIFIERS (n)
  the program uses a very large number of identifiers.

In addition to the flags mentioned, there exist flags that were used in the development of the compiler. These flags, DIM_TABLE (d), PARAM_STACK (p), MULTI_TABLE (m), and SWITCH_TABLE (s), should rarely, if ever, be used. In the case of a problem with one of these internal compiler tables, the user gets an error message describing the problem and what to do to solve the problem.

2.6 FLOATING-POINT EMULATION

Two different floating point emulation options are available with the GNX—Version 3 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 3 Support Libraries Reference Manual*. The use of the `Hfp` package is indicated by the `-KFemulation` compiler option (`/TARGET=(FPU=emulation)` on VMS). The use of the `fpee` package is indicated by the `-f` or `-KFnofpu` compiler option (`/TARGET=(FPU=nofpu)` on VMS). These options may also be set permanently by using the GTS facility.

2.6.1 Floating-point Emulation — Native Host

There is no way to unconfigure the FPU on the SYS32/30 or SYS32/20, and no floating-point emulation is therefore required. To use the `fpee` library you must do the following:

1. Include `asm("bsr _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 — Native Host, Cross

On a *Series 32000*/UNIX system, cross-application linking to the floating-point emulation library must be explicitly requested. For example,

```
cc -c file1.c file2.c
ld GNXDIR/lib/db_fcrt0.o file1.o file2.o -ldb_fpe -db_c
```

2.6.3 Floating-point Emulation — VAX UNIX system

On a VAX/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:

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

2.6.4 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

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, 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`.

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

See Section 6.6 for use of the AVAIL_SWAP environment variable.

Chapter 3

EXTENSIONS TO THE C LANGUAGE

3.1 INTRODUCTION

The GNX—Version 3 C compiler is National Semiconductor Corporation's implementation of the UNIX portable C compiler, `pcc`. The GNX—Version 3 C compiler is fully compatible with `pcc`. All `pcc` extensions to the C language (as defined by Kernighan and Ritchie) are implemented by the GNX—Version 3 C compiler. In addition, several features from the draft-proposed ANSI C Standard are implemented. This chapter lists and describes the extensions implemented by the GNX—Version 3 C compiler.

3.2 SINGLE-PRECISION FLOATING CONSTANTS

Single-precision 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;`. Without the cast or the suffix, both `fmax` and the value `17.0` would have been converted to double-precision for a double-precision addition. The result then would have been converted back to single-precision.

3.3 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.

3.4 ENUMERATED TYPE

Enumerated types allow the user to name lists of identifiers. They are analogous to the `enumeration` type of Pascal and can be used in any place an integer is used. See *C, A Reference Manual*, Second Edition, by Harbison and Steele, for more details.

3.5 STRUCTURE HANDLING

The GNX—Version 3 C compiler implements the following improvements to structure handling:

- implements structure assignment
- allows structures as function arg and return values
- allows structures as function arguments and return values
- allows reuse of structure and union member names
- does not limit structure size

See *C, A Reference Manual*, Second Edition, by Harbison and Steele, for more details.

3.6 VOID DATA TYPE

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

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

3.7 BITFIELDS

The GNX—Version 3 C compiler implements signed bitfields as well as unsigned bitfields. Due to the *Series 32000* architecture, the code for unsigned bitfields is more efficient than the code for signed bitfields.

3.8 VOLATILE AND CONST

The compiler includes a partial implementation of the draft-proposed ANSI C standard type qualifiers `const` and `volatile`. The compiler recognizes the full syntax involving these new keywords. The semantics are a subset of the semantics of the draft, as described in the following.

NOTE: The words `volatile` and `const` are reserved keywords. Using them as identifiers is a syntax error. Existing programs using such identifiers will have to be modified.

3.8.1 Volatile

This feature is a refinement of the optimizer's `/VOLATILE` flag (`-Oi` on UNIX systems). Using this keyword, it is possible to be more specific than turning *all* global variables and *all* pointer dereferences to volatile, which is what the `/VOLATILE` flag does. Because the programmer can specify what is volatile, better optimization of the code results.

The semantics of this implementation are as follows:

- All variables declared with the keyword `volatile` appearing anywhere in its type are treated as volatile.
- All elements of an array declared with the keyword `volatile` appearing anywhere in its type are treated as volatile.
- All members of a structure/union declared with the keyword `volatile` appearing anywhere in its type are treated as volatile.
- Any dereference involving a pointer declared with the keyword `volatile` appearing anywhere in its type is treated as volatile.

Note that this is not a full implementation of the draft-proposed ANSI C standard. It is, however, more conservative and, therefore, safer than the draft-proposed ANSI C standard requirement. For instance, a nonvolatile pointer to a volatile integer in the draft-proposed ANSI C standard meaning is not possible by declaration.

Example: In the declaration

```
volatile int *ptr_to_vol;
```

the variable `ptr_to_vol`, as well as the `int` to which it points, will be treated as volatile.

In order to have a nonvolatile pointer to a volatile integer, explicit cast operators must be used. Any `lvalue` expression may be cast to a type that includes the keyword `volatile`. This cast will treat the `lvalue`, and all expressions including it, as volatile.

Example: To achieve the effect of a nonvolatile pointer to a volatile integer, use:

```
int *p;
.
.
.
(volatile) *(p+15) = 237;
```

Note that expressions involving `p` may be optimized as nonvolatile expressions. In the previous example, the result of `(p+15)` may be kept in a register. The dereference, *i.e.*, where `p+15` is pointing, is treated as volatile.

3.8.2 Const

This feature enables the user to allocate program entities in a read-only area of memory. The `const` keyword is useful for two reasons:

1. constants can be made a part of the program code and placed into ROMs for ROM-based applications. For example, the code to boot a system, together with any messages to be displayed, can be placed on a ROM.
2. if during run-time an attempt is made to change an intended constant, a trap will occur.

The keyword `const` will have an effect only on global or static entities with one of the following types:

- simple typed objects with `const` in their type
- pointers with the last (*) asterisk followed by `const`
- an array of one of the two above allowed types

Such an object, that is declared as `const`, will be allocated in read-only memory (the `.text` area) if it is initialized and is not also declared to be volatile.

No compile-time checks are made to detect attempts to change to value on a `const` entity.

Example:

```
const char string[] = "for the snark WAS a boojum, you
see.";
const int LIMIT = 101;
const short action[ ] = {
        -1, 0, 57, 280, 58, 280,
        -2, 0, -1, 61, -1, 71
        };
const struct {
        int year;
        char *name;
        } lwork = { 1876, "The hunting of the snark"};
```

Assuming these are global declarations, `string`, `actions`, `LIMIT`, and `lwork` will be placed in the `.text` segment of the assembly file (read-only memory).

The following uses of the keyword `const` will have no effect in the current implementation:

- `static const int i;` no initialization
- `static volatile const int j = 0773;` also volatile
- `const int *p = &i;` `p` is declared as pointing to a constant integer. This currently has no effect. For the pointer itself to be placed in read-only memory, use `int *const p = &i;`.
- casts

3.9 ASM

The keyword `asm` is recognized to enable insertion of assembly instructions directly into the assembly file generated. The syntax of its use is

asm (*constant-string*) **;**

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

`Asm` can be used inside of functions as a statement and out of functions in the scope of global declarations. A newline character will be appended to the given string in the assembly code.

Example: if for the C source:

```
i++;
j += 2;
```

the assembly code generated is:

```
addqd $1,   _i
addqd $2,   _j
```

then the assembly code generated for:

```
i++;
asm("movd _i,  r0");
j += 2;
```

will be:

```
addqd $1,   _i
movd  _i,   r0
addqd $2,   _j
```

NOTE: The word `asm` is a reserved keyword. Using `asm` as an identifier is a syntax error. Existing programs using such identifiers will have to be modified.

3.10 IDENT

A new `cpp`-style directive is recognized for placing strings into the .comment section of the object file.* Its 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 3 COFF Programmer's Guide* and the *Series 32000 GNX — Version 3 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 3 C compiler behaves under the listed issues. Section 4.3 lists issues that cause an undefined behavior of the GNX—Version 3 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 3 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.

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).

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.

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.

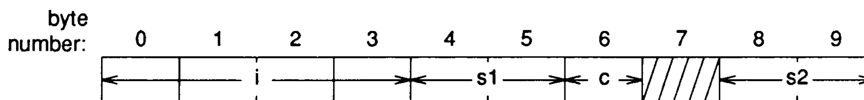
A buswidth setting of 4 means "align to a double-word boundary" (a byte whose address is divisible by four). Variables that are 2 bytes long start on a word boundary; variables that are 4 bytes or larger in size start on a double-word boundary. This means that there may be padding of up to three bytes.

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



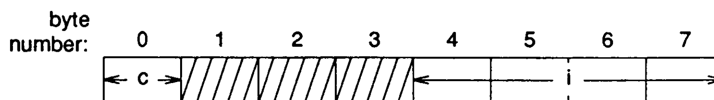
GX-01-0-U

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

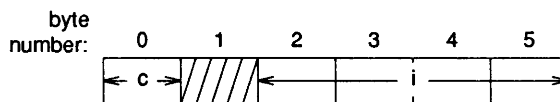
```
char c; int i;
```

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 (4 for all but the NS32008, the NS32CG16, 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.

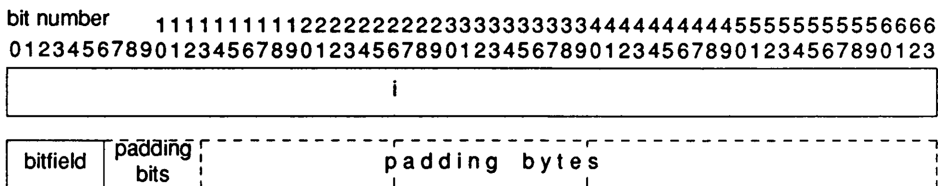
An attempt is made to pack consecutive bit-fields as much as possible, as long as the bit-fields remain in the same fetching unit. As soon as a field "spills over" into the next fetching unit, the alignment is set to the next memory unit (byte, word, or double-word, according to the align switch value 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:



GX-04-0-U

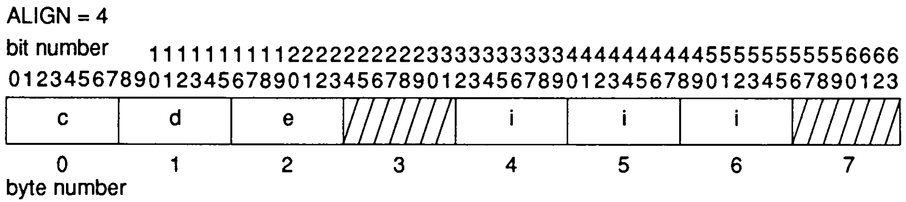
Figure 4-1. Bitfield Padding

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;
        };
```



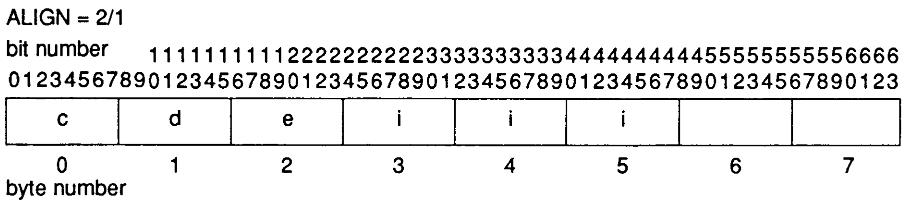


Figure 4-2. Alignment on Bitfields

CAUTION

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

Suppose the example program includes `<stdio.h>`. The file `<stdio.h>` contains the following definitions:

```
extern FILE _iob[_NFILE];

typedef struct {
        int             cnt;
        unsigned char   *_ptr;
        unsigned char   *_base;
        char            _flag;
        char            _file;
} FILE;
```

Note that FILE has two char members at its end. If align=4, any variable declared to be of type FILE 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 `<stdio.h>` is compiled with align=4 and later linked with a module compiled with align=1 or align=2 that tries to use iob[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. This situation actually does arise if a user module, compiled with align=1 or align=2, is linked with the default library libc, which is compiled with align=4.

The solution to this problem is to make sure all modules are compiled using either the same alignment setting, including all include files and libraries, 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, FILE is revised to look like:

```
typedef struct {
        int             cnt;
        unsigned char   *_ptr;
        unsigned char   *_base;
        char            _flag;
        char            _file;
        /* padding */   int:16;
} FILE;
```

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 3 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 3 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 supporting those features. By default, the GNX—Version 3 tools do not support 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 3 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 3 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 6.5.7.

4.2.11 Floating-Point Arithmetic

The floating-point arithmetic conversion rules of the GNX—Version 3 C compiler differ from most 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`. This behavior differs from previous compilers which 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 3 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 3 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 3 C compiler and other compilers is the optimizer. Recompiling and optimizing with the GNX—Version 3 C compiler will result in a 10 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 3 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
- Dead-code removal
- Loop-invariant code motion
- Strength reduction
- Induction variable elimination
- Register-allocation by coloring
- Peephole optimizations
- Memory-layout optimizations
- Fixed frame

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

5.2 THE OPTIMIZER

The optimizer, shared by all the GNX—Version 3 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 all the 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 3 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 3 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. 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.

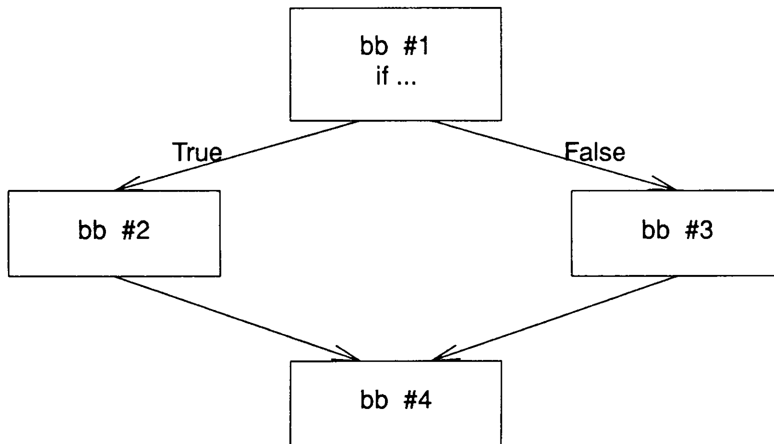


Figure 5-2. Flow Graph

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.

- **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 record number 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 saved value to replace the redundant computation. An example of this optimization is shown in Figure 5-3.

  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 the expression is saved on the entry path in a register, while the redundant computation in the loop is replaced by that register.

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 5-3. Example of Partial Redundancy Elimination

- **Strength Reduction**
  This optimization globally replaces complex operations by simpler ones. This is primarily useful for reducing complex array-subscript computations (involving multiplication into simpler additions).

Example:

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

is transformed into:

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

- **Induction Variable Elimination**
  Induction variables are variables that 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 = 0;
```

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 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 reorganizations take 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.

- Switch 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 the evaluation order of parameters is unpredictable because parameters that take more space to evaluate are treated first to save space.

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 optimizations are trade-off situations which are affected 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 3 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 compiler reorders these variables to save overall frame space.
- Code is aligned after every unconditional jump.

Chapter 6

GUIDELINES ON USING THE OPTIMIZER

6.1 INTRODUCTION

The following sections are provided as helpful guidelines on using the GNX—Version 3 C compiler. Experienced programmers should understand this compiler's optimization techniques in order to:

- Learn how to port programs to the GNX—Version 3 C compiler.
- Understand how to recognize and avoid nonportable code.
- Avoid using programming tricks which 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.

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 (`-OCFIUMLRS`). 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 option might be turned off.

Even when the optimizer pass is omitted, some local optimizations are performed by the code generator. Not specifying the optimizer pass is equivalent to entering `-OocfIumlrS`. 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.

Table 6-1. Optimization Options

UNIX	VMS	DESCRIPTION
o	NOOPT	does not invoke the optimizer phase.
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.
U	NOUSER_REGISTERS	ignores user register declarations.
u	USER_REGISTERS	allocates user-declared register variables in registers as done by `pc`.
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).

Also note that using the compiler target option (-KB1) favors space over speed by saving alignment holes normally produced when the buswidth is the default (n = 4).

6.2.2 Optimization Options on the Command Line — VMS Systems

The fastest possible code 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)
```

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 option might be turned off.

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)
```

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 buswidth is the default (n = 4).

6.2.3 Turning off 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.

6.3 PORTING EXISTING C PROGRAMS

Almost every program which runs when compiled by other C compilers, will compile and run under the GNX—Version 3 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 3 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.

Table 6-2. Turning off Optimization Options

OPTION	REASON FOR TURNING OFF OPTION	SEE ALSO
NOFIXED_FRAME (-Of)	to debug the program or to compile nonportable programs that assume knowledge of the run-time stack.	Sections 6.3.2 and 6.4
VOLATILE (-Oi)	to compile system programs, such as device drivers, which contain variables that change or are referenced spontaneously.	Section 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).	Section 6.3.5
NOFLOAT-FOLD (-Oc)	to compile programs whose correct execution depends on the order in which floating-point expressions are evaluated.	Section 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.	
USER_REGISTERS (-Ou)	to compile programs which rely on the register allocation scheme of pcc.	Section 6.5.7
NOREGISTER_ALLOCATION (-Or)	to run programs that cease to work when performing register allocation.	Section 6.5.7
NOSPEED_OVER_SPACE (-Os)	to compile programs which must fit as tightly as possible in memory.	Section 6.5.9
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. For instance, -OoF turns fixed frame on without running the optimizer, while -Of turns off fixed frame but runs the optimizer.	

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 his or her 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 3 C compiler are very different from those of other compilers, a local variable may end 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 will certainly contain zero. This may become false as a result of the register allocation phase of the GNX—Version 3 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, which 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 which can be affected by external events 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 that might be used or changed by asynchronous events, such as I/O or interrupts. The `/VOLATILE` (`-Oi` on UNIX systems) 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 the `volatile` type modifier. 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 `/USER_REGISTERS` flag (`-Ou` on UNIX systems) turned on (see Section 6.5.7).
     ```

- **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 3 C compiler will normally differ from another compiler's code. See Section 6.5.6.

6.3.5 Using Nonstandard Library Routines

The GNX—Version 3 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. This is normally a safe assumption since it is unusual for a program to redefine (and thereby hide) these standard routines. In addition, the functions `abs`, `fabs`, and `ffabs` actually compile into in-line code and do not generate a procedure call at all.

Table 6-3. Recognized Library Routines

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

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 (`-O -Fl` on UNIX), which turns off the recognition of all library routines.

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 very 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. With this in mind, special care must be used when reviewing assembly code 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.
- Note that if symbolic information is requested, then slightly different code is generated. This happens because the `/FIXED_FRAME` optimizing flag (`-Of` on UNIX systems) is automatically disabled when the `/DEBUG` qualifier (`-g` 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.

6.5 ADDITIONAL GUIDELINES FOR IMPROVING CODE QUALITY

Using the following programming guidelines results in programs which take advantage of the GNX—Version 3 C compiler optimizations.

6.5.1 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 3 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.

6.5.2 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`.

6.5.3 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.

6.5.4 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 `ffabs()` instead of `abs()`, `fsin()` instead of `sin()`, etc.

6.5.5 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 values many 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. Location `*q` *may* be changed; therefore, it has 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/nonvolatile registers**
  By convention, the registers are divided into volatile registers (registers R0 through R2 and F0 through F3) and nonvolatile registers (registers R3 through R7 and F4 through F7). Volatile registers may be changed by a procedure call, whereas nonvolatile registers are guaranteed to retain their value across procedure calls. Therefore, all nonvolatile 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 to 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 because the 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 or her 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.

6.5.6 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 differs 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 before and after inserting `asm` statements into a program.

6.5.7 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 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`).

6.5.8 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.

6.5.9 Optimizing for Space

The default behavior of the GNX—Version 3 C compiler optimizes for optimal speed. There are several things that can be done to improve code density:

- optimize with the `/NOSPEED_OVER_SPACE` 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.
- squeeze all record definitions by using the `/ALIGN=1` (`-J1` on UNIX systems) switch. See Section 4.2.4.

6.6 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 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 a Memory Management Unit (MMU) 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.

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 neccessary), floating-point values use eight bytes and are represented as `long` values; structures (records) use a multiple of double-words.

NOTE: Stack alignment is maintained by all GNX—Version 3 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 `FIXED_FRAME` (`-OF`) 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 NS32381 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 NS32381) are used, their values should be saved (onto the stack or in temps) 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.

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

  A long floating-point value that returns from a function, returns in register pair F0-F1. A float-returning function returns the value in register F0.

  If a function returns a structure, 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 during execution of the return statement. Note that functions that return structures must be correctly declared as such, even if the return value is ignored.

Example:

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

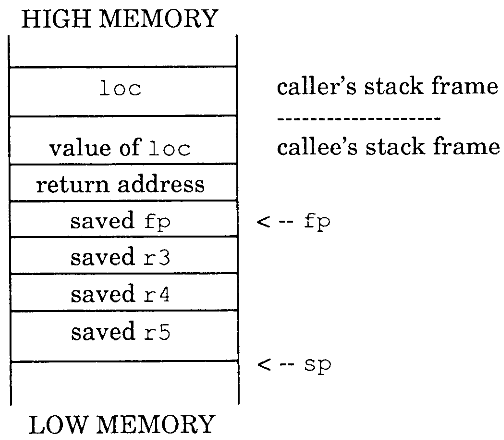
ifunc(p1)
int p1;
{
     int i, j, k;
     j = 0;
     for (i = 1; i <= p1; i++)
         j = j + f(i);
     return(j);
}
```

The compiler may generate the following code:

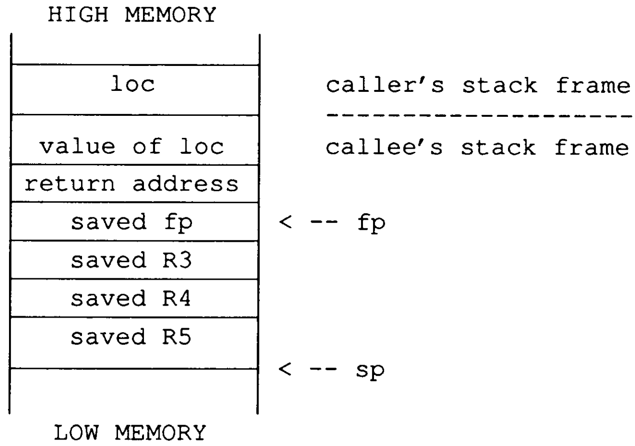
```
_m:
         enter    [],4          #Allocate local variable
         movd     -4(fp),tos    #Push argument
         bsr      _ifunc
         adjspb   $(-4)         #Pop argument off stack
         movd     r0,_iglob     #Save return value
         exit     []
         ret      $(0)

_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:



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 previously defined rules in the context of each possible cross-language pair.

	C	Pascal	FORTRAN 77	Modula-2	*Series 32000* **Assembly**
Series 32000 **Assembly**	'_' prefix	'_' prefix include ext case sensitivity	'_' prefix '_' suffix ref args case sensitivity	'_' prefix DEF & IMPORT init code	
Modula-2	DEF & IMPORT init code	DEF & IMPORT init code include ext case sensitivity	'_' suffix DEF & IMPORT init code ref args case sensitivity		'_' prefix DEF & IMPORT init code
FORTRAN 77	'_' suffix ref args case sensitivity	'_' suffix include ext ref args		'_' suffix DEF & IMPORT init code ref args case sensitivity	'_' prefix '_' suffix ref args case sensitivity
Pascal	include ext case sensitivity		'_' suffix include ext ref args	DEF & IMPORT init code include ext case sensitivity	'_' prefix include ext case sensitivity
C		include ext case sensitivity	'_' suffix ref args case sensitivity	DEF & IMPORT init code	'_' prefix

Figure B-1. Cross-Language Pairs

RULE 1 case sensitivity

This rule is of interest to every programmer who mixes programming languages.

Modula-2, C, and *Series 32000* assembly are case sensitive while FORTRAN 77 and Pascal are not (at least according to the standard). 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, Modula-2, or assembly code.

The FORTRAN 77 compiler appends an underscore to each high-level identifier name (in addition to the action described in RULE 1). 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, Modula-2, 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, Modula-2, 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.

Pascal and Modula-2 programs 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, Pascal, or Modula-2 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 must define these variables exactly once in an external header (include) file. The external header file has to be included in all Pascal source files which access the shared global variable, and its name must have a `.h` extension.

RULE 6 **DEF and IMPORT**

This rule is of interest to those who mix Modula-2 with other languages.

Modula-2 modules which access external symbols must import external symbols. If external symbols are not defined in Modula-2 modules but defined in other languages, the programmer must export these symbols to conform with the strict checks of the Modula-2 compiler.

External symbols can be exported by writing a "dummy" DEFINITION MODULE which exports all of the foreign language symbols, making them available to Modula-2 programs.

This export must be nonqualified to prevent the module name from being prepended to the symbol name.

RULE 7 **init code**

This rule is of interest to those who mix Modula-2 with other languages.

Modula-2 modules which import from external modules activate the initialization code of the imported modules before they start executing. The initialization code entry-point is identical to the imported module name.

To avoid getting an "Undefined symbol" message from the linker, the programmer should define a possibly empty, initialization function for every imported module. This is in case the implementation part of that

module is not written in Modula-2. It should be noted that the initialization code is not necessarily called during run-time. Initialization code is executed if, and only if, the following two conditions hold true:

1. The main program code is written in Modula-2.
2. The Modula-2 routines which are supposed to activate the intialization part are not called indirectly through some non-Modula-2 code.

In addition to these rules, a few points should be noted. First, GNX—Version 3 FORTRAN 77 allows identifiers longer than the six character maximum of traditional FORTRAN compilers. Second, the family of GNX—Version 3 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 (draft-proposed 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:
```
function func_ : integer ; external;
procedure proc_ ; external;
```
or
```
#include "var_def.h"
```

where the file `var_def.h` contains the following declaration:

```
var
        var_name_ : integer;
```

as explained in RULE 5 (include ext).

Modula-2:

```
FROM modula_name IMPORT func_
   or
FROM module_name IMPORT var_name_
```

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. Three points should be mentioned in conjunction with the successful linking and loading of programs. These are as follows:

- 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.

In some cases, a standard is not so widely accepted, as with Modula-2. In these cases, the user must be aware of the libraries that are available and the calling conventions of the main program used by the operating system.

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 C programs and `_MAIN__` in Pascal, FORTRAN 77, or Modula-2 programs.

Table B-1. Compilers and their Associated Libraries

COMPILER (DRIVER) NAME	LIBRARIES
cc (cross nmcc)	libc
f77 (cross nm77)	libF77, libI77, libm, libc
pc (cross nmpc)	libpas, libm, libc
m2c (cross nm2c)	libmod2, libm, libc

Note that the last three compilers completely ignore the user's main program name. Therefore, in C, the user-written code is called directly from `crt0.o`. In Pascal, FORTRAN 77, and Modula-2, `_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 (assember, linker, *etc.*) on UNIX systems relieve a user's concern about external libraries, initialization code, and entry-points. This is due to the coherency and consistency of the GNX—Version 3 compilers and their integration through the use of a common driver.

When using a GNX—Version 3 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.

As mentioned earlier, the driver program used by C, Pascal, FORTRAN 77, and Modula-2 programmers is exactly the same program on UNIX systems. The respective driver names are `cc`, `pc`, `f77`, and `m2c` (`nmcc`, `nmpc`, `nf77`, and `nm2c` for cross-support).

The driver program looks at its own name in order to determine the libraries that are 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` extension).

The -v (verbose) 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 five source modules written in five different languages (`c_utils.c` written in C, `f_utils.f` written in FORTRAN 77, `p_tuils.p` written in Pascal, `m_utils.m` written in Modula-2, and `s_utils.s` written in assembly), and there is a sixth 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 m_utils.m 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 dmod_fun.def dummy.def f77_fun.f \
        imod_fun.m 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
nm2c dmod_fun.def
nm2c dummy.def
nf77 f77_fun.f
nm2c imod_fun.m
nmpc pas_fun.p
nasm asm_fun.s
```

After 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, Modula-2 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 !!!\n";    /* 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 */
        mod_func (&d, 3)       &&   /* in Modula-2, at 0 */
        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 initialization function for Modula-2 */
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
```

dmod_fun.def

```
DEFINITION MODULE mfunc_module;
    EXPORT mod_func;
    PROCEDURE mod_func(VAR c: CHAR; index: INTEGER): BOOLEAN;
END mfunc_module.
```

dummy.def

```
(*
 *   This definition module was written in order to 'satisfy' Modula-2
 *   strict conformance checks regarding the foreign language functions
 *   and in order to define the global character array 'str[]'.
 *   The external functions are called from the Modula-2 main program,
 *   so they must be exported from somewhere...
 *)
DEFINITION MODULE dummy ;
    EXPORT
        str_, c_func, pas_func, f77_func_, asm_func;
    (* external function declarations *)
    PROCEDURE c_func    (VAR c : CHAR; index : INTEGER)  : BOOLEAN ;
    PROCEDURE pas_func  (VAR c : CHAR; index : INTEGER)  : BOOLEAN ;
    PROCEDURE f77_func_(VAR c : CHAR; VAR index : INTEGER)  : BOOLEAN ;
    PROCEDURE asm_func  (VAR c : CHAR; index : INTEGER)  : BOOLEAN ;
    VAR
        str_ : ARRAY [0..14] OF CHAR;
END dummy .
```

imod_fun.m

```
(*
 *   Definition of the Modula-2 function 'mod_func()'
 *)
IMPLEMENTATION MODULE mfunc_module;
    FROM dummy IMPORT str_;
    PROCEDURE mod_func(VAR c: CHAR; index: INTEGER): BOOLEAN;
    BEGIN
        c := str_[index];
        RETURN(TRUE);
    END mod_func;
END mfunc_module.
```

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;
```

asm_fun.s

```
#
#   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, demonstrate 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 MESSAGES

C.1 INTRODUCTION

The GNX—Version 3 C compiler error messages and warnings are listed in this appendix. Error diagnostics are divided into two categories: warnings and errors.

C.1.1 Warnings

An input which conforms to the language, but which the compiler suspects the programmer may not mean what is written, causes the compiler to invoke a warning message. These messages are intended for information only and do not affect the code generated. Warning messages can be disabled by the **/NOWARNING** option (`-w` on UNIX operating systems).

C.1.2 Errors

Errors are those which the compiler detects but does not know what the user's intention is. In this case, together with the error message, code generation is suspended.

C.2 Error Messages

A list of error and warning messages follows:

1. ERRORS

```
<name> undefined

BCD constant exceeds 6 characters

Compilation aborted, there may be more errors in the file

Ran out of memory

arguments not allowed in function declaration

array of functions is illegal

assignment of different structures

attempted to take address of a register

bad scalar initialization
```

cannot declare <name>(void type)

cannot initialize char/short member with address

cannot initialize extern or union

case not in switch

constant expected

constant too big for cross-compiler

declared argument <name> is missing

default not inside switch

division by 0

division by zero

duplicate case in switch, <val>

duplicate default in switch

empty character constant

field outside of structure

field too big

fortran declaration must apply to function

fortran function has wrong type

function declaration in bad context

function has illegal storage class

function illegal in structure or union

function returns illegal type

gcos BCD constant illegal

illegal break

illegal character: <value>(octal)

illegal class

illegal continue

illegal do control

illegal field size

illegal field type

illegal for control

illegal function

illegal hex constant

illegal if control

illegal indirection

illegal initialization

illegal lhs of assignment operator

illegal pointer subtraction

illegal register declaration

illegal storage class

illegal switch control

illegal type combination

illegal type (array of void)

illegal types in :

illegal use of field

illegal while control

illegal {

maximum string length exceeded

member of structure or union required

nesting too deep for initialization

newline in BCD constant

newline in string or char constant

no automatic aggregate initialization

non-constant case expression

nonunique name demands struct/union or struct/union pointer

null dimension

only const or volatile pointer modifiers

only one storage class allowed

operands of <op> have incompatible types

pointer required

ran out of hash tables

redeclaration of <name>

size is too big for pointer arithmetic

size too big

storage class illegal in cast

storage class illegal in structure or union

struct/union is illegal for <op>

structure reference must be addressable

structure too large

syntax error

too many characters in character constant

too many initializers

too many initializers for inner aggregate

too many local variables

too many post-increments/decrements in expression

type clash in conditional

unacceptable operand of &

undefined enumeration

undefined structure

undefined structure or union

undefined union

unexpected EOF

unknown size

void type illegal in expression

yacc stack overflow

zero-sized field

zero-sized storage allocation

zero-sized structure/union

2. ERRORS REGARDING THE -N OPTION

IR buffer full

dimension table overflow

identifier symbol table full

multi hash table full

nesting too deep

out of tree space

parameter stack overflow

switch table overflow

symbol table full

whiles, fors, etc. too deeply nested

whiles, fors, switches, etc. too deeply nested

3. WARNINGS

#ident string too long

<value> overflows destination

& before array or function: ignored

-I overrides -n

-g ignored with -Q

-n ignored with -Q

-p ignored with -Q

<char> is not an octal digit

<name> redefinition hides earlier one

Too many files, file <name> ignored

Took address of setjmp or longjmp, DO NOT OPTIMIZE

a function is declared as an argument

ambiguous assignment: assignment op taken

attempted to take address of a register

combination of different sized arrays

constant argument to NOT

constant in conditional context

empty array declaration

enumeration type clash, operator <op>

float constant cannot fit in float

floating point overflow/underflow

illegal member use: <name>

illegal pointer combination

illegal structure-pointer combination

illegal zero-sized structure/union member: <name>

integer overflow in truncation

loop not entered at top

meaningless comparison(0 always <= unsigned)

meaningless comparison(unsigned always >= 0)

meaningless comparison(unsigned always >=0)

missing " in # ident

missing closing " in # ident string

multiple 'const' ignored

multiple 'volatile' ignored

non-null byte ignored in string initializer

old-fashioned assignment operator

old-fashioned initialization: use =

precedence confusion possible: parenthesize!

questionable comparison - unsigned can never be negative.

questionable comparison - unsigneds can never be negative

redeclaration hides formal parameter <name>

redefined standard library routine <name>

sizeof returns 0

statement not reached

struct/union or struct/union pointer required

structure/union member must be named

undeclared initializer name <name>

zero or negative dimension size

zero-sized storage allocation

zero-sized structure/union member

Appendix D

COMPILER OPTIONS

D.1 INTRODUCTION

This appendix contains tables for quick reference. 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	Accept only standard C.
-A2	This option is useful only when compiling Modula-2 programs.
-A3	This option is useful only when compiling Modula-2 programs.
-a*flags*	This option is useful only when compiling FORTRAN 77, Modula-2 and Pascal programs.
-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.
-i*dir*	This option is useful only when compiling Modula-2 programs.
-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	This option is useful only when compiling FORTRAN 77 and Assembly language programs.
-n	Put C source lines as comments into assembly output file.
-N [*flags*]*nnn*	Change the size of the compiler's static tables from its default size to *nnn*. This is a FORTRAN 77 and C compiler option.

Table D-1. UNIX Operating System Options
Sheet 2 of 2

OPTION	FUNCTION
-O*flags*	Perform optimizations according to *flags*.
-p	Prepare profiling information for profiling.
-Q	Compile only, verify for syntax errors.
-R	Put all literal strings in read-only memory. This is a C compiler option.
-S	Do not assemble, leave assembly in *file*.s.
-T	This option is useful only 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*.
-w	Suppress warnings.
-w66	This option is useful only when compiling FORTRAN 77 programs.
-Zc	Use an alternate library. This is a C compiler option.

-X

Table D-2. VMS Operating System Compiler Options

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*].
/[NO]DEBUG	[do not] prepare symbolic debug information for debugger
/[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 Pascal source lines as comments into assembly output file.
/ROM_STRINGS	put all literal strings in read-only memory. This is a C compiler option.
/ALIGN[=*width*]	force alignment boundary within structs to *width*.
/[NO]WARNING	[do not] output warning diagnostics.
/[NO]STANDARD	[do not] use standard C.
/[NO]PRE_PROCESSOR	[do not] run the source code through the preprocessor.
/TABLE_SIZE[=*size*]	sets the size of the identifier table to *size*. This is a C and FORTRAN 77 compiler option.
/[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]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	Default type of variables is undefined.
-V	Print linker version information.
-x	Do not preserve local symbols in the symbol table.

INDEX

A

- -A 2-6
- Additional guidelines
  - asm statements 6-13
  - floating-point computations 6-10
  - improving code 6-10
  - integer variables 6-10
  - local variables 6-10
  - optimizing for space 6-14
  - pointer usage 6-11
  - register allocation 6-13
  - setjmp() 6-14
  - static functions 6-10
- /ALIGN 2-11
- Alignment 2-6, 2-11
- Allocate variables as standard 2-6, 2-11
- /ANNOTATE 2-11
- ANSI C language draft proposal 3-1
- Asm 3-5
- /ASM 2-10
- Asm statements 6-13
- Assembly program 2-2
- Audience 1-2
- AVAIL_SWAP 6-15

B

- Bitfields 3-2, 4-1

C

- -C 2-6
- -c 2-5
- C language extensions 1-3
- Calling sequence 4-9, 5-8, A-1
- Char 4-1
- CMDDIR 2-17
- Code generator 2-1, 2-2, 5-8
- Code portability 4-1, 6-5
- Coloring algorithm 5-7
- Command line 2-2
- .comment 3-6
- /COMMENT 2-12
- Common subexpression elimination 5-1, 5-5
- Common subexpressions 6-12
- Compilation options
  - UNIX 2-2, 2-4
  - VMS 2-9
- Compilation process 2-1
- Compilation time requirements 6-15
- Compile but do not link 2-5, 2-10
- Compile leaving assembly files 2-5, 2-10
- Compiler options
  - -A 2-6
  - /ALIGN 2-11
  - /ANNOTATE 2-11
  - /ASM 2-10
  - -C 2-6
  - -c 2-5
  - /COMMENT 2-12
  - -D 2-8
  - /DEBUG 2-10
  - /DEFINE 2-12
  - -E 2-8
  - -e 2-9
  - /ERROR 2-12
  - /EXPAND 2-13
  - -F 2-4
  - -f 2-8
  - -g 2-5
  - -I 2-8
  - /INCLUDE 2-13
  - -J 2-6
  - -K 2-7, 2-13
  - -l 2-9
  - -M 2-8
  - -m 2-7
  - /MODULAR 2-12
  - -N 2-7, 2-14
  - -n 2-6
  - -O 2-4, 6-3
  - -o 2-6
  - /OBJECT 2-10
  - /OPTIMIZE 2-10, 6-3
  - -P 2-9
  - -p 2-5
  - /PRE_PROCESSOR 2-12
  - /PROFILE 2-10
  - -Q 2-5
  - -R 2-6
  - -r 2-9
  - /ROM_STRINGS 2-11
  - -S 2-5
  - -s 2-9
  - /STANDARD 2-11
  - /TABLE_SIZE 2-11, 2-14
  - /TARGET 2-12
  - -U 2-9
  - -u 2-9
  - /UNDEFINE 2-13
  - -V 2-9
  - -v 2-7
  - /VERBOSE 2-11
  - -vn 2-7
  - /VN 2-11

-W 2-9
-w 2-6
/WARNING 2-11
-X 2-7
-x 2-9
-Z 2-7
Compiler structure 2-1
code generator 2-2
driver 2-1
front end 2-1
language parser 2-1
macro preprocessor 2-1
optimizer 2-1
Compiling mixed-language programs B-6
Compiling system code 6-6
Const 3-2, 3-4
Constant folding 5-1, 5-2
Copy propagation 5-2

D

-D 2-8
Data flow analysis 5-2
Dead code removal 5-1, 5-4
/DEBUG 2-10
Debugging of optimized code 6-9
Define 2-8, 2-12
/DEFINE 2-12
Define entry point 2-9
Driver program 2-1

E

-E 2-8
-e 2-9
Enumerated type 3-2
Environment variables 2-17
AVAIL_SWAP 6-15
/ERROR 2-12
Error Detection C-2
Error Messages C-2
Errors C-2
Executable filename 2-6
Executable program 2-2
/EXPAND 2-13
Extensions to structures 3-2
Extensions to the C language 3-1

F

-F 2-4
-f 2-8
Features 1-2
Filename conventions 2-3
Files 2-3
assembly 2-2
executable 2-2
object 2-2
Fixed frame 5-1, 5-8
Floating-point arithmetic 4-11
Floating-point computations 6-10
Floating-point constants 3-1
Floating-point emulation 2-8, 2-15
native cross support 2-16
native host 2-16
VAX/UNIX system 2-16
VAX/VMS system 2-17
Flow optimizations 5-1, 5-4
Front end 2-1

G

-g 2-5
Generate an error log file 2-12
Generate makefile dependencies 2-8
Generate modular code 2-7, 2-12
GTS
target setup 2-2
Guidelines on using the optimizer 6-1

I

-I 2-8
.ident 3-6
Imbed source lines as comments 2-6, 2-11
Implementation issues 4-1
Importing routines and variables B-5
/INCLUDE 2-13
INCLUDEPATH 2-17
Induction variable elimination 5-1, 5-6
Integer variables 6-10
Intermediate form 2-1
Internal compiler tables 2-14
Invocation syntax
UNIX 2-2
VMS 2-9

J

-J 2-6

K

-K 2-7
Keywords
asm 3-5
const 3-4
volatile 3-3

L

-l 2-9
Language parser 2-1
Leave comments in 2-6, 2-12
LIBPATH 2-17
Library routines 6-7
Linker 2-2, 2-3
compiler options passed to 2-9
Linker version 2-9
Literal strings in read-only memory 2-6, 2-11
Local variables 6-10
Longjmp() 6-14
Loop invariant code motion 5-1
Loop invariant expressions 5-6
Low-level interface 6-7
relying on frame structure 6-7
relying on register order 6-7
using asm statements 6-7

M

-M 2-8
-m 2-7
Macro preprocessor 2-1
Memory allocation 4-10
Memory layout optimizations 5-1, 5-9
Memory representation 4-1
Mixed-language programming 2-3, 4-9, B-1
Compilation on UNIX operating systems B-7
Compilation on VMS operating systems B-8
/MODULAR 2-12

N

-N 2-7, 2-14
-n 2-6
No local symbols in symbol table 2-9

O

-O 2-4, 6-3
/OBJECT 2-10
Object code program 2-2
Optimization flags 6-1
Optimization options default 6-3
Optimization options on the command line
UNIX systems 6-3
VMS systems 6-3
Optimization techniques 5-1
Optimize 2-4, 2-10
/OPTIMIZE 2-10, 6-3
Optimizer 2-1, 5-2
Optimizing for space 6-14
Options 2-4
alignment 2-6
Alignment 2-11
allocate variables as standard 2-6, 2-11
compile but do not link 2-5, 2-10
compile leaving assembly files 2-5, 2-10
debug information 2-5, 2-10
define 2-8, 2-12
define entry point 2-9
floating-point emulation 2-8
generate error log file 2-12
generate makefile dependencies 2-8
generate modular code 2-7, 2-12
imbed source lines as comments 2-6, 2-11
leave comments in 2-6, 2-12
linker version 2-9
no local symbols in symbol table 2-9
optimize 2-4, 2-10
pass options 2-9
pass to C preprocessor 2-12
profile information 2-5, 2-10
quick compilation 2-5
read-only memory 2-6, 2-11
redirect output to .i file 2-9
rename output file 2-6
retain relocation 2-9
run cpp only 2-8, 2-13
set identifier table size 2-7, 2-11
set target 2-7, 2-12, 2-13
show do not execute 2-7, 2-11
specify include file directory 2-8, 2-13
specify program library 2-9
strip 2-9
undefine 2-9, 2-13
undefine symbol in symbol table 2-9
use alternative library 2-7
use the m4 preprocessor 2-7
verbose 2-7, 2-11
warning diagnostics 2-6, 2-11
Order of evaluation 4-10
Overview 1-1

P

-P 2-9
-p 2-5
Partial redundancy 5-4
Partial redundancy elimination 5-1
Pass options to compilation phase 2-9
Pass source file to the C preprocessor 2-12
Pcc 3-1
Peephole optimizations 5-1, 5-8
Pointer usage 6-11
Portability 4-1, 6-5
Portable C compiler 3-1
Prepare debug information 2-5, 2-10
Prepare profile information 2-5, 2-10
Preprocessor 2-1
compiler options passed to 2-8
m4 2-7

macro 2-1, 2-8
/PRE_PROCESSOR 2-12
/PROFILE 2-10
Programming in other languages 4-9

Q

-Q 2-5
Quick compilation 2-5

R

-R 2-6
-r 2-6, 2-9
Recommended reference book 1-2
Redirect output to .i file 2-9
Redundant assignment elimination 5-1, 5-2
Register allocation 5-6, 6-13
Register allocation by coloring 5-1
Register parameters 5-7
Register variables 4-10
Registers
safe 5-7
scratch 5-7
Reliance on naive algebraic relations 6-8
Rename the output file 2-6
Retain relocation 2-9
Return value 4-11, 6-5, A-2
/ROM_STRINGS 2-11
Run cpp only 2-8, 2-13
Run-time library 6-7

S

-S 2-5
-s 2-9
Safe registers 5-7
Scratch registers 5-7
Set identifier table size 2-7, 2-11
Set target configuration 2-7, 2-12, 2-13
Setjmp() 6-14
Show, but do not execute 2-7, 2-11
Specify a program library 2-9
Specify directory for included files 2-8, 2-13
Speed over space 5-9
/STANDARD 2-11
Standard calling convention A-1
Static functions 6-10
Strength reduction 5-1, 5-6, 5-8
Strip 2-9
Structure returning function 4-9, 6-5
System code 6-6

T

/TABLE_SIZE 2-11, 2-14
/TARGET 2-12
Target setup 2-2
Timing assumptions 6-7
TMPDIR 2-17
Turning off optimization options 6-4
Type representations 4-1
Types and conversions 4-2

U

-U 2-9
-u 2-9
Undefine 2-9, 2-13
/UNDEFINE 2-13
Undefine symbol in symbol table 2-9
Undefined behavior 4-11
Undetected program errors 6-5
failing to declare a function 6-5
relying on memory allocation 6-5
uninitialized local variables 6-5
UNIX
invocation syntax 2-2
Unsigned constants 3-1
Use alternative library 2-7
Use the m4 preprocessor 2-7

V

-V 2-9
-v 2-7
Value propagation 5-1, 5-2
Variable and structure alignment 4-2
Verbose 2-7, 2-11
/VERBOSE 2-11
VMS
invocation syntax 2-9
-vn 2-7
/VN 2-11
Void 3-2
Volatile 3-2, 3-3
Volatile variables 6-6

W

-W 2-9
-w 2-6
/WARNING 2-11
Warning diagnostics 2-6, 2-11
Warnings C-2
Writing Mixed-Language Programs B-1

X

-X 2-7
-x 2-9

Z

-Z 2-7

National Semiconductor

MICROCOMPUTER SYSTEMS GROUP

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:

____

____

____

____

____

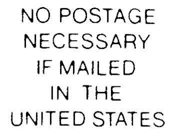
GNX — Version 3 C Optimizing Compiler Reference Manual

FOLD, STAPLE, AND MAIL

424010516-003B

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA



POSTAGE WILL BE PAID BY ADDRESSEE

National Semiconductor Corporation
Microcomputer Systems Division
Technical Publications Dept., M/S E265
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-9968

National Semiconductor

MICROCOMPUTER SYSTEMS DIVISION

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) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada 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 __________

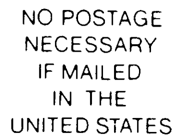
Do you require a response? ☐ Yes ☐ No PHONE ____________________

Comments:

GNX — Version 3 C Optimizing Compiler Reference Manual

FOLD, STAPLE, AND MAIL

424010516-003A



FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Software
Problem Report

Name: ______________________________
Street: ______________________________
City: ____________________ State: ______ Zip: __________
Phone: ____________________ Date: __________

Instructions

Use this form to report bugs, or suggested enhancements. Mail the form to National Semiconductor. Technical Support may be contacted at:

(800) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada only
((0)8141) 103-330 - West Germany

Category

☐ Software Problem ☐ Request For Software Enhancement
☐ Other ☐ Documentation Problem, Publication # __________

Software Description

National Semiconductor Product ______________________________
Version __________ Registration # __________
Host Computer Information ______________________________
Operating System ______________________________
Rev. __________ Supplier __________

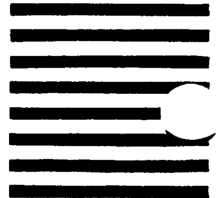
Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

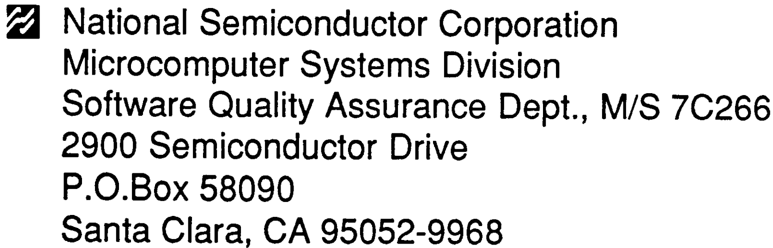
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS **PERMIT NO. 409** **SANTA CLARA, CA**



POSTAGE WILL BE PAID BY ADDRESSEE



National Semiconductor Corporation
Microcomputer Systems Division
Software Quality Assurance Dept., M/S 7C266
2900 Semiconductor Drive
P.O.Box 58090
Santa Clara, CA 95052-9968

National Semiconductor Use Only

Tech Support ________________ Date Received ________________

Software Q.A. ________________ Date Received ________________

Report Number ________________

Action Taken :

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 Piedias
(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

Series 32000®

GNX — Version 3
Linker User's Guide

Customer Order Number 424010506-003
NSC Publication Number 424010506-003A
August 1988

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
A	08/88	First Release. *Series 32000®* GNX – Version 3 Linker User's Guide NSC Publication Number 424010506-003A.

PREFACE

The GENIX™ Native and Cross-Support (GNX) Linker is an essential component of any *Series 32000®* microprocessor software development tool set.

The Linker can be used to quickly and easily create an executable file for any *Series 32000*-based native application or, via the facility of a powerful and flexible linker directives language, create an executable image for any *Series 32000*-based cross-development need.

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

No part of this document may be reproduced in any form or by any means without the prior written consent of National Semiconductor Corporation.

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

GENIX, NSX, ISE, ISE16, ISE32, SYS32, and TDS 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

Chapter 1 OVERVIEW

1.1 INTRODUCTION 1-1
1.2 MANUAL OVERVIEW 1-1
1.3 DOCUMENTATION CONVENTIONS 1-2
1.3.1 General Conventions 1-2
1.3.2 Conventions in Syntax Descriptions 1-2

Chapter 2 COMMAND LINE INVOCATION

2.1 INTRODUCTION 2-1
2.2 UNIX ENVIRONMENT INVOCATION 2-1
2.2.1 Library Path Environment Variable 2-2
2.3 VMS ENVIRONMENT INVOCATION 2-2
2.3.1 Logical Names for Library Specification 2-3
2.4 COMMAND LINE OPTIONS 2-3
2.4.1 Specify Library File 2-3
2.4.2 Specify Directory for Libraries 2-3
2.4.3 Request Output Memory Map 2-6
2.4.4 Specify Output Filename 2-6
2.4.5 Strip Symbolic Information 2-7
2.4.6 Strip Local Symbolic Information 2-7
2.4.7 Issue Warning for Defined Common Data 2-7
2.4.8 Suppress Size Warning for Common Data 2-8
2.4.9 Suppress Error Messages 2-8
2.4.10 Output Linker Version Information 2-8
2.4.11 Retain Relocation Information 2-9
2.4.12 Keep Relocation Information in Executable File 2-9
2.4.13 Specify Program Entry Point 2-9
2.4.14 Specify Fill Value for Section Holes 2-10
2.4.15 Specify Directives File 2-10
2.4.16 Specify Undefined Symbol 2-11
2.4.17 Specify Version Stamp 2-11

Chapter 3 LINKER DIRECTIVES LANGUAGE

3.1 INTRODUCTION 3-1
3.1.1 How the Directives Language is Used 3-1
3.2 SPECIFYING CONFIGURED MEMORY 3-1
3.3 INPUT FILE AND LIBRARY SPECIFICATION 3-2

3.3.1 Object Files 3-3
3.3.2 Library Files 3-3

3.4 SECTION DEFINITION DIRECTIVES 3-4
3.4.1 Input Files and Libraries 3-5
3.4.2 Input Sections 3-6
3.4.3 COMMON and MOD Input Sections 3-7
3.4.4 Binding a Section to a Memory Address 3-8
3.4.5 Aligning a Section to a Value 3-8
3.4.6 Blocking a Section to a File Address 3-9
3.4.7 Directing a Section to Memory by Name 3-9
3.4.8 Directing a Section to Memory by Attributes 3-10
3.4.9 Setting the Section Type Flags 3-11
3.4.10 Associating Module Names with Sections 3-11

3.5 GROUPING OUTPUT SECTIONS 3-12

3.6 CREATING HOLES WITHIN AN OUTPUT SECTION 3-14
3.6.1 Specifying the Fill Value of Section Holes 3-14

3.7 CREATING AND DEFINING SYMBOLS AT LINK TIME 3-15

3.8 SPECIFYING OUTPUT FILE OPTIONS 3-16
3.8.1 Native Output File Configuration 3-16
3.8.2 Optional Header Magic Number 3-16

3.9 COMMENTS IN A DIRECTIVES FILE 3-16

Chapter 4 DIRECTIVES LANGUAGE EXPRESSIONS

4.1 INTRODUCTION 4-1

4.2 VALID INTEGER SYNTAX 4-1
4.2.1 Decimal Value Syntax 4-1
4.2.2 Octal Value Syntax 4-1
4.2.3 Hexadecimal Value Syntax 4-1

4.3 UNARY OPERATIONS 4-2
4.3.1 Logical Negation 4-2
4.3.2 One´s Complement 4-2
4.3.3 Two´s Complement 4-3

4.4 BINARY OPERATIONS 4-3
4.4.1 Shift Operations 4-3
4.4.2 Relational Operations 4-3

4.5 ASSIGNMENT OPERATIONS 4-5
4.5.1 Current Location Assignments 4-5

4.6 SPECIAL FUNCTIONS 4-6
4.6.1 Sizeof Function 4-7
4.6.2 Memory Address Function 4-7
4.6.3 File Address Function 4-7
4.6.4 Next Address Function 4-8

4.6.5 Highest Memory Address Function 4-8

Chapter 5 Basic Linker Operations

5.1 INTRODUCTION 5-1

5.2 BASIC LINKER OPERATIONS 5-1

5.2.1 Linker Allocation Algorithm 5-2

5.3 OBJECT FILE FORMAT 5-4

5.3.1 COFF Sections 5-4

Appendix A DIRECTIVES LANGUAGE SYNTAX

A.1 SYNTAX A-1

Appendix B OUTPUT MAP

B.1 FORMAT DESCRIPTION B-1

Appendix C SECTION TYPE OPTIONS

Appendix D LINKER ERROR MESSAGES

D.1 INTRODUCTION D-1

D.2 WARNINGS D-1

D.3 ERRORS D-2

D.4 FATAL ERRORS D-2

D.5 INTERNAL ERRORS D-4

Appendix E SAMPLE LINKER DIRECTIVE FILES

E.1 CROSS APPLICATION E-1

E.2 NATIVE APPLICATION E-2

FIGURES

Figure 3-1. Group Link 3-13

Figure 5-1. Basic Linking Process 5-2

TABLES

Table 2-1. Unix Environment Command Line Options 2-4

Table 2-2. VMS Environment Command Line Options 2-5

Table 4-1. Unary Operators 4-2

Table 4-2. Binary Operators 4-4

Table 4-3. Special Functions 4-6

Table A-1. Syntax Diagram of Input Directives A-1

Table B-1. Linker Output Map B-1

Table C-1. Type Options C-1

INDEX

Chapter 1

OVERVIEW

1.1 INTRODUCTION

The GNX Linker is a language support tool used on *Series 32000*-based development systems to create executable files. Relocatable object files are inputs to the Linker produced by either a GNX compiler, assembler, or a previous linker run. To form either a relocatable or an executable object file, the Linker combines object files, performs relocation, and resolves external references.

The Linker operation is controlled by command line options and a powerful Linker directives language. Command line options and default linker actions vary depending on the host operating system.

1.2 MANUAL OVERVIEW

Chapter 2 explains Linker invocation, command line options, and related host-dependent issues for the UNIX® and VMS™ operating system environments.

Chapter 3 details the syntax and functionality of each Linker directive.

Chapter 4 describes the syntax and use of Linker expressions.

Chapter 5 describes the basic Linker operations controlled by the directives language and provides a brief overview of the common object file format (COFF) utilized by the Linker.

Appendix A contains a syntax diagram of input directives.

Appendix B contains a Linker output map.

Appendix C describes section type options.

Appendix D lists Linker error messages.

Appendix E contains a sample directives file.

The information presented in this manual is sufficient for a vast majority of applications, but familiarity with the contents of the *Series 32000 GNX—Version 3 COFF Programmer's Guide* and the *Series 32000 GNX—Version 3 Assembler Reference Manual* is advised.

1.3 DOCUMENTATION CONVENTIONS

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

1.3.1 GeneralConventions

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.

1.3.2 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 a syntactical term which may be repeated zero or more times.

[ ] Large brackets enclose an optional syntactical term or terms.

( ) Large parentheses enclose items which must be treated as a single syntactic term.

| Logical OR sign can separate items within large braces, large brackets, or large parentheses. A logical OR within large braces represents any possible sequence of the enclosed terms. A logical OR within large brackets requires the choice of either one term or none. A logical OR within large parentheses requires the choice of one term.

␣ 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.)

Chapter 2

COMMAND LINE INVOCATION

2.1 INTRODUCTION

This chapter explains Linker invocation, host-specific dependencies, and command line options for the UNIX and VMS operating system environments.

2.2 UNIX ENVIRONMENT INVOCATION

The Linker invocation name depends on the GNX software development host system. Currently, the Linker is available for several cross-environment host systems and several *Series 32000*-based host systems.

On a cross-development host, such as VAX™/UNIX, the Linker is named `nmeld`. On a *Series 32000*-based host system, such as VR32, SYS32/20, or SYS32/30, the Linker is named `ld`, the traditional name for a Linker on UNIX systems.

The Linker is invoked by specifying the appropriate Linker name followed by a list of command line arguments. These arguments specify a list of object and library files to link and optional linking actions and functions.

The invocation syntax for the Linker hosted on a *Series 32000*-based UNIX system is:

`ld` [{ *option* | *filename* }]

The invocation syntax for the Linker hosted on a cross-development (*i.e.*, VAX) UNIX system is:

`nmeld` [{ *option* | *filename* }]

Filename is any valid object file. Object filenames must be separated by a space. *Filename* must include a complete or relative pathname if the file is not in the user's current directory. *Option* is any valid Linker command line option. All options are preceded by a (—) dash, and options must be separated by a space.

The syntax does not specify a particular sequential preference between *filename* and *option*, but it is important to remember that libraries specified through the `-l` command line option and directives files specified through the `-d` command line option are processed as they are encountered.

Like most UNIX syntax, the Linker invocation syntax is case-sensitive.

The Linker is invoked either directly by entering the appropriate Linker name at the system command interface level, or indirectly via a call from a compiler driver. A compiler driver invokes the Linker with a predetermined set of Linker options. If the compiler driver-chosen Linker options are not suitable due to some special need, the compilation process must be terminated at the end of object file creation, and the Linker must be invoked directly. The direct Linker invocation includes the compilation object file as input along with the required special command line options.

2.2.1 Library Path Environment Variable

`LIBPATH` is the environment variable name which may be set to point to the directories containing libraries for linking. If `LIBPATH` is set, the Linker uses those directories as the location of libraries to search at link-time. If `LIBPATH` is not set, the Linker uses the default location. The default location for a UNIX-based cross-development environment (*e.g.*, VAX) is *gnxdir*`/lib`. The default locations for a UNIX *Series 32000*-based system (*i.e.*, SYS32/20) are *gnxdir*`/usr/lib`, *gnxdir*`/lib`, `/usr/lib`, and `/lib` (where *gnxdir* is the top-level directory of the installed GNX tools).

2.3 VMS ENVIRONMENT INVOCATION

The Linker is invoked by specifying the Linker name followed by a list of command line arguments. These arguments specify a list of object and library files to link and optional linking actions and functions.

The invocation syntax for the Linker hosted on a VMS system is:

nmeld [{ *option* | *filename* {,*filename*} }]

Filename is any valid object file or library filename. Object filenames and library filenames must be separated by a comma. *Filename*`.OBJ` may be entered as *filename*. In a list of filenames, object filenames must precede any library filenames that resolve external references. Object filenames must include a complete or relative pathname if the file is not in the user's current directory. Library filenames must include a complete path if the library is not in the user's current directory or if a VMS logical name is not defined to that library (see Section 2.3.1).

Option is any valid Linker command line option, and all options start with a slash (/).

The syntax does not specify a particular sequential preference between *filename* and *option*.

Like most VMS syntax, the Linker invocation syntax is not case sensitive. Case-sensitivity can be achieved by placing strings in double-quotes (" ").

2.3.1 Logical Names for Library Specification

`GNX$LIBRARY` and `GNX$LIBRARY_`*n* are VMS logical names that define libraries for linking. The definition must consist of the complete path and the library filename. The logical name definitions must start with `GNX$LIBRARY`, followed by the numbered `GNX$LIBRARY_`*n* definitions. The numbered logical name must start with 1 and proceed upwards in sequence. The upper limit for *n* is a system-dependent function.

When a logical name is defined for a library, it does not have to be explicitly mentioned on the command line. The Linker automatically processes these libraries at the end of the command line.

2.4 COMMAND LINE OPTIONS

This section describes the Linker command line options for both the UNIX and VMS environments.

Two tables provide an abbreviated syntax guide for the options. Table 2-1 lists the UNIX environment command line options. Table 2-2 lists the VMS environment command line options.

UNIX options begin with a dash (—) and the VMS options begin with a slash (/).

2.4.1 Specify Library File

The specify library file option is used only in the UNIX environment. (In the VMS environment, the library name is treated as another input object filename.) The command line option syntax for specifying a system library is

```
-lx
```

Where *x* is the sequence of up to nine characters from a UNIX system library name, `libx.a`. The `-lx` option must follow the list of object files with external references resolved in the library. The Linker searches for the library first in the directories specified through the `-L` command line option (see Section 2.4.2) and then in the default library locations. (Refer to Section 2.2.1.)

2.4.2 Specify Directory for Libraries

The specify directory for libraries option is used only in the UNIX environment. (In the VMS environment, the library name is treated as another input object filename.)

Table 2-1. Unix Environment Command Line Options

OPTION	EXPLANATION	SECTION
`-l`*x*	Specify a library file for linking	Section 2.4.1
`-L`*dir*	Specify a directory to search for libraries	Section 2.4.2
`-m`	Request an output memory map	Section 2.4.3
`-o` *filename*	Specify an output filename	Section 2.4.4
`-s`	Strip symbolic information	Section 2.4.5
`-x`	Strip local symbolic information	Section 2.4.6
`-M`	Issue warning for defined common data	Section 2.4.7
`-t`	Suppress size warning for common data	Section 2.4.8
`-S`	Suppress error messages	Section 2.4.9
`-V`	Output Linker version information	Section 2.4.10
`-r`	Retain relocation information	Section 2.4.11
`-k`	Keep relocation information in executable file	Section 2.4.12
`-e` *symbol*	Specify program entry point	Section 2.4.13
`-f` *int*	Specify fill value for section holes	Section 2.4.14
`-d` *filename*	Specify directives file	Section 2.4.15
`-u` *symbol*	Specify undefined symbol	Section 2.4.16
`-VS` *int*	Specify version stamp	Section 2.4.17

Table 2-2. VMS Environment Command Line Options

OPTION	EXPLANATION	SECTION
/MAP [=*filename*]	Request an output memory map	Section 2.4.3
/OUTPUT=*filename*	Specify an output filename	Section 2.4.4
/STRIP	Strip symbolic information	Section 2.4.5
/NOLOCAL	Strip local symbolic information	Section 2.4.6
/MULDEFS	Issue warning for defined common data	Section 2.4.7
/NOWARNING	Suppress size warning for common data	Section 2.4.8
/SILENT	Suppress error messages	Section 2.4.9
/VERSION	Output Linker version information	Section 2.4.10
/RETAIN	Retain relocation information	Section 2.4.11
/KEEP	Keep relocation information in executable file	Section 2.4.12
/ENTRY=*symbol*	Specify program entry point	Section 2.4.13
/FILL=*int*	Specify fill value for section holes	Section 2.4.14
/DIRECTIVES=*filename*	Specify directives file	Section 2.4.15
/USYM=*symbol*	Specify undefined symbol	Section 2.4.16
/STAMP=*int*	Specify version stamp	Section 2.4.17

The command line option syntax for specifying a library location other than the default is

```
-Ldir
```

Where *dir* is any valid directory pathname containing user libraries. For the `-L` option to be effective, it must precede any `-l` option. The Linker searches for libraries specified through the `-l` command line option first in *dir* and then in the default library locations (refer to Section 2.2.1).

2.4.3 Request Output Memory Map

The request output memory map option generates a memory map of the output file. The format and contents of the map are detailed in Appendix B. The command line option syntax to request a memory map is

-m	(UNIX)
/MAP [=*map_filename*]	(VMS)

In the UNIX environment, by default, the map is sent to standard output. Redirect standard output to create a map file.

In the VMS environment, by default, the map file is named *output_filename*.MAP, where *output_filename* is the name of the Linker output file. If *map_filename* is specified, the map file is named *map_filename*.MAP.

2.4.4 Specify Output Filename

The command line option syntax to specify the name of the Linker output file is

-o *filename*	(UNIX)
/OUTPUT=*filename*	(VMS)

Specifying the output filename overrides the default output filename.

In a UNIX environment, by default, the Linker output filename is `a32.out` on a cross-development system and `a.out` on a *Series 32000*-based system.

In a VMS environment, by default, the Linker output filename is *output_filename*.EXE, where *output_filename* is the name of the first filename encountered on the command line.

2.4.5 Strip Symbolic Information

The strip symbolic information option removes the symbol table and line number information from the output file and reduces the size of an executable file. The command line option syntax to strip symbolic information from the output file is

`-s`	(UNIX)
`/STRIP`	(VMS)

This option should be used only when the output of the Linker is an executable file.

2.4.6 Strip Local Symbolic Information

The strip local symbolic information option removes only the local symbolic information from the Linker output file, but still allows the output file to be used as input in a subsequent link command. This option is useful in reducing the size of an object file.

The command line option syntax to strip local symbolic information is

`-x`	(UNIX)
`/NOLOCAL`	(VMS)

2.4.7 Issue Warning for Defined Common Data

The term "common data" refers to variables declared by the `.comm` assembler directive, uninitialized variables declared outside of any procedure in a C program, and FORTRAN common data. The Linker treats common data by consolidating all references to a common variable and allocating space for it in the .bss section of the file.

If a program contains common data that is later associated with a section and defined (storage space allocated for it), the Linker uses the defined instance of the variable and does not allocate space in the .bss section.

The issue warning for defined common data option causes the Linker to issue a warning whenever a common variable is later defined in a program.

The command line option syntax to issue a warning for defined common data is

`-M`	(UNIX)
`/MULDEFS`	(VMS)

This warning message should not be confused with the multiply defined symbol error message. In the case of the multiply defined symbol error message, a symbol is defined more than once in source; this is always an error condition.

2.4.8 Suppress Size Warning for Common Data

The term common data refers to variables declared by the `.comm` assembler directive, uninitialized variables declared outside of any procedure in a C program, and FORTRAN common data. The linker treats common data specially by consolidating all references to a common variable and allocating space for it in the .bss section of the file.

If all the references to a common variable indicate data of the same size, the Linker performs the consolidation process quietly. But if the common variable references indicate data of different sizes, the Linker issues a warning message for every common variable declaration that is not the same size as the initial common variable declaration. Since this can result in a large number of warning messages, a command line option is provided to suppress these warnings.

The command line option syntax to suppress the size warning for common data is

`-t`	(UNIX)
`/NOWARNING`	(VMS)

2.4.9 Suppress Error Messages

A problem encountered during linking might result in several error messages before the Linker aborts. Only fatal errors stop the Linker. The user may request the Linker to work silently and suppress any nonfatal error messages.

The command line option syntax to suppress nonfatal error messages is

`-s`	(UNIX)
`/SILENT`	(VMS)

2.4.10 Output Linker Version Information

The output Linker version option produces information regarding the version and revision numbers of the Linker being executed. By default, this information is sent to the user's standard error location.

The command line option syntax to produce version information is

`-v`	(UNIX)
`/VERSION`	(VMS)

2.4.11 Retain Relocation Information

The retain relocation information option must be used if the output file is only partially linked (not all symbolic references resolved) and if the output file is meant for use as input in a subsequent link. The Linker retains relocation information and does not issue a fatal linking error for unresolved external references.

The command line option syntax to retain relocation information is

-r	(UNIX)
/RETAIN	(VMS)

2.4.12 Keep Relocation Information in Executable File

Relocation information is used to calculate the actual address of a data or routine reference. Normally, executable files do not have relocatable information since the final addresses have been calculated by the Linker. This option instructs the Linker to keep the relocation information in a fully relocated executable file. This may be useful on systems that implement dynamic (load-time) address relocations.

The command line option syntax to keep relocation information in an executable file is

-k	(UNIX)
/KEEP	(VMS)

2.4.13 Specify Program Entry Point

The entry point of a program is used by the system loader as the starting point for program execution. Entry point information is special information in the COFF object file and does not necessarily represent the actual beginning of the text (code) section.

By default, the Linker uses the symbol `start` as the entry point. If `start` is not found, the Linker uses symbol `_main` as the entry point. If `_main` is not found, the Linker sets the entry point to 0.

The command line option syntax to specify a program entry point is

-e *symbol*	(UNIX)
/ENTRY=*symbol*	(VMS)

To preserve case-sensitivity on VMS, *symbol* may be enclosed in double-quotes (" "). If *symbol* is not found, the Linker issues a warning and then sets the entry point by following the above mentioned default procedure.

2.4.14 Specify Fill Value for Section Holes

By default, the Linker fills in any holes created in output sections with a zero value. The command line option syntax to specify a fill value other than zero is

`-f` *int*	(UNIX)
`/FILL=`*int*	(VMS)

Int is a 16-bit value; therefore, desired fill values of one byte must be specified twice (*e.g.*, a desired fill value of 0xFF must be specified as 0xFFFF). See Section 4.2 for valid *int* syntax.

2.4.15 Specify Directives File

The specify directives file option specifies the directives file to use to create the Linker output file. If this option is specified, the Linker will not use the default Linker directives file.

The command line option syntax to specify a Linker directives file is

`-d` *filename*	(UNIX)
`/DIRECTIVES=`*filename*	(VMS)

Filename is any valid Linker directives file. *Filename* must include a complete path if the file is not in the user's current directory.

Because the Linker directives file is processed when it is encountered on the command line, this option should follow any input object files controlled by the directives. Specifically, the "*" directive applies only to input object files already seen and processed by the Linker.

If a directives file is not specified on the command line, the Linker attempts to use the directives file specified by the `LINKERFILE` parameter of the GNX Target Setup (gts) utility program. Refer to the *Series 32000 GNX—Version 3 Commands and Operations Manual* for a detailed discussion of the `gts` utility program. If `LINKERFILE` is undefined, the Linker uses the default directives file `linker.def` located in *gnxdir* on a cross-development system or *gnxdir*`/lib` in a *Series 32000* environment, where *gnxdir* is the top-level directory of the GNX tools. If `linker.def` does not exist, the Linker issues a warning message and then follows a predefined trivial link process. Applications should not depend on the trivial link process to produce meaningful results.

2.4.16 Specify Undefined Symbol

The Linker will link object files from libraries only if those object files resolve a currently undefined external symbol reference. Sometimes, however, it is desirable to link object files exclusively from a library. By creating an unresolved external reference to a symbol defined within a library, the user forces the Linker to process the appropriate object files within that library.

The command line option syntax to specify an undefined symbol is

-u *symbolname*	(UNIX)
/USYM=*symbolname*	(VMS)

To preserve case sensitivity on VMS, *symbolname* may be enclosed in double-quotes (" ").

2.4.17 Specify Version Stamp

A version stamp is a 16-bit decimal value used to identify an output file. The version stamp is stored in a special field in the optional header of the output file.

The command line option syntax to specify a version stamp value is

-VS *int*	(UNIX)
/STAMP=*int*	(VMS)

Chapter 3

LINKER DIRECTIVES LANGUAGE

3.1 INTRODUCTION

This chapter details the syntax and functionality of each directive in the Linker directives language. Chapter 4 describes the syntax and functionality of Linker expressions.

The directives language dictates the layout of the Linker output file through memory configuration, section allocation, and section content. The Linker directives language syntax is case-insensitive, except when reference is made to filenames or symbols. References to filenames or symbols must obey the rules of the host system (*i.e.*, a VAX/VMS host is case-insensitive, while a UNIX host is case-sensitive).

3.1.1 How the Directives Language is Used

The Linker requires a directives file to produce an output image that is significant to the system on which it will be executed. By default, the Linker looks for a directives file named `linker.def` in the top-level directory of the GNX tools on a cross-development system, or in the `/lib` directory of the GNX tools on a *Series 32000*-based system. The directives in the default file enable the linker to produce an executable image based on the host development system.

On a native host system, the default directives file produces an object file executable on the host system. On a cross-development system, the default directives file produces an object file executable on a *Series 32000* Development Board.

If the image created by the Linker is to be executed in an environment other than those mentioned above, any special loading requirements must be specified via Linker directives, and that directives file must be included as an argument on the Linker command line. Section 2.4 describes the actual command line syntax to specify a directives language file.

3.2 SPECIFYING CONFIGURED MEMORY

MEMORY directives are used to specify the configured and unconfigured areas of the virtual memory space and the total size of the virtual memory space of the target machine.

If a MEMORY directive is not specified, the Linker assumes the maximum amount of configured virtual address space, 0x0 through 0xFFFFFFFF.

If one or more MEMORY directives are specified, the Linker treats all virtual memory not mentioned in the directive as unconfigured. Unconfigured memory is not used in the Linker allocation process and, therefore, nothing can be assigned to an address within unconfigured memory. Configuration information is specified in a Linker directives file using the following syntax:

```
MEMORY
            {
            mem_name (attributes)  : ORIGIN = int , LENGTH = int
            }
```

The MEMORY directive declares one or more memory ranges. Multiple memory ranges may be separated by an optional comma. Memory ranges include a memory origin value, memory length in bytes, optional memory attributes, and a symbolic name for the declared range.

An area of memory is configured starting at address ORIGIN and containing LENGTH number of bytes. ORIGIN may be abbreviated to ORG, and LENGTH may be abbreviated to LEN. *Int* is any valid integer value. *Mem_name* is associated with the specified configured memory and may be referenced symbolically. Any symbolic reference to *mem_name* must follow the MEMORY directive which defines it. Any number of configured memory areas may be declared. If more than one memory area is declared, conflict (overlap) must not exist among configured memory areas. A memory conflict causes a Linker error message and then aborts the link process.

The optional memory range attributes, which must be enclosed in parentheses, may be any of the following single letters or combination of letters:

- I – The named memory range is initializable.
- R – The named memory range is readable.
- W – The named memory range is writable.
- X – The named memory range is executable.

3.3 INPUT FILE AND LIBRARY SPECIFICATION

In addition to specifying input object files and libraries on the Linker command line, object files and libraries may be included in a Linker directives file.

3.3.1 Object Files

Input object filenames may appear anywhere outside the curly braces of a MEMORY or SECTIONS directive. The placement is significant because the Linker processes these input files as they are encountered. Specifying input files this way is identical to specifying input files on the Linker command line. For example:

```
          .
          .
          .
my_input_filename_1
my_input_filename_2
my_input_filename_3
          .
          .
          .
MEMORY
          {
          my_mem : ORG = 0x0,LEN = 0x100
          }
```

All three input files are processed by the Linker in the order in which they are seen in the directives file. *Filename* is any valid input object filename. The filename may include a full or partial pathname. A filename containing special characters may be enclosed in double-quotes (" ") to avoid conflict with Linker directives language syntax.

Section 3.4.1 describes how input files can be specified within a SECTIONS directive to force input file contents to be placed in a particular output section.

3.3.2 Library Files

Library files may appear anywhere outside the curly braces of a MEMORY or SECTIONS directive. The placement is significant because the Linker processes these library files as they are encountered. Specifying library files this way is identical to specifying library files on the linker command line. Library files are processed only once — at the time they are encountered. The symbol information within the library file is processed as many times as is necessary to resolve currently undefined external references. For this reason, the ordering of object files within a library file is not significant for anything other than efficiency. For example:

```
            .
            .
my_input_filename_1
my_input_filename_2
my_input_filename_3
my_library_1
            .
            .
```

After the three input object files are processed, the library my_library_1 is searched to resolve any currently undefined external references. Only those library members containing symbol definitions which resolve external references are processed by the Linker and included in the Linker output.

A library name is any valid input library filename and may include a full or partial pathname. A library name containing special characters may be enclosed in double-quotes ("") to avoid conflict with Linker directives language syntax. Section 3.4.1 describes how library files can be specified within a SECTIONS directive to force library contents to be placed in a particular output section.

3.4 SECTION DEFINITION DIRECTIVES

Section definition directives direct input sections into output sections, assign memory address boundaries and memory start values to sections, and control the ordering of output sections.

A typical SECTIONS definition directive is

```
SECTIONS {
        secname1 : {
                input file,library,and section specifications,
                assignment statements
        }
        secname2 : {
                input file,library,and section specifications
                assignment statements
        }
        .
        .
        .
}
```

By default, if SECTIONS definition directives are not specified, the Linker associates input sections with output sections by name. For example, if two input files are linked containing input sections .text, .data and .bss, the output file will also contain the three sections .text, .data and .bss which are composed of the input section contents in the order they have been seen and processed by the Linker.

Details of the various input file and section specifications allowed in a SECTIONS directive are included in the remaining sections of this chapter. The use of assignment statements within a SECTIONS definition directive is detailed in Section 3.6.

3.4.1 Input Files and Libraries

A filename within the curly braces of an output section specification causes the Linker to place all sections from the named file into the specified output section. Specification of a filename causes the Linker to read and process that file, if it has not already done so.

Filename is any valid input object filename. The filename may include a full or partial pathname. A filename containing special characters may be enclosed in double-quotes (" ") to avoid conflict with Linker directives language syntax.

An example of using an input filename in an output section specification is as follows:

```
.text : { myfile1, myfile2 }
```

All sections in object files myfile1 and myfile2 are directed into the .text output section.

Similarly, a library name may be specified in an output section specification, causing any sections pulled in from the library for resolution of symbolic references to be directed into the specified output section.

A library name is any valid input library filename. The library name may include a full or partial pathname. A library name containing special characters may be enclosed in double-quotes (" ") to avoid conflict with Linker directives language syntax.

An example of using a library name in an output section specification is as follows:

```
.text : { mylibrary }
```

All input sections extracted from mylibrary to resolve current external references will be placed in the .text output section. A .data or .bss section from mylibrary may be placed in the .text output section.

3.4.2 Input Sections

Input sections may come from the object files specified on the Linker command line or from object files specified in the directives file. Input sections may be specified in the `SECTIONS` directive in several ways. The input sections from a particular input file may be explicitly referenced by stating the actual filename, or input sections from groups of files may be implicitly referenced using a wild-card character for filenames.

A particular section of an input file may be specified by enclosing the section name in parentheses immediately following the filename. An example of a specific filename and section specification is

```
SECTIONS
          {
          .text : { file1 (.text) }
          .data : { file2 (.data) }
          }
```

The .text section from `file1` is directed into the .text output section, and the .data section from `file2` is directed into the .data output section. Assuming both input files contained .text and .data sections, the Linker will then place the .text section from `file2` into the .text output section and the .data section from `file1` into the .data output section. This results from the Linker's default action, which directs all unallocated input sections to output sections of like names.

The following example demonstrates the use of the wild-card character:

```
SECTIONS
          {
          .text : { * (.text) }
          .data : { * (.data) }
          .bss  : { * (.bss) }
          }
```

The .text output section contains all of the .text sections from all of the input object files processed by the Linker. Likewise, the .data and .bss output sections contain the .data and .bss sections from all the input object files, respectively. Wild-cards are matched only against files already processed by the Linker. For example, files listed on the UNIX command line after the directives file will not be matched.

Section names may also refer to user-defined sections. User-defined sections have limited use. User-defined section names are typically used to create an output section only when linking with the retain command line option. The resulting output file is used as input for a subsequent link at which point the user-defined section is directed into one of the normal COFF sections. User-defined section names should only be used if the system loader supports nonstandard section names.

This process involves at least two invocations of the Linker. The first invocation directs the COFF sections (.text, .data, etc.) from a compilation or assembly object file into a user-defined output. The SECTIONS directive looks like this:

```
SECTIONS
            {
            my_output_section1: { *(.text) }
            my_output_section2: { *(.data) }
            }
```

This link does not produce an executable image; the Linker needs to be invoked with the retain command line option to make the output suitable as input for a subsequent link. The output file contains two sections, my_output_section1 and my_output_section2.

The second Linker invocation uses, as input, the output file from the first link plus any additional necessary object files. The following SECTIONS directive directs the user-defined section to a standard System V COFF section:

```
SECTIONS
            {
            .text : { my_object_file(my_output_section1), * (.text) }
            .data : { my_object_file(my_output_section2), * (.data) }
            }
```

The final output is executable or useful under a strict COFF convention environment because there are no more user-defined sections. The remainder of the manual uses only COFF section names.

3.4.3 COMMON and MOD Input Sections

The [COMMON] and [MOD] input section specifications refer to input sections containing common data and undefined module symbols, respectively.

Common data refers to variables declared by the .comm assembler directive, uninitialized variables declared outside of any procedure in a C program, and FORTRAN common data. The Linker consolidates all references to a common variable and allocates space for it in the .bss section. For example,

```
.data : { myfile1 [COMMON] }
```

The common data from myfile1 is directed into the .data output section rather than the default .bss output section.

The .module assembler directive associates a symbolic module name with an object module. The module symbol may be undefined under certain circumstances. By default, the Linker associates these undefined module symbols with the .mod output

section. The [MOD] option provides a way to associate undefined module symbols with output sections other than .mod. For example,

```
.data : { myfile1 [MOD] }
```

The undefined module symbols from `myfile1` are directed into the .data output section rather than the default .mod output section. Refer to the *Series 32000 GNX – Version 3 Language Tools Technical Notes* for information on *Series 32000* modularity support.

3.4.4 Binding a Section to a Memory Address

Any output section may be bound to a particular address in memory. The `BIND` directive instructs the Linker to assign a particular configured memory address to the output section. The following example demonstrates the use of binding:

```
SECTIONS
            {
            .text BIND (0x100) : { * (.text) }
            .data BIND (0xf000): { * (.data) }
            }
```

The Linker places all the .text sections from all of the input files into a .text output section and assigns it a memory address of 0x100. Similarly, the .data sections from all of the input files are placed in a .data output section and assigned a memory address of 0xF000.

The bind value must be within available configured memory. The Linker will report an error message if the specified bind address is not in available configured memory or if there is insufficient configured memory to hold the output section.

Binding an output section to a particular memory address overrides aligning the section or directing the section to a named configured memory region. A warning is generated.

3.4.5 Aligning a Section to a Value

Aligning an output section to a value ensures that the output section will be assigned a memory address that is a multiple of the `ALIGN` value. The align option may be specified in the output section specification of a `SECTIONS` directive.

An example of aligning an output section is

```
.text ALIGN (0x100) : { *(.text) }
```

The .text output section begins at the first available memory address which is a multiple of 0x100. If the previous output section ends at 0x101, the .text output section begins at 0x200.

Aligning an output section and binding an output section are mutually exclusive operations.

3.4.6 Blocking a Section to a File Address

Blocking an output section to a file address ensures that section will reside at the specified address in the Linker output file. The block option may be specified in the output section specification of a `SECTIONS` directive. An example of blocking a section to a file address follows:

```
.text BLOCK (5000) : { *(.text) }
```

The .text output section will reside at address 5000 in the Linker output file.

The block option does not affect binding or aligning; it may be specified along with `BIND` or `ALIGN`. For example:

```
.text BLOCK (0x300) BIND (0x2000): { *(.text) }
```

The .text output section starts at byte 0x300 in the output file, but its memory address is 0x2000.

The specified blocking address must be greater than the size of the COFF header information, since header information always resides at the beginning of an output file. Blocking to an address which overlaps COFF header information results in a Linker error message. Blocking addresses must be used in ascending order (*i.e.*, the first block address must be less than any subsequent block address; the second block address must be greater than the first but less than any subsequent block address, etc.). Blocking may cause gaps to appear within the output file.

3.4.7 Directing a Section to Memory by Name

The `MEMORY` directive declares a specific configured memory area and associates that memory with a symbolic name. The output sections defined within a `SECTIONS` directive may be routed to a particular memory area by simple symbolic association. For example:

```
MEMORY
          {
          my_mem1 : ORIGIN = 0x0 , LENGTH = 0x100
          my_mem2 : ORIGIN = 0x100 , LENGTH = 0xfff
          }

SECTIONS
          {
          .text : { * (.text) } >my_mem1
          .data : { * (.data) } >my_mem2
          }
```

The redirection symbol ">" directs the Linker to assign the output section to a memory location within the named memory area. In the above example, the Linker assigns the .text output section a memory address of 0x0 and the .data section a memory address of 0x100. The memory address assigned to an output section that is directed to a particular named memory area is allocated on a first-fit basis within the memory area. An error is generated if the size of the output section exceeds the amount of available configured memory in the named memory area.

3.4.8 Directing a Section to Memory by Attributes

A configured memory area declared with the MEMORY directive may optionally have an associated set of memory attributes. An output section may be redirected to a memory area specified by *name* as well as to a memory area of specific memory attributes. For example:

```
MEMORY
          {
          my_mem1 (RW) : ORIGIN = 0x0 , LENGTH = 0x100
          my_mem2 (R) : ORIGIN = 0x100 , LENGTH = 0xfff
          }

SECTIONS
          {
          .text : { * (.text) } >(R)
          .data : { * (.data) } >(RW)
          }
```

The redirection symbol ">" directs the Linker to assign the output section a memory address within a memory area of the specified attributes. In the above example, the Linker assigns the .text output section a memory address of 0x100 and the .data section a memory address of 0x0, based on the attributes of each memory area. The memory address assigned to an output section that is directed to a memory area of specific attributes is allocated on a first-fit basis within the memory area. An error is generated if the size of the output section exceeds the amount of available configured memory in that memory area.

3.4.9 Setting the Section Type Flags

The type of a section is information stored in the section header of the COFF file to indicate how the section is to be handled by the Linker and loader and what category of data is contained within the section. By default, the Linker determines the type of an output section and its contents based on the input sections comprising it. Two options are available to override the default action and set the type explicitly. Appendix C details the effects of each particular output section type. The type that controls how the output section is processed may be any one of the following:

```
DSECT | NOLOAD | COPY | INFO | OVERLAY | LIB
```

The type option that specifies the contents of an output section may be any one of the following:

```
TYP_TEXT | TYP_DATA | TYP_LINK | TYP_BSS | TYP_MOD
```

An example of using the type option is

```
.text (INFO)(TYP_TEXT) : {
    *(.text), *(.data) }
```

The .text output section is processed as an INFO section, and the flags are set to indicate the section contains executable text.

3.4.10 Associating Module Names with Sections

Module names are declared with the `.module` assembler directive and are associated with at least a single source file.

Input sections can be combined only if they are associated with the same module or associated with no module at all.

The MODULE option associates a module name with an output section; therefore, only input sections of the same module or input sections with no module can be directed into the output section.

An example of using the `MODULE` directive is

```
.text MODULE(my_mod) : { .text }
```

The .text output section contains only .text input sections with the `my_mod` module name and .text sections with no module name. Refer to the *Series 32000 GNX—Version 3 Language Tools Technical Notes* for information on *Series 32000* modularity support.

The module option may be specified along with any other output section option.

3.5 GROUPING OUTPUT SECTIONS

A group of output sections declared with the GROUP directive is guaranteed to reside at contiguous addresses in memory and appear in the same order as they are presented within the GROUP directive.

The GROUP directive must appear within a SECTIONS directive. An example of using the GROUP directive follows:

```
SECTIONS {
    .text : { *(.text) }
    GROUP {
         .data : { *(.data) }
         .bss  : { *(.bss) }
    }
}
```

In this example, the .data and .bss output sections are guaranteed to reside at contiguous areas in memory.

Because the GROUP directive forces output sections to be processed together, options such as BIND, ALIGN, and BLOCK must be used with the entire group of output sections and not with the individual sections themselves. For example:

```
SECTIONS
             {
             .text BIND (0x1000) : { * (.text) }

GROUP BIND (0xa000) :
             {
             .data : { * (.data) }
             .bss  : { * (.bss) }
             }
             }
```

The .text output section is bound at 0x1000; the data output section starts at 0xa000 and is followed immediately by the .bss output section. Figure 3-1 illustrates this example.

The Linker will issue an error message if a group of sections is bound to an address without enough available configured memory for the entire group of output sections.

A group of output sections may be bound or aligned to the memory address, blocked to a file address, or redirected to a particular memory area by name or memory attributes.

An individual output section within a GROUP directive may be specified with the type option only. The six different *type_options* are listed in Section 3.4.9.

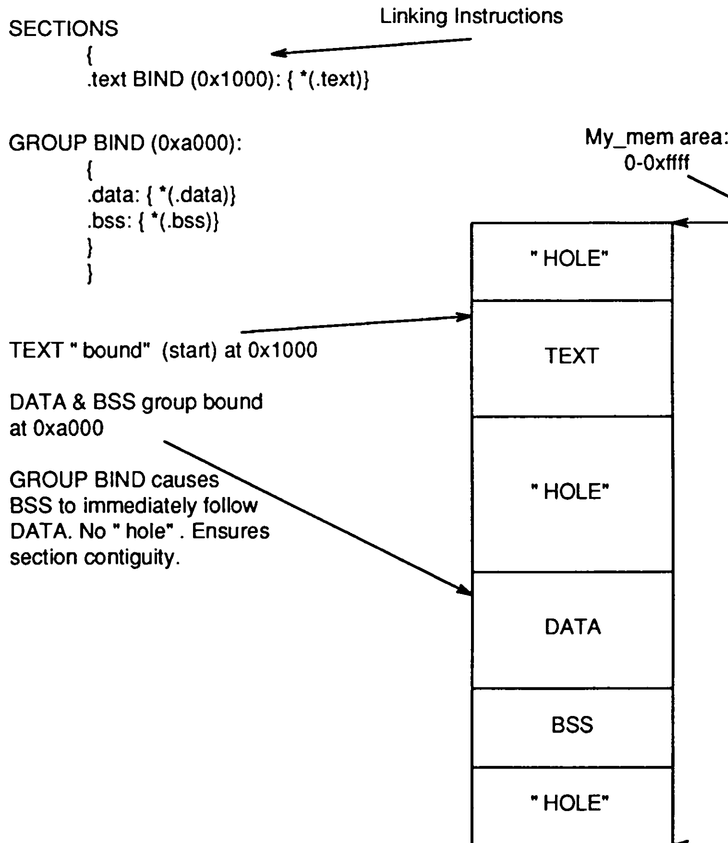


Figure 3-1. Group Link

3.6 CREATING HOLES WITHIN AN OUTPUT SECTION

The Linker normally combines input sections in a contiguous fashion when creating an output section. That is, if an output section is comprised of three input sections, the raw data from those input sections are placed one immediately after the other in the output section. It is possible to create a "hole" in the output section through the use of a current location (.) assignment. The current location denotes the address in memory to which the Linker is allocating sections. Changing the value of the current location creates a gap of unallocated space. The current location can never be decremented. Because the concept of current memory location is valid only during the allocation phase of the link process, current location assignments must always appear within a `SECTIONS` definition directive. An example of a current location assignment to create a hole in an output section is

```
SECTIONS {
        .text : {  *(.text) ,
                   . += 100; }
}
```

The .text output section contains the .text input sections from all the input object files, followed by a 100-byte hole. Similarly, the following example creates a 20-byte hole between the .data section from `file1` and the .data section from `file2` within the .data output section.

```
SECTIONS {
        .data : {  file1(.data) ,
                   . += 20;
                   file2(.data) }
}
```

3.6.1 Specifying the Fill Value of Section Holes

By default, the Linker fills the holes created within an output section with zeros. That is, when the Linker is writing out the raw data of an output section to the output file, zeros are written out for unallocated areas of memory. The following syntax is used to specify a fill value other than zero for any specific output section:

```
SECTIONS {
        .text : {  *(.text) ,
                   . += 100; } = 0xFFFF
}
```

In this case, the 100-byte "hole" in output section `.text` will be filled with the value 0xFFFF.

The Linker command line option to specify a fill value can be used when a fill value other than the default is desired for all output section holes. Specifying a fill value within a `SECTIONS` directive overrides the fill value specified on the Linker command line.

3.7 CREATING AND DEFINING SYMBOLS AT LINK TIME

A symbol may be assigned an absolute address at link time through the use of assignment statements.

If the symbol has not already been declared in an input object file, the Linker will create an external symbol table entry and assign it the specified absolute address.

If the symbol has already been declared in an input object file but remains undefined, the Linker assigns the symbol the specified absolute address.

If the symbol exists and is already defined as an absolute address, the Linker will redefine the symbol with the new absolute address. If the symbol exists and is defined within an input object file as a relocatable address, the Linker issues an error message and terminates.

The basic syntax for symbol assignment is

symbol = *int* ;

The syntax supports a wide range of operations in addition to "=" and supports expressions in place of *int*. This syntax usage is explained in detail in Chapter 4.

Care should be used in symbol address assignments to prevent excessive memory fragmentation or cause conflict with MEMORY, SECTIONS, or GROUP memory mappings. Symbol assignment address conflicts do not cause link-time error messages.

Because symbolic assignments are made at the end of the Linker allocation phase, the placement of the assignment statement within a Linker directive file is not important. The assignment may be made anywhere within the directive file.

An example of using an assignment statement to assign an absolute address is

```
.
.
.
SECTION {
  .text : { * (.text)}
}
special_sym = 0x1000;
```

Special_sym will be assigned the absolute address of 0x1000.

3.8 SPECIFYING OUTPUT FILE OPTIONS

Two output file characteristics may be controlled by Linker directives. These directives specify the default output filename and set the optional header magic number.

3.8.1 Native Output File Configuration

By default, the Linker produces an output file named `a32.out` that does not have execute file permissions set. To override this default, the following statement can be included anywhere within the Linker directives file:

```
OPTION NATIVE
```

This results in a Linker output file named `a.out` with execution file permissions set. This option is necessary when producing executable files on a *Series 32000*-based system.

3.8.2 Optional Header Magic Number

The optional header magic number in a COFF file provides memory loading information to the operating system or loader. The syntax to set the optional header magic number is

OPTION OMAGIC *int*

This statement may appear anywhere within a Linker directives file. All default Linker directives files explicitly set the optional header magic number for that particular execution environment.

3.9 COMMENTS IN A DIRECTIVES FILE

Comments may be used to document the purpose of the directives file. Comments begin with a slash and asterisk (/*) followed by one or more lines of comment text. The comment is terminated with a asterisk and slash (*/). The comment can either appear alone on a line or follow a Linker directive statement.

Chapter 4

DIRECTIVES LANGUAGE EXPRESSIONS

4.1 INTRODUCTION

Expressions can be used in two places in the Linker directives language:

- As arguments to the `BIND`, `ALIGN`, `BLOCK` and `NEXT` directives.
- On the right-hand side of an assignment statement.

The simplest form of an expression is a single integer term (*int*), while the more complex expression is a sequence of integer terms and/or special function terms separated by operands and may include nesting.

4.2 VALID INTEGER SYNTAX

The Linker accepts three radices for unsigned integer input: decimal (the default), hexadecimal, and octal. Integer input in the Linker directives language is denoted by the word `int`. Unless otherwise noted, `int` represents a 32-bit unsigned long integer value.

4.2.1 Decimal Value Syntax

A decimal value begins with a digit in the range of 1 through 9 followed by optional digits in the range of 0 through 9.

4.2.2 Octal Value Syntax

An octal value begins with 0 followed by digits in the range of 0 through 7.

4.2.3 Hexadecimal Value Syntax

A hexadecimal value begins with either `0x` or `0X` followed by digits in the range of 0 through 9, and/or letters in the range of A through F (either upper- or lower-case).

4.3 UNARY OPERATIONS

A unary expression consists of a single term, which can be either an integer, a symbol name, or a special function. Single-term expressions may be preceded by one of three valid unary operators: logical negation, one's complement, and two's complement.

A unary operation has the highest precedence in expression evaluation and is always performed before any other operation. Table 4-1 lists the unary operators.

Table 4-1. Unary Operators

OPERATOR	FUNCTION
!	Logical negation
~	One's complement
-	Two's complement

4.3.1 Logical Negation

The logical negation operation changes any nonzero value to zero, or changes a zero value to one. An example of logical negation follows:

```
!0xffffffe0
```

The logical negation operation changes the integer value in the example to zero.

4.3.2 One's Complement

The one's complement operation is a bit-wise logical negation. All one bits are set to zero, and all zero bits are set to one. An example of one's complement operation follows:

```
~0xffffffe0
```

The one's complement operation changes the integer value in the example to 0x1f.

4.3.3 Two's Complement

The two's complement operation is the result of a bit-wise logical negation plus one. All one bits are set to zero, all zero bits are set to one, and then one is added. An example of two's complement operation is

```
-0xffffffe0
```

The two's complement operation changes the integer value in the example to 0x20.

4.4 BINARY OPERATIONS

The Linker supports the following binary operators (listed in order of precedence, highest to lowest):

- * (multiplication), / (division), and % (modulus)
- + (addition) and - (subtraction)
- >> (right shift) and << (left shift)
- > (greater than), < (less than), >= (greater than or equal), and <= (less than or equal)
- == (equal) and != (not equal)
- & and | (bitwise AND and bitwise OR)
- && and || (logical AND and logical OR)

Table 4-2 lists the binary operators.

4.4.1 Shift Operations

The shift operators perform a shift on the left unsigned operand by the number of bits specified by the right operand. For example:

```
X << 2
```

The value of *X* is shifted two bits to the left. If *X* is one, its value is 4 after the shift. Vacated bits of right and left shifts are zero-filled.

4.4.2 Relational Operations

The result of a relational operation is either zero (false) or one (true). The Boolean result of a relational operation is useful in constructing expressions which can control dependencies. For example:

```
.text BIND( (sizeof(.data) >= 0x100) * 0x100 + 0x100)
```

Table 4-2. Binary Operators

<table>
<tr><th>OPERATOR</th><th>FUNCTION</th><th>PRECEDENCE (Highest = 1)</th></tr>
<tr><td>*</td><td>Multiplication</td><td>1</td></tr>
<tr><td>/</td><td>Division</td><td>1</td></tr>
<tr><td>%</td><td>Modulus</td><td>1</td></tr>
<tr><td>+</td><td>Addition</td><td>2</td></tr>
<tr><td>-</td><td>Subtraction</td><td>2</td></tr>
<tr><td>>></td><td>Shift right</td><td>3</td></tr>
<tr><td><<</td><td>Shift left</td><td>3</td></tr>
<tr><td>></td><td>Greater than</td><td>4</td></tr>
<tr><td><</td><td>Less than</td><td>4</td></tr>
<tr><td>>=</td><td>Greater than and equal</td><td>4</td></tr>
<tr><td><=</td><td>Less than and equal</td><td>4</td></tr>
<tr><td>==</td><td>Equal</td><td>5</td></tr>
<tr><td>!=</td><td>Not equal</td><td>5</td></tr>
<tr><td>&</td><td>Bitwise AND</td><td>6</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>6</td></tr>
<tr><td>&&</td><td>Logical AND</td><td>7</td></tr>
<tr><td>| |</td><td>Logical OR</td><td>7</td></tr>
</table>

If the size of the data section is greater than or equal to 0x100, the relational operation returns a one, and the resulting multiplication yields 0x100 which is added to 0x100. In this case, the bind address is 0x200.

If the size of the data section is less than 0x100, the relational operation returns a zero, and the resulting multiplication yields zero which is added to 0x100. In this case, the bind address is 0x100.

4.5 ASSIGNMENT OPERATIONS

The value of an expression may be assigned to a symbol in one of five ways:

`symbol = ` *expr*;	(assign the value of *expr* to `symbol`)
`symbol += ` *expr*;	(equivalent to: `symbol = symbol +` *expr*)
`symbol -= ` *expr*;	(equivalent to: `symbol = symbol -` *expr*)
`symbol *= ` *expr*;	(equivalent to: `symbol = symbol *` *expr*)
`symbol /= ` *expr*;	(equivalent to: `symbol = symbol /` *expr*)

The assignment syntax always requires a semicolon after the expression.

4.5.1 Current Location Assignments

The current location assignment is a special type of assignment which may be used only within the input section specification of a `SECTIONS` directive. The current location term (.) denotes the current allocation address in memory.

Symbols may be assigned the value of the current memory location within an output section specification. Unlike other assignment statements, current location assignments are evaluated during the allocation phase of the link process, so positioning of the current location assignment statement is important.

An example of a current location assignment involving the current location term is

```
SECTIONS {
    .text : { *(.text),
              end_text = . ;
    }
}
```

In this example, the symbol `end_text` is assigned the value of the memory address following the end of the .text section contents. If the starting address of the .text output section is zero and the size of all the input .text sections is 100 bytes, `end_text` will have the value 100.

4.6 SPECIAL FUNCTIONS

Five special functions provide useful information about the output sections and Linker allocation addresses. These functions and the information they return are listed in Table 4-3.

Table 4-3. Special Functions

FUNCTION	RETURNED VALUE
SIZEOF	Output section size
ADDR	Output section address
FILADDR	Output section file offset
NEXT	Next memory address multiple (which is a multiple of a specified value)
HIGHMEMADDR	Highest available memory location

The SIZEOF, ADDR, and FILEADDR functions return valid results only for output sections which have already been created in the output. For example:

```
SECTIONS {
        .text BIND( 0 ) : { *(.text) }
          .data BIND( SIZEOF(.text) ) : { *(.data) }
}
```

The SIZEOF(.text) function is used after the directive that creates the .text output section. Both output sections are bound to a specific address, but since the .text section directive precedes the .data section directive, the .text section is created in the output first; therefore, the SIZEOF request is valid. The .data output section will start at the next byte after the end of the .text section.

4.6.1 Sizeof Function

The SIZEOF function returns the number of bytes in the specified output section. The syntax for the SIZEOF function is

SIZEOF (*section_name*)

The SIZEOF function can return a valid value only for a section which has already been created in the output. If the section has not yet been created in the output, a zero is returned.

If more than one section exists with the specified section name, the information returned will be relevant only to the first section recognized.

4.6.2 Memory Address Function

The memory address function returns the starting address of the specified output section. The syntax for the memory address function is

ADDR (*section_name*)

The ADDR function can return a valid value only for a section which has already been allocated memory space, otherwise it returns zero.

If more than one section exists with the specified section name, the information returned will be relevant only to the first section recognized.

4.6.3 File Address Function

The file address function returns the file address of a section in the Linker output file. The syntax for the file address function is

FILEADDR (*section_name*)

The FILEADDR function can return a valid value only for a section which has already been allocated file space in the Linker output file, otherwise, it returns zero.

If more than one section exists with the specified section name, the information returned will be relevant only to the first section recognized.

4.6.4 Next Address Function

The next address function returns the next available memory address which is a multiple of a specified value. The syntax for the next function is

NEXT (*expr*)

Expr must evaluate to a value greater than zero.

4.6.5 Highest Memory Address Function

The highest memory address function returns the next memory address after the highest address allocated in memory. The syntax for the highest memory address is

HIGHMEMADDR

Chapter 5

Basic Linker Operations

5.1 INTRODUCTION

The Linker's primary goal is to produce executable files which run on a specified target machine. To accomplish this goal, the executable code must accommodate any special requirements imposed by the target loader or operating system. The Linker directives language allows the user to create executable files tailored to any target system.

This chapter provides a description of the basic Linker operations controlled by the directives language and a brief overview of the COFF object file format utilized by the Linker.

5.2 BASIC LINKER OPERATIONS

The basic linking process involves combining and/or manipulating the sections of input object files. Sections from input object files are known as input sections. These input sections are combined and/or manipulated to create corresponding output sections. The executable image (Linker output file) is composed of these output sections. The Linker performs four basic operations when creating an executable file:

- Memory Configuration – Establishes the actual memory range. The Linker can work with one single contiguous piece of configured memory or with memory segments. All of this information is specifiable.
- Allocation – The assignment of a starting address to an output section. The Linker directives language allows a great deal of control over this process. The ability to map input sections to an output section at a particular address allows the creation of executable-image memory mapping which meets any need.
- Resolution – The process of matching an external symbolic declaration in one source file with the symbolic definition in another file.
- Relocation – The process of calculating the actual address of symbolically referenced data or routine.

Figure 5-1 illustrates the basic linking process.

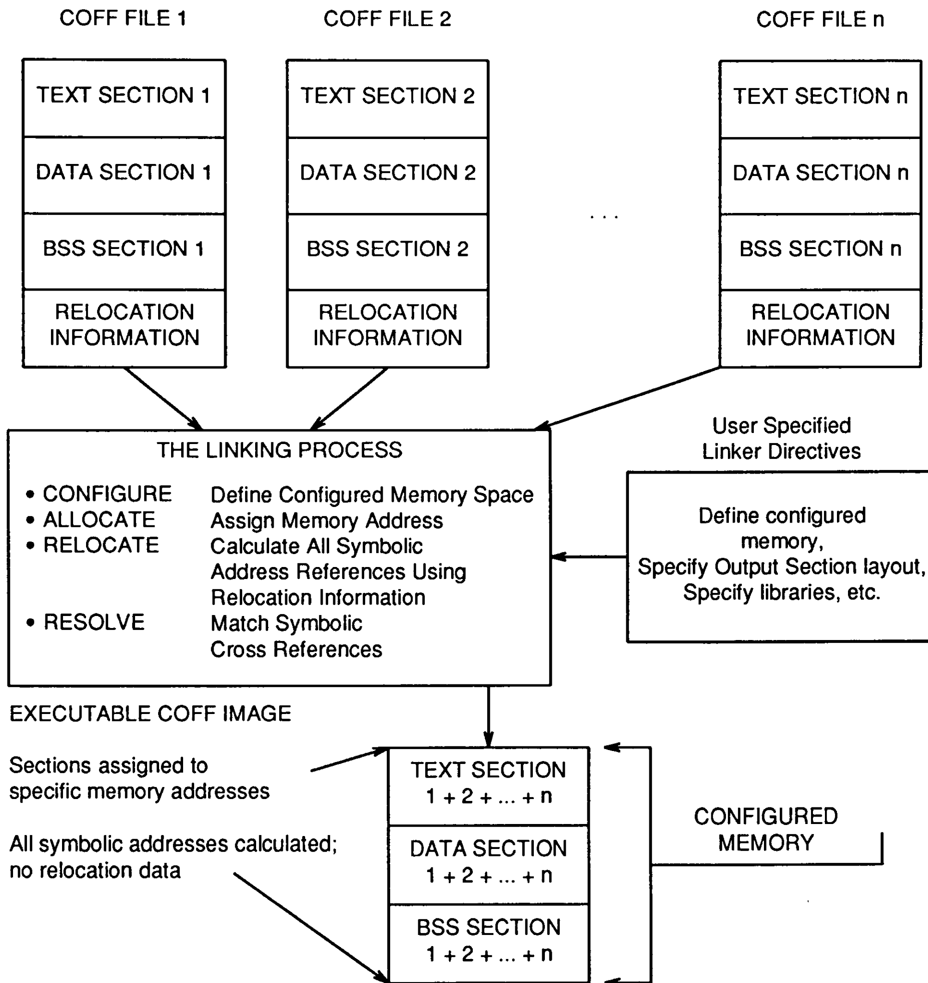


Figure 5-1. Basic Linking Process

5.2.1 Linker Allocation Algorithm

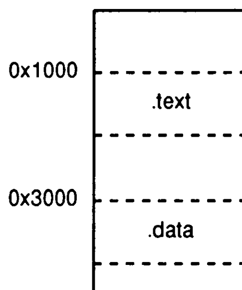
Consider two slightly different output mappings. In Mapping 1, both the text section and the data section are bound to a specific address and appear at their respective addresses in the output file. In Mapping 2, only the text section is bound and its address is roughly in the middle of the MEMORY declared address range. Therefore, the data section is placed in the first area large enough to accommodate it. If the Linker finds that the first area large enough for the sum of the input data sections is before the text section, that is where the Linker directs the input data sections even though in the `SECTIONS` directive, data input sections appear after text.

Mapping 1

The directives file for mapping 1 is

```
SECTIONS {
      .text BIND(0x1000):{*(.text)}
      .data BIND(0x3000):{*(.data)}
}
```

Which produces the following layout in memory:

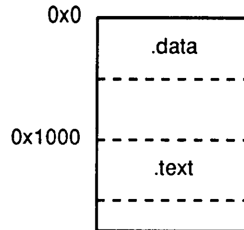


Mapping 2

The directives file for mapping 2 is

```
SECTIONS {
      .text BIND(0x1000):{*(.text)}
      .data :{*(.data)}
}
```

Which produces the following layout in memory:



Output sections are processed by the Linker according to a simple allocation algorithm:

1. Output sections, or groups of output sections, bound to a specific memory address are allocated. This guarantees the output section will begin at the bind address.
2. Output sections, or groups of output sections, directed to a particular named memory area are allocated within the named memory area. The Linker assigns addresses on a first-fit basis.
3. Output sections, or groups of output sections, directed to a memory area of particular attributes are allocated. The Linker will assign addresses on a first-fit basis within any memory area having the specified attributes.
4. All other output sections, or groups of output sections, are allocated addresses on a first-fit basis from the remaining available configured memory.

Therefore, the order of output sections within a Linker directives file does not necessarily reflect the order of the output sections in memory.

5.3 OBJECT FILE FORMAT

National Semiconductor's GNX software development environment has adopted as its standard the Common Object File Format (COFF). COFF was first developed by AT&T as the standard format for object files under UNIX System V. This standardization has many advantages, from portability of object/executable files among all *Series 32000*-based UNIX systems to the establishment of common data structures for symbolic debugging information.

5.3.1 COFF Sections

A section of an object file is the smallest unit of relocation and must occupy a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in "section headers" at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be "holes" or gaps between input sections and between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory. The "virtual address" of a section or symbol is the relative offset from address zero of the address space.

A COFF object file contains at least three special sections, .text, .data, and .bss.

- The .text section contains executable code. Executable code is segregated to its own section because it is protected against accidental modification. It is write protected by the system.
- The .data section contains initialized data. This data is available at run-time without any explicit assignment statement from the program. An example of this type of data is literal text for a print statement or C language variables initialized at the time of definition. The data section is separated from the text section because data area is readable and writable.
- The .bss section contains uninitialized data and, for efficiency reasons, is separate from the data section. Since this data is uninitialized, the only information needed for system purposes is the size of this area, and basically the bss section has just that. The values found by the program at run-time depend on the system loader (not the Linker). Most UNIX systems initialize the bss section to zeroes.

Familiarity with the structure of the input file and its basic sections is required in order to control the link process and achieve a specific format for the output.

For a comprehensive account of the COFF object file format, see the *Series 32000 GNX–Version 3 COFF Programmer's Guide* and the *Series 32000 GNX – Version 3 Assembler Reference Manual.*

Appendix A

DIRECTIVES LANGUAGE SYNTAX

A.1 SYNTAX

The following table provides a quick reference of the directives language syntax.

Table A-1. Syntax Diagram of Input Directives
Sheet 1 of 4

DIRECTIVES	→	EXPANDED DIRECTIVES
directives_file	→	{ *cmd* }
cmd	→ → → → → →	*memory* *sections* *assignment* *filename* *libname* *option*
memory	→	MEMORY { *memory_spec* { [,] *memory_spec* } }
memory_spec	→	*name* [*attributes*] : *origin_spec* [,] *length_spec*
attributes	→	({R | W | X | I})
origin_spec	→	*origin* = *long*
length_spec	→	*length* = *long*
origin	→	ORIGIN | ORG
length	→	LENGTH | LEN
sections	→	SECTIONS { *sec_or_group* { [,] *sec_or_group* } }
sec_or_group	→	*section* | *group*
group	→	GROUP {*group_options*} : { *section_list* } [*mem_spec*]
section_list	→	*section* { [,] *section* }

Table A-1. Syntax Diagram of Input Directives
Sheet 2 of 4

<table>
<tr><th>DIRECTIVES</th><th>→</th><th>EXPANDED DIRECTIVES</th></tr>
<tr><td>section</td><td>→</td><td>section_name {sec_options} : { statement_list }
[fill] [mem_spec]</td></tr>
<tr><td>group_options</td><td>→
→</td><td>bind_option | align_option
block_option</td></tr>
<tr><td>sec_options</td><td>→
→
→
→
→</td><td>bind_option | align_option
block_option
type_option
module_option
flags_option</td></tr>
<tr><td>module_option</td><td>→</td><td>module(module_name)</td></tr>
<tr><td>module</td><td>→</td><td>MODULE</td></tr>
<tr><td>bind_option</td><td>→</td><td>long | bind (special_expr)</td></tr>
<tr><td>bind</td><td>→</td><td>BIND</td></tr>
<tr><td>align_option</td><td>→</td><td>align (special_expr)</td></tr>
<tr><td>align</td><td>→</td><td>ALIGN</td></tr>
<tr><td>block_option</td><td>→</td><td>block (expr)</td></tr>
<tr><td>block</td><td>→</td><td>BLOCK</td></tr>
<tr><td>type_option</td><td>→</td><td>(DSECT) | (NOLOAD) | (COPY) |
(INFO) | (OVERLAY) | (LIB)</td></tr>
<tr><td>flags_option</td><td>→</td><td>(TYP_TEXT) | (TYP_DATA) | (TYP_BSS) |
(TYP_LINK) | (TYP_MOD)</td></tr>
<tr><td>fill</td><td>→</td><td>= long</td></tr>
<tr><td>mem_spec</td><td>→
→</td><td>>mem_name
>attributes</td></tr>
<tr><td>statement_list</td><td>→</td><td>statement { [,] statement }</td></tr>
<tr><td>statement</td><td>→
→
→
→
→
→
→</td><td>file_name
file_name ((name_list) | [COMMON] | [MOD])
* ((name_list) | [COMMON] | [MOD])
assignment
dotassign
libname
null</td></tr>
<tr><td>name_list</td><td>→</td><td>section_name { [,] section_name }</td></tr>
<tr><td>assignment</td><td>→</td><td>lside assign_op expr end</td></tr>
<tr><td>dotassign</td><td>→
→</td><td>. assign_op special_expr end
lside assign_op . end</td></tr>
</table>

Table A-1. Syntax Diagram of Input Directives
Sheet 3 of 4

<table>
<tr><th>DIRECTIVES</th><th>→</th><th>EXPANDED DIRECTIVES</th></tr>
<tr><td>lside</td><td>→</td><td>name</td></tr>
<tr><td>assign_op</td><td>→</td><td>= | += | -= | *= | /=</td></tr>
<tr><td>end</td><td>→</td><td>;</td></tr>
<tr><td>special_expr</td><td>→</td><td>expr</td></tr>
<tr><td>expr</td><td>→
→</td><td>expr binary_op expr
term</td></tr>
<tr><td>binary_op</td><td>→
→
→
→
→
→
→
→</td><td>* | / | %
+ | -
>> | <<
== | != | > | < | <= | >=
&
|
&&
||</td></tr>
<tr><td>term</td><td>→
→
→
→
→
→
→
→
→</td><td>long
name
(expr)
unary_op term
sizeof (section_name)
next (expr)
addr (section_name)
fileaddr (section_name)
highmemaddr</td></tr>
<tr><td>unary_op</td><td>→</td><td>! | ~ | -</td></tr>
<tr><td>sizeof</td><td>→</td><td>SIZEOF</td></tr>
<tr><td>next</td><td>→</td><td>NEXT</td></tr>
<tr><td>addr</td><td>→</td><td>ADDR</td></tr>
<tr><td>fileaddr</td><td>→</td><td>FILEADDR</td></tr>
<tr><td>highmemaddr</td><td>→</td><td>HIGHMEMADDR</td></tr>
</table>

Table A-1. Syntax Diagram of Input Directives
Sheet 4 of 4

DIRECTIVES	→	EXPANDED DIRECTIVES
option	→	OPTION *option_keyword* [*option_value*]
option_keyword	→ →	NATIVE OMAGIC
option_value	→	long
mem_name	→	Any valid memory specification name.
name	→	Any valid symbol name.
long	→	Any valid unsigned long integer constant.
filename	→	Any valid operating system filename. This may include a full or partial pathname. Any filename may be enclosed in double quotes in order to avoid conflicts of special characters in the filename with the directives language syntax.
section_name	→	Any valid section name, up to 8 characters.
libname	→	Any valid library name. Any library name may be enclosed in double quotes in order to avoid conflicts of special characters in the filename with the directives language syntax.
module_name	→	Any valid module name.

Appendix B

OUTPUT MAP

B.1 FORMAT DESCRIPTION

The Linker output map illustrates the layout of the output file in memory and details the composition of the output sections.

Table B-1 is an example of an output map created when linking a small file `myfile.o` and a library `mylib.a` using the default directives file for a cross-development environment.

Table B-1. Linker Output Map

OUTPUT SECTION	INPUT SECTION	MEMORY ADDRESS	SIZE	SECTION CONTENTS
.text		e000	40	
	.text	e000	34	myfile.o
	.text	e034	c	mylib.a:foo.o
	.link	e040	0	myfile.o
	.link	e040	0	mylib.a:foo.o
UNUSED		e040 through f000		
.data		f000	8	
	.data	f000	4	myfile.o
	.data	f004	4	mylib.a:foo.o
	.static	f008	0	myfile.o
	.static	f008	0	mylib.a:foo.o
.bss		f008	4	
	.bss	f008	0	myfile.o
	.bss	f008	0	mylib.a:foo.o
	.bss	f008	4	linker_defined
UNUSED		f00c through ffffffff		

The output section column lists each output section of the file in the order it appears in memory.

The input section column lists each input section name comprising the aforementioned output section.

The memory address column denotes the address in memory at which a particular input or output section begins (addresses are in hexadecimal).

The size column contains either the total size of the output section or the individual sizes of the input sections.

The section contents column details the exact source of the input section. This is either an object file name or a library name followed by a colon and the particular library member name. The term "linker_defined" in this column indicates the space the Linker has defined for certain data (*i.e.*, common symbols in the .bss section).

The term "fill space," which may appear in the section contents column, indicates the hole created in the output section through the use of the current location assignment directive.

UNUSED refers to a hole in memory at the specified address.

Appendix C

SECTION TYPE OPTIONS

All COFF output sections have a type associated with them that the Linker uses when determining how to process that section. This appendix describes the action the Linker takes for each type of output section.

A combination of the following three actions is performed on every section:

Allocation — The Linker allocates available memory for the section.

Relocation — The sections base address is updated, relocation is performed on the raw data of the section, and the section's symbol values are updated.

Loading — The Linker writes the raw data from the section to the output file.

Table C-1 lists the combination of actions taken by the Linker for each particular section type. Only one type may be associated with any particular output section.

Table C-1. Type Options

TYPE	DESCRIPTION	ALLOCATED	RELOCATED	LOADED
Default	Regular section	yes	yes	yes
DSECT	Dummy section	no	no	no
NOLOAD	Noload section	yes	no	no
COPY	Copy section (relocation and line number entries processed normally)	no	partially (no symbol relocation)	yes
INFO	Comment section	no	no	yes
OVERLAY	Overlay section	no	yes	yes
LIB	For .lib section (shared library), treated the same as INFO	no	no	yes

Appendix D

LINKER ERROR MESSAGES

D.1 INTRODUCTION

The Linker produces four categories of error messages:

- Warnings - Provide information which might be important.
- Errors - Indicate a problem which precludes successful linking. The Linker does not stop but continues and provides additional useful error messages.
- Fatal Error - Linker stops.
- Internal Error - Linker stops. Has detected extreme input file corruption.

D.2 WARNINGS

Binding an output section overrides assigning it to a particular named memory area.

Cannot open default directives file *filename*. Proceeding with basic, default linker action.

Entrypoint *symbol_name* does not exist. Using default entrypoint.

Invalid memory attribute *attribute* specified. Ignored.

Multiple definition of symbol *symbol_name*.

Object files being linked are not entirely modular or not entirely relocatable.

Symbol *symbol_name* multiply defined with differing sizes. Larger size used.

Unknown option *option ignored.*

Bad line number entry number in file *filename*.

D.3 ERRORS

Procedure descriptor to a symbol in a non-modular section reference to symbol index *number*, reference from section *section_name*, in *filename*.

Module address out of range reference to symbol index *number*, reference from section *section_name*, in *filename*.

pb relative offset out of range reference to symbol index *number*, reference from section *section_name*, in *filename*.

One byte number too big reference to symbol index *number*, reference from section *section_name*, in *filename*.

Two byte number too big reference to symbol index *number*, reference from section *section_name*, in *filename*.

One byte displacement too big reference to symbol index *number*, reference from section *section_name*, in *filename*.

Two byte displacement too big reference to symbol index *number*, reference from section *section_name*, in *filename*.

Four byte displacement too big reference to symbol index *number*, reference from section *section_name*, in *filename*.

One byte immediate too big reference to symbol index *number*, reference from section *section_name*, in *filename*.

Two byte immediate too big reference to symbol index *number*, reference from section *section_name*, in *filename*.

Undefined symbol *symbol_name*, first referenced in file *filename*.

Linkentry offset not divisible by 4 reference to symbol index *number*, reference from section *section_name*, in *filename*.

Specified undefined symbol *symbol_name* never resolved.

D.4 FATAL ERRORS

Specified bind address, *address*, not in available configured memory.

Output section *section_name* does not fit at specified bind address.

Cannot allocate named memory space for output section *section_name*.

Cannot allocate memory space of specified attributes for output section *section_name.*

Cannot allocate configured memory space for output section *section_name.*

Output section *section_name* not found when evaluating SIZE, FILEADDR or ADDR expression.

Symbol *symbol_name* not found.

Symbol *symbol_name* already defined.

Cannot open file *filename* for reading.

Bad magic number in file *filename.*

Cannot seek to symbol table in input file *filename.*

Cannot open archive file *filename.*

Cannot open directives file *filename.*

No input object files specified.

Insufficient command line arguments.

Option *option* requires an argument.

Number exceeds 32 bits.

Ending quote expected near line *number* in directives file.

Unknown linker OPTION directive near line *number* in directives file.

Cannot combine sections of type NOLOAD with other input sections.

Cannot combine sections of type LIB with sections of other types.

Input sections of type OVERLAY not allowed.

Multiply defined symbol *symbol_name.* First defined in file *filename.*

Symbol *filename* already exists in symbol table.

Cannot access GNX target setup file. Using default directives file.

Archive file *filename* is missing global symbol information. Use the GNX archive utility with the -s option to restore symbolic information.

number is not a valid integer value.

Specified version stamp number exceeds 16-bit maximum.

Specified fill value exceeds 16-bit maximum.

Input file *filename* is not in proper COFF format.

Number exceeds 32 bits near line *number* in directives file.

Error opening output file *filename.*

Unable to recover from previous errors.

Optional header magic value specified exceeds 16 bits.

Specified MEMORY region *filename* exceeds 32 bit address range.

Bad magic number in archive member *member_name* in file *filename.*

Cannot block section *section_name* to specified output file address.

Syntax error near line number "number".

D.5 INTERNAL ERRORS

Unable to access *symbol_name* during relocation phase.

Unable to open *filename* during relocation phase.

No raw data for created mod section.

Unable to allocate space for a section header.

Unable to locate defining input section for a defined external. Section number *number*, symbol index *number*, symbol name *symbol_name*, in *filename.*

Call to calloc() failed to allocate *number* bytes.

Class C_LABEL or C_FCN symbol without a section. Symbol index *number* in file *filename.*

Class C_BLOCK symbol without a section. Symbol index *number* in file *filename.*

Bad storage class *number*, symbol index *number* in file *filename.*

Unable to read string table of *filename*.

Unable to read section *number* of *filename*.

Bad relocation entry number *number* of section number *number* in input *filename*.

Unable to read raw data of section number *number* in file *filename*.

Unable to read hole in raw data for relocation entry number *number* of section number *number* in *filename*.

Unable to open input *filename* for a module symbol.

Unable to access input *filename* for a module symbol.

Unable to read section *number* of input *filename* for a module symbol.

Unable to read module (+0) symbol *symbol_name*, symbol index *number*, of *filename*.

Unable to read module (+4) symbol *symbol_name*, symbol index *number*, of *filename*.

Unable to read module (+8) symbol *symbol_name*, symbol index *number*, of *filename*.

No input section for R_STATIC_SEC relocation reference to symbol index *number*, reference from section *section_name*, in *filename*.

No input section for R_LINK_SEC relocation reference to symbol index *number*, reference from section *section_name*, in *filename*.

No input section for R_TEXT_SEC relocation reference to symbol index *number*, reference from section *section_name*, in *filename*.

Overlapping virtual memory regions.

Floating-point exception.

Bad hash table index generated for symbol *filename*.

Illegal use of '.' in directives file near line *number*.

Cannot read symbol table entries in file *filename*.

Cannot read string of symbols in archive file *filename*.

Cannot read section headers of file *filename*.

Cannot seek to start of raw data for section *filename* in file *filename*.

Cannot seek to symbol entry position in outputfile.

Bad section number for symbol index *number* in file *filename.*

Problem getting symbol name for symbol index *number* in file *filename.*

Illegal relocation entry type in relocation entry *number* of input section *number* in file *filename.*

Problems accessing archive file member containing symbol *symbol_name* in archive file *filename.*

Appendix E

SAMPLE LINKER DIRECTIVE FILES

E.1 CROSS APPLICATION

The following default directives file is provided for linking in a cross-development environment for execution on a *Series 32000* development board.

```
/*
 * db.link    1.3 (National Semiconductor) 3/31/88 18:15:25
 * copyrights and other release restrictions may apply
 */

OPTION OMAGIC 0417 /* Set GNX optional header magic number */

MEMORY {
/* Set length to  needed value
 * Origin set to start of available memory */

          db_mem : origin=0xE000, length= length
}

SECTIONS {
/* mod must be in low memory */
           .mod BIND(0xE000) : { *(.mod) }

/* Align data to page boundary. Useful if running with MMU */
        .text BIND(NEXT(4)) : { *(.text) *(.link) }
        .data BIND(NEXT(0x1000)) : { *(.data) *(.static)}
        .bss BIND(NEXT(4)) : { *(.bss) }
}

/* Special symbols used by sbrk routine in libc.a */

_etext = ADDR(.text) + SIZEOF(.text);
etext = ADDR(.text) + SIZEOF(.text);
_edata = ADDR(.data) + SIZEOF(.data);
edata = ADDR(.data) + SIZEOF(.data);
_end = HIGHMEMADDR;
end = HIGHMEMADDR;
```

E.2 NATIVE APPLICATION

The following default directives file is provided for linking a *Series 32000* development environment for execution on the same machine (*e.g.*, SYS32/20, SYS32/30).

```
/*
 * sys32_30.link    1.2 (National Semiconductor) 3/28/88 10:05:21
 * copyrights and other release restrictions may apply
 */

OPTION NATIVE           /* Mark file as executable in native mode */
OPTION OMAGIC 0413      /* Set GNX optional header magic number */

SECTIONS {
/* Mark text section as containing only text */
    .text (TYP_TEXT) BIND(NEXT(0x400000) + fileaddr(.text)) : {
        *(.init) *(.text) *(.link) *(.mod)
    }
    GROUP BIND (NEXT(0x400000) + ((ADDR(.text) + SIZEOF(.text)) % 0x1000)) : {
     .data : { *(.data) *(.static) }
     .bss : { *(.bss) }
    }
/* Bind address aligned to MMU determined boundaries */

    .comment : { *(.comment) }
}

/* Special symbols used by sbrk routine in libc.a */

_etext = ADDR(.text) + SIZEOF(.text);
etext = ADDR(.text) + SIZEOF(.text);
_edata = ADDR(.data) + SIZEOF(.data);
edata = ADDR(.data) + SIZEOF(.data);
_end = ADDR(.bss) + SIZEOF(.bss);
end = ADDR(.bss) + SIZEOF(.bss);
```

INDEX

* Wild-card character 3-6

A

Addr 4-7
Address function 4-7
Align 3-8
Aligning 3-8
Allocation 5-1
Attributes 3-2
  memory range 3-2

B

Basic Linker Operations 5-1
  allocation 5-1
  configuration 5-1
  relocation 5-1
  resolution 5-1
Basic linking process, figure of 5-2
Basic operations 5-1
Binary operators 4-3
Binary operators, table of 4-4
Bind 3-8
Binding 3-8
Block 3-9
Blocking 3-9
Bss section 5-5

C

COFF file 5-4
COFF sections 5-4
  bss 5-5
  data 5-5
  text 5-5
Command line invocation, intro to 2-1
Command line options 2-3
  debug info 2-7
  directive file 2-10
  entry point 2-9
  error message 2-8
  hole value 2-10
  keep relocation information 2-9
  local symbolic info 2-7
  memory map 2-6
  output filename 2-6
  retain relocation information 2-9
  system library 2-3
  undefined symbol 2-11
  Unix 2-3
  user library 2-3
  version 2-8
  version stamp 2-11
  VMS 2-3
  warning message 2-8
Comments 3-16
COMMON input sections 3-7
Configuration 5-1
Copy 3-11
Creating holes within output section 3-14
Creating symbols at link time 3-15
Cross application directive files E-1
Current location assignments 4-5

D

d Command line option 2-10
Data section 5-5
Defining symbols at link time 3-15
Directives Command line option 2-10
Directives file 3-1
  samples E-1
  syntax A-1
Directives language, how used 3-1
Documentation conventions 1-2
Dsect 3-11

E

e Command line option 2-9
Entry Command line option 2-9
Entry point 2-9
Environment variable for Unix 2-2
Error messages D-1
Error messages, list of D-2
Expression 4-1
Expression, simple form of 4-1
Expression
  assignment 4-5
  single term 4-2
  special function 4-6
  use 4-1

F

f Command line option 2-10
Fatal errors, list of D-2
Fileaddr 4-7
Filename 3-5
Filenames for UNIX 3-1
Filenames for VMS 3-1
Fill Command line option 2-10

Fill value of section holes 3-14
Format description B-1

G

General conventions 1-2
GNX Target Setup - gts 2-10
GNX$LIBRARY 2-3
Group 3-12
Group directive 3-12
Group link, figure of 3-13

H

Highest memory address 4-8
Highmemaddr 4-8

I

I 3-2
Info 3-11
Input directives, diagram of A-1
Input file specification 3-2
Input files 3-5
Input libraries 3-5
Input sections 3-6
Input section, source of 3-6
Input-section statement 3-5
Integer syntax 4-1
  decimal 4-1
  hexadecimal 4-1
  octal 4-1
Internal errors, list of D-4
Invocation 2-1
  Unix environment 2-1
  VMS environment 2-2

K

k Command line option 2-9
Keep Command line option 2-9
Keep relocation information 2-9

L

l Command line option 2-3
Length 3-2
Lib 3-11
LIBPATH 2-2
Library files 3-3
Library path environment variable 2-2
Library specification 3-2
Linker allocation algorithm 5-3
Linker directives, intro to 3-1
Linker error messages D-1
Linker, intro to 1-1
Linker operations, intro to 5-1
Linker output map B-1
Linker.def file 2-10
LINKERFILE 2-10
Logical name for VMS 2-3
Logical negation 4-2

M

m Command line option 2-6
M Command line option 2-7
Manual overview 1-1
Map Command line option 2-6
Memory 3-2
Memory address function 4-7
Memory directive 3-1, 3-2
Memory range attributes 3-2
Memory range 3-10
  sections 3-9, 3-10
MOD input sections 3-7
Module 3-11
Module name 3-11
Muldefs Command line option 2-7
Multiply defined symbols 2-7

N

Native 3-16
Native application directive files E-2
Next 4-8
Next function 4-8
Noload 3-11
Nolocal Command line option 2-7
Nowarning Command line option 2-8

O

o Command line option 2-6
Object file 5-4
  format 5-4
Object files 3-3
Offset function 4-6
Omagic 3-16
one´s complement 4-2
Options 2-3
  see command line options 2-3
Origin 3-2
Output Command line option 2-6
Output file options, specifying 3-16
Output file 3-16
  characteristics 3-16
  magic number 3-16
  native environment 3-16
Output filename 2-6
  default 2-6
  Unix 2-6

VMS 2-6
Output map B-1
Output section option
  align 3-8
  bind 3-8
  block 3-9
  module 3-11
  type 3-11
Overlay 3-11

P

Partial link 3-7
  retain 3-7
Program entry point 2-9

R

R 3-2
r Command line option 2-9
Relational operations 4-3
Relocation 5-1
Resolution 5-1
Retain Command line option 2-9
Retained link 2-9

S

s Command line option 2-7
S Command line option 2-8
Sample directives files E-1
Section definition directives 3-4
Section holes 3-14
Section names 3-6
  user defined 3-6
Section type options C-1
Sections 3-6
Sections start value 3-8
  binding 3-8
Shift operations 4-3
Silent Command line option 2-8
Size function 4-7
SIZEOF 4-6, 4-7
Special functions 4-6
Special functions, table of 4-6
Stamp Command line option 2-11
Strip command line option 2-7
Syntax conventions 1-2

T

t Command line option 2-8
Text section 5-5
Trivial link 2-10
Two´s complement 4-3
Typ_bss 3-11
Typ_data 3-11
Type options C-1
Type options, table of C-1
Typing 3-11
Typ_link 3-11
Typ_mod 3-11
Typ_text 3-11

U

u Command line option 2-11
Unary operator 4-2
Unary operators, table of 4-2
Unix options, table of 2-4
Usym Command line option 2-11

V

V Command line option 2-8
Version Command line option 2-8
VMS environment invocation 2-2
VMS options, table of 2-5
VS Command line option 2-11

W

W 3-2
Warning for defined common data 2-7
Warning messages, list of D-1
Wild card 3-6

X

X 3-2
x Command line option 2-7

National Semiconductor | **MICROCOMPUTER SYSTEMS DIVISION**

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) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada 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:

____

____

____

____

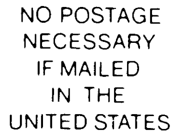
____

GNX — Version 3 Linker User's Guide

FOLD, STAPLE, AND MAIL **424010506-003A**

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA



POSTAGE WILL BE PAID BY ADDRESSEE

National Semiconductor Corporation
Microcomputer Systems Division
Technical Publications Dept., M/S 7C261
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-9968

Software
Problem Report

Name: ______________________________
Street: ______________________________
City: ____________________ State: ______ Zip: __________
Phone: ____________________ Date: __________

Instructions

Use this form to report bugs, or suggested enhancements. Mail the form to National Semiconductor. Technical Support may be contacted at:

(800) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada only
((0)8141) 103-330 - West Germany

Category

☐ Software Problem ☐ Request For Software Enhancement
☐ Other ☐ Documentation Problem, Publication # __________

Software Description

National Semiconductor Product ____________________
Version __________ Registration # __________
Host Computer Information ____________________
Operating System ____________________
Rev. __________ Supplier __________

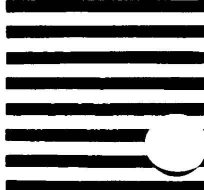
Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

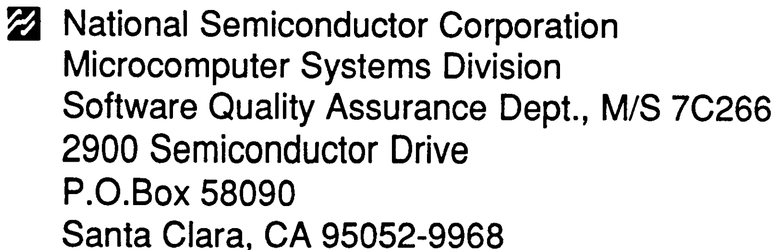
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA



POSTAGE WILL BE PAID BY ADDRESSEE



National Semiconductor Corporation
Microcomputer Systems Division
Software Quality Assurance Dept., M/S 7C266
2900 Semiconductor Drive
P.O.Box 58090
Santa Clara, CA 95052-9968

National Semiconductor Use Only

Tech Support ____________________ Date Received ______________
Software Q.A. ____________________ Date Received ______________
Report Number ____________________
Action Taken :

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 Piedias
(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

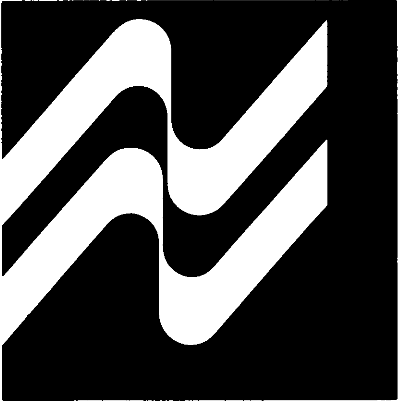
National Semiconductor Hong Kong Ltd.
Southeast Asia Marketing
Austin Tower, 4th Floor
22-26A Austin Avenue
Tsimshatsui, Kowloon, H.K.
Tel: 852 3-7243645
Cable: NSSEAMKTG
Telex: 52996 NSSEA HX

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



Series 32000®

GNX — Version 3
COFF Programmer's Guide

Customer Order Number 424010507-003
NSC Publication Number 424010507-003A
August 1988

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
A	08/88	First Release. *Series 32000®* GNX — Version 3 COFF Programmer's Guide NSC Publication Number 424010507-003A.

PREFACE

This manual describes the GNX (GENIX™ Native and Cross-Support) implementation of the Common Object File Format (COFF). The intended audience of this manual is the implementor of language tools or operating systems for the *Series 32000®* microprocessor family. This audience includes writers of compilers, assemblers, linkers, debuggers, kernels, or other tools which must create or access object code information. This manual aids in understanding the object file format, which lies at the heart of the implementation of the GNX Language Tools. This manual is also useful for creating new tools and modifying existing GNX tools.

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

No part of this document may be reproduced in any form or by any means without the prior written consent of National Semiconductor Corporation.

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

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

Series 32000 is a registered trademark of National Semiconductor Corporation.

Portions of this document are derived from AT&T copyrighted material and reproduced under license from AT&T; portions are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California.

UNIX is a registered trademark of AT&T.

VAX, VMS, and DEC are trademarks of Digital Equipment Corporation.

IBM is a registered trademark of International Business Machines Corporation.

CONTENTS

Chapter 1 OVERVIEW

1.1 INTRODUCTION 1-1
1.2 GENERAL 1-2
1.3 DEFINITIONS AND CONVENTIONS 1-3
1.3.1 Sections 1-3
1.3.2 Headers 1-3
1.3.3 Physical and Virtual Addresses 1-3
1.3.4 Target Machine 1-4

Chapter 2 HEADERS

2.1 INTRODUCTION 2-1
2.2 FILE HEADER 2-1
2.2.1 Description of the Fields of the File Header 2-2
2.2.2 Contents of the File Header Flags 2-2
2.2.3 Description of the File Header Flags 2-2
2.2.4 Guidelines for Using the File Header Flags 2-2
2.3 OPTIONAL HEADER INFORMATION 2-4
2.3.1 Guidelines for Using the Optional Header 2-5
2.3.2 The Optional Header Magic Numbers 2-5
2.3.3 The Optional Header Flags 2-6
2.4 SECTION HEADERS 2-8
2.4.1 Use of the Section Header 2-9
2.4.2 Section Header Flags 2-9
2.4.3 .bss Section Header 2-10

Chapter 3 SECTIONS

3.1 INTRODUCTION 3-1
3.2 LOADING A FILE WITH MODULAR FEATURES 3-1

Chapter 4 RELOCATION INFORMATION

4.1 INTRODUCTION 4-1
4.2 RELOCATION ENTRY 4-1
4.3 COFF RELOCATION ENTRY STRUCTURE 4-2
4.4 SEMANTICS 4-4

Chapter 5 LINE NUMBERS

5.1 INTRODUCTION 5-1

5.2 USING LINE NUMBERS 5-2

Chapter 6 SYMBOL TABLE

6.1 INTRODUCTION 6-1
6.2 SPECIAL SYMBOLS 6-1
6.2.1 Inner Blocks 6-4
6.3 SYMBOLS AND FUNCTIONS 6-6
6.4 SYMBOL TABLE ENTRIES 6-7
6.4.1 Symbol Names 6-8
6.4.2 Storage Classes 6-9
6.4.3 Storage Classes for Special Symbols 6-11
6.4.4 Symbol Value Field 6-11
6.4.5 Section Number Field 6-13
6.4.6 Section Numbers and Storage Classes 6-13
6.4.7 Type Entry 6-15
6.4.8 Symbol Interpretation Environment 6-18
6.4.9 Type Entries and Storage Classes 6-18
6.4.10 Structure for Symbol Table Entries 6-20
6.5 AUXILIARY TABLE ENTRIES 6-20
6.5.1 Filenames 6-20
6.5.2 Sections 6-22
6.5.3 Tagnames 6-22
6.5.4 Structures, Unions, and Enumerations 6-23
6.5.5 Functions 6-24
6.5.6 Arrays 6-24
6.5.7 End of Blocks and Beginning and End of Functions . . . 6-24
6.5.8 Beginning of Blocks 6-25
6.5.9 Auxiliary Entry Declaration 6-26
6.6 LINKED LISTS IN THE SYMBOL TABLE 6-26
6.7 STRING TABLE 6-28

FIGURES

Figure 1-1. GNX Common Object File Format 1-2
Figure 2-1. File Header Contents 2-1
Figure 2-2. Optional Header Contents 2-4
Figure 2-3. Section Header Contents 2-8
Figure 4-1. Relocation Section Contents 4-2
Figure 5-1. Line Number Grouping 5-1
Figure 5-2. Line Number Structure Lineno 5-2

Figure 6-1.	GNX COFF Symbol Table	6-2
Figure 6-2.	Special Symbols (.bb and .eb)	6-4
Figure 6-3.	Nested Blocks	6-5
Figure 6-4.	Symbol Table	6-6
Figure 6-5.	Symbols for Functions	6-6
Figure 6-6.	The Special Symbol .target	6-7
Figure 6-7.	Symbol Table Entry Format	6-8
Figure 6-8.	Name Field	6-9
Figure 6-9.	Auxiliary Entry for Filenames	6-22
Figure 6-10.	Auxiliary Entry for Sections	6-22
Figure 6-11.	Auxiliary Entry for Tagnames	6-23
Figure 6-12.	Auxiliary Entry for Structures, Unions and Enumerations	6-23
Figure 6-13.	Auxiliary Entry for Functions	6-24
Figure 6-14.	Auxiliary Entry for Arrays	6-25
Figure 6-15.	Auxiliary Entry for Beginning of Function and End of Block/Function	6-25
Figure 6-16.	Auxiliary Entry for Beginning of Block	6-26
Figure 6-17.	Linked List Structures in the Symbol Table	6-27
Figure 6-18.	String Table	6-28

TABLES

Table 2-1.	File Header Flags	2-3
Table 2-2.	Optional Header Magic Numbers	2-6
Table 2-3.	Optional Header Flags	2-7
Table 2-4.	Flags for Section Handling	2-10
Table 2-5.	Flags for Type of Data Contained in Section	2-10
Table 4-1.	Relocation Type Flag Definitions	4-3
Table 6-1.	Special Symbols in the Symbol Table	6-3
Table 6-2.	Storage Classes	6-10
Table 6-3.	Storage Class by Special Symbols	6-11
Table 6-4.	Storage Class and Value	6-12

Table 6-5. Section Number 6-13
Table 6-6. Section Number and Storage Class 6-14
Table 6-7. Fundamental Types 6-16
Table 6-8. Derived Types 6-17
Table 6-9. Type Entries by Storage Class 6-19
Table 6-10. Auxiliary Symbol Table Entries 6-21

INDEX

Chapter 1

OVERVIEW

1.1 INTRODUCTION

This manual describes the GNX Language Tools' implementation of the Common Object File Format (COFF) for *Series 32000* microprocessor-based systems, and it serves as a "how to" guide for implementors of language tools. Because National's GNX COFF is derived from AT&T's UNIX® COFF, the word "common" is descriptive and widely accepted.

There are two kinds of GNX COFF files: relocatable and executable. Relocatable files, or object files as they are normally called, are produced by the assembler and may contain unresolved external references. One or more object files are combined by the linker to produce an executable file which has no unresolved references and may contain additional symbolic information for the debugger.

The assembler creates the object file, and assembler directives control the creation of specific sections in the object files. For example, .text denotes the start of a text section. Generally, a High-Level Language (HLL) compiler generates the assembly source code. Thus, there is an extremely strong interaction between the compilers and the assembler and linker which must support the compilers.

Because of compatibility with other operating systems using COFF, some symbols and fields are included in the format to maintain commonality. GNX COFF allows the use of all the hardware features of this microprocessor (notably, modular relocation capabilities.)

The content of the object file is determined at compile time with command line options to the assembler, linker and compiler. These options vary depending on the host operating system. The examples of options given in this manual are for a UNIX or UNIX-derived host. For the options to the assembler, linker, and compiler for your specific host, see the *Series 32000 GNX — Version 3 Commands and Operations Manual.*

GNX COFF is structurally general and extensible. This manual describes how to:

- Add system-dependent information to the object file without obsoleting access utilities.
- Access symbolic information used for debuggers and other applications.

1.2 GENERAL

The overall structure of a GNX COFF file is shown in Figure 1-1. Sections 1 through *n* may be user-defined. Extensive information is included for symbolic software debugging.

FILE HEADER
OPTIONAL HEADER
Section 1 Header
•••
Section *n* Header
Raw Data for Section 1
•••
Raw Data for Section *n*
Relocation Info for Section 1
•••
Relocation Info for Section *n*
Line Numbers for Section 1
•••
Line Numbers for Section *n*
SYMBOL TABLE
STRING TABLE

Figure 1-1. GNX Common Object File Format

The last three types of information (line numbers, symbol table, and string table) may be empty if the program is linked with the "strip" option of the linker or if the symbol table is removed by the "strip" command. The line number information does not appear unless the program compiles with the compiler option to produce additional symbol table information (*e.g.*, –g). In addition, if there are no unresolved external references after linking, then the relocation information is no longer needed and is absent. The string table may also be absent if the source file does not contain any symbols with names longer than eight characters.

The term "executable" refers to an object file that contains no errors or unresolved references. Specific target operating systems may place additional constraints on an executable file such as requiring the presence of an optional header.

1.3 DEFINITIONS AND CONVENTIONS

Before proceeding, the user should become familiar with the terms and conventions of sections, physical addresses, virtual addresses and target machines.

1.3.1 Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. Each section defines raw data, relocation information for the raw data, and line numbers for the section. Raw data, relocation information, or line numbers may be absent when not needed. Symbolic information in the symbol table contains references to sections. In addition to the sections .text, .data, .bss, .mod, .link, .static, .lib and .comment, user-defined sections are also supported by the assembler and the linker.

NOTE: It is a mistake to assume that every GNX COFF file has a certain number of sections, or to assume characteristics of sections such as their order, location in the file, or load address in memory. This information must be obtained through access of the appropriate data fields after the GNX COFF file has been created. This information is contained in the file and section headers.

1.3.2 Headers

Headers contain file pointers that are used to locate the various components of the COFF file. File pointers are byte offsets from the beginning of the file that can be used to directly locate the symbol table, raw data, relocation, or line number information. File pointers can be used readily with the standard C library function fseek.

1.3.3 Physical and Virtual Addresses

The terms "physical address" and "virtual address" are considered the same in this document. They both refer to an object's location in the program's memory space. For targets with a Memory Management Unit (MMU), this address is not necessarily the same as the address of that object in physical memory. The latter is usually known as the physical address and generates from the virtual address by the MMU. This address is unknown to the program and is irrelevant to the object file format.

For historical reasons, some of the data structures in the object file contain fields for both virtual and physical addresses. Usually, they have the same values, but sometimes GNX COFF programs use only one of these fields and the other is invalid.

1.3.4 Target Machine

The term "target machine" refers to the machine on which the object file is destined to run. For a native set of tools this is the same machine as the one on which the code was developed. Generally, the GNX cross tools cross-compile when the target and development machine differ. This document describes the use of GNX COFF in both cases.

Chapter 2

HEADERS

2.1 INTRODUCTION

Three types of headers describe the overall content of the object file: the file header, the optional header, and the section headers. The file header describes the style of code and the number of sections. The optional header describes the attributes, size, and location of the .text, .data, and .bss sections in memory.* The section headers describe each section and the data location for the section in the file.

2.2 FILE HEADER

The file header describes the style of code and the number of sections. Figure 2-1 shows the contents of the file header.

Bytes	Declaration	Name	Description
0-1	unsigned short	f_magic	magic number
2-3	unsigned short	f_nscns	number of section headers
4-7	long int	f_timdat	time and date stamp
8-11	long int	f_symptr	file pointer to the start of the symbol table
12-15	long int	f_nsyms	number of entries in the symbol table
16-17	unsigned short	f_opthdr	number of bytes in the optional header
18-19	unsigned short	f_flags	flags (see Table 2-1)

Figure 2-1. File Header Contents

NOTE: The corresponding C structure definition for this file may be found in the header file `filehdr.h`. This header file maps correctly to the structure in Figure 2-1 when it compiles with the GNX C compiler.

* The current optional header is specifically for a UNIX or UNIX-derived operating system and may vary for different targets in the future.

2.2.1 Description of the Fields of the File Header

- f_magic — The magic number specifies the style of code for a particular operating system or down-load program. The mnemonic NS32GMAGIC = 0524 octal is used for all fully relocatable GNX COFF files; NS32SMAGIC = 0525 octal is used for GNX COFF files that contain modular code.
- f_nscns — Indicates the number of section headers which equals the number of sections.
- f_timdat — The time and date stamp indicates when the file was created expressed in terms of the number of elapsed seconds since 00:00:00 GMT, January 1, 1970. (This value is host operating system dependent.)
- f_symptr — The file pointer contains the starting address of the symbol table.
- f_nsyms — Number of entries in the symbol table (includes symbols and their auxilliaries).
- f_opthdr — Number of bytes in the optional header. This is used by all referencing programs that seek to the beginning of the section header table. This ensures compatibility of a utility across differing target operating systems and future versions of COFF.
- f_flags — Flags (see Table 2-1). These last 2 bytes (f_flags field) are used by the linker and the object file utilities.

2.2.2 Contents of the File Header Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Some of these flags are no longer used but are kept to maintain commonality. The currently defined flags are given in Table 2-1.

2.2.3 Description of the File Header Flags

In general, COFF is designed to work with either left-to-right or right-to-left byte ordering. However, *Series 32000* COFF files always use the F_AR32WR flag. The flags F_MINMAL, F_UPDATE, F_SWABD, F_AR16WR, F_AR32W, and F_PATCH specify other architectures and are never used by the GNX tools.

2.2.4 Guidelines for Using the File Header Flags

F_RELFLG — The linker normally strips relocation information from the file after all references resolve in the linking process. The –r option retains this information for further linking.

F_EXEC — The linker turns this on when it finds no unresolved external references.

F_LNNO and F_LSYMS — The strip utility or the –s linker option strip line numbers and local symbols from the file.

Table 2-1. File Header Flags

MNEMONIC	FLAG	DESCRIPTION
F_RELFLG	00001	Relocation information stripped from the file.
F_EXEC	00002	File is executable (*i.e.*, no unresolved external references).
F_LNNO	00004	Line numbers stripped from the file.
F_LSYMS	00010	Local symbols stripped from the file.
F_MINMAL	00020	Not used by the GNX Language tools.
F_UPDATE	00040	Not used by the GNX Language tools.
F_SWABD	00100	Not used by the GNX Language tools.
F_AR16WR	00200	File has the byte ordering used by the PDP™-11/70 processor. Not used on *Series 32000* COFF files.
F_AR32WR	00400	File has the byte ordering used by the VAX™-11/780 and the *Series 32000* (*i.e.*, 32 bits per word, least significant byte first).
F_AR32W	01000	File has the byte ordering used by the 3B 20S computers (*i.e.*, 32 bits per word, most significant byte first). Not used on *Series 32000* COFF files.
F_PATCH	02000	Not used by the GNX Language tools.
NOTE:	Flags F_MINMAL, F_UPDATE, F_SWABD, F_AR16WR, F_AR32W, and F_PATCH are reserved for use by other implementations. Effects are undefined if set.	

F_AR32WR — File has the byte ordering used by the VAX-11/780 and the *Series 32000* (*i.e.*, 32 bits per word, least significant byte first). Currently, this flag is always used. If the GNX tools port to other host architectures in the future, other values such as AR32W may be used.

2.3 OPTIONAL HEADER INFORMATION

The optional header contains system-dependent information about the object file. (Currently all executable object files produced by the linker contain the optional header.) The fields of the *Series 32000* version of the optional header are described in Figure 2-2.

NOTE: The corresponding C structure definition for this header may be found in the `aouthdr.h` header file. This header file maps correctly to the structure (as shown in Figure 2-2) when it compiles with the GNX C compiler.

Bytes	Declaration	Name	Description
0-1	short	magic	magic number (see Section 2.3.2)
2-3	short	vstamp	version stamp
4-7	long int	tsize	size of text in bytes
8 -11	long int	dsize	size of initialized data in bytes
12-15	long int	bsize	size of uninitialized data in bytes
16-19	long int	msize	size of module table in bytes
20-23	long int	mod_start	start address of module table
24-27	long int	entry	entry point memory address
28-31	long int	text_start	base address of first text section
32-35	long int	data_start	base address of first data section
36-37	unsigned short	entry_mod	memory address of the module table entry of the module containing the entry point
38-39	unsigned short	flags	see Section 2.3.3

Figure 2-2. Optional Header Contents

The size entries in the optional header of a section are calculated as the difference between the starting address of the first section of that name and the ending address of the last section of that name. If a section of a different type intervenes the sections

whose addresses are being calculated, the size does include the intervening section. Therefore, size is most meaningful when sections are grouped (*i.e.*, no intervening sections).

The field tsize is computed as the difference between the next address following the last non-empty .text or .link section and the base address of the first such section. Field dsize is computed as the difference between the next address following the last non-empty .data or .static section and the base address of the first such section. Fields bsize and msize are computed similarly based on sections .bss and .mod.

2.3.1 Guidelines for Using the Optional Header

General utility programs such as the symbol table access library functions are not concerned with the contents of the optional header. Such utilities seek past this record by using the size of optional header information in the file header (the f_opthdr field) or, preferably, by using the standard access routines to seek to the desired location.

By default, the linker sets Vstamp to zero. A user can set the version number at linker invocation with –VS *version_number*, where *version_number* is a C short (16-bit) value. See the *Series 32000 GNX— Version 3 Linker User's Guide* for details.

2.3.2 The Optional Header Magic Numbers

In general, magic numbers provide a quick way for utilities to check how a file has been processed. The magic number in the optional header supplies operating system dependent information about the object file. See the `aouthdr.h` header file for this set of machine-dependent values. Whereas the magic number in the file header specifies the type of machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should load. (Specifically, it indicates how the *Series 32000* kernel processes a COFF file when loading it to produce a process image. See Section 3.2 for further details.)

The magic numbers recognized by the operating system are given in Table 2-2.

Table 2-2. Optional Header Magic Numbers

VALUE	DESCRIPTION
0407	The text section is not write-protected or sharable; the data section is contiguous with the text section.
0410	The data section starts at the next segment following the text section; the text section is write protected.
0413	The data section starts at the next segment following the text section; the text section is write protected. Relocation and alignment within the file are appropriate for paging.
0417	Do not use the optional header for loading; Use section headers instead.
0443	The object file is configured for shared libraries. (GENIX V.3 only.)

Typical segment sizes are 64-Kbyte or 1-Mbyte. These are controlled by the linker directives language.

2.3.3 The Optional Header Flags

The flags field of the COFF GNX version records the alignment granularity and the protections to be assigned sections when loaded. Flags are also reserved for distinguishing between system types. Alignment granularity positions the raw data for sections with respect to the beginning of the containing COFF file. The meaning of the flags for both alignment granularity and the protections to be assigned sections when loaded, according to the definitions in Table 2-3.

Table 2-3. Optional Header Flags

FIELD NAME	MNEMONIC	FLAG	MEANING
U_AL (mask 0x07)			section alignment
	U_AL_NONE	0x00	full-word alignment
	U_AL_512	0x01	512-byte alignment
	U_AL_1024	0x02	1-Kbyte alignment
	U_AL_2048	0x03	2-Kbyte alignment
	U_AL_4096	0x04	4-Kbyte alignment
	U_AL_8192	0x05	8-Kbyte alignment
		0x06	reserved
		0x07	reserved
U_PR (mask 0x38)			section protections ("1" if writable and "0" if it is read only.)
	U_PR_DATA	0x08	data section
	U_PR_TEXT	0x10	text section
	U_PR_MOD	0x20	module section
		0x40	reserved
		0x80	reserved
U_SYS			system type
	U_SYS_5	0x100	(reserved for future expansion; do not use)
	U_SYS_42	0x200	(reserved for future expansion; do not use)

2.4 SECTION HEADERS

Every object file has section headers that specify the data layout within the file. There is one section header for every section in the file. The section header is described in Figure 2-3.

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null-padded section name
8-11	long int	s_paddr	physical address of section
12-15	long int	s_vaddr	virtual address of section
16-19	long int	s_size	section size in bytes (due to padding, this value is always a multiple of 4 bytes)
20-23	long int	s_scnptr	file pointer to raw data
24-27	long int	s_relptr	file pointer to relocation entries
28-31	long int	s_lnnoptr	file pointer to line number entries
32-33	unsigned short	s_nreloc	number of relocation entries
34-35	unsigned short	s_nlnno	number of line number entries
36-39	long int	s_flags	s_flags (see Section 2.4.2)
40-43	long int	s_modsym	symbol table index (if s_modsym is greater than 0, then this field indicates the symbol index which contains the section; if there is no mod symbol, then s_modsym = -1)
44-45	unsigned short	s_modno	memory address of the module table entry associated with this section
46-47	short	s_pad	padding to 4-byte multiple

Figure 2-3. Section Header Contents

NOTE: The corresponding C structure definition for this header may be found in the `scnhdr.h` include file. This header file maps correctly to the structure shown in Figure 2-3 when compiled with the GNX C compiler.

2.4.1 Use of the Section Header

The file pointers in the Section Header are byte offsets from the beginning of the file that directly locate the start of data, relocation, line number, or symbol table entries for the section. Because of this definition, they can be readily used with the standard C library function fseek. For example, fseek may be called with s_scnptr to prepare a program to read the raw data section of the file.

2.4.2 Section Header Flags

The lower 12 bits of the flags field indicate a section type. The bit definitions are shown in Tables 2-4 and 2-5. Table 2-4 shows the flags which define the handling of the section by the linker; these flags are mutually exclusive. Table 2-5 shows the flags which specify the data type in a section.

The following three paragraphs define the terms used in Table 2-4.

The term "allocated" indicates that the section does use space in configured memory, is a unique memory area, and shows up in the linker's output map.

The term "relocated" indicates that relocation information applies to the section so that the section symbols appear appropriately updated in the symbol table.

The term "loaded" indicates that the section is included in the linker's output file and should load into memory by the operating system or down-load program. The raw data of nonloaded sections are not included in the linker's output.

The STYP flags are interpreted by the GNX linker in the following manner:

- GROUP, RELOC, COLLAPSE, PROT, and PAD are not used in GNX.
- REG means that the section is not one of the following: DSECT, NOLOAD, COPY, INFO, OVER, or LIB. A REG section may be TEXT, DATA, BSS, MOD, or LINK.
- BSS is regular except that its does not have raw data. The pointer to raw data (s_scnptr) is 0. The BSS flag is mutually exclusive with TEXT, DATA, MOD, or LINK.
- TEXT means that the section contains code. DATA means that the section contains initialized data. MOD means that the section contains module tables. LINK means that the section contains link table entries. These flags are not mutually exclusive.
- A LIB section cannot combine with anything other than a LIB section. NOLOAD sections cannot combine at all. OVERLAY sections are not allowed as input to the linker.
- If a BSS section combines with any other section, its contents become all zeroes and it changes to a DATA section.

2.4.3 .bss Section Header

The entry for uninitialized data in a .bss section deviates from the normal rule in the section header table. A .bss section has a size, symbols that refer to it, and symbols that are defined in it. At the same time, a .bss section has no relocation entries, no line number entries, and no data. Therefore, a .bss section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a .bss section header, is zero.

The section header flag indicating .bss data is mutually exclusive with the other flags in Table 2-5. If a .bss section combines with any other section, its type becomes STYP_DATA, and its data is set to all zeroes.

Table 2-4. Flags for Section Handling

MNEMONIC	FLAG	DESCRIPTION	ALLOCATED	RELOCATED	LOADED
STYP_REG	0x00	regular section	yes	yes	yes
STYP_DSECT	0x01	dummy section	no	yes	no
STYP_NOLOAD	0x02	noload section	yes	yes	no
STYP_COPY	0x10	copy section (relocation and line number entries pro-cessed normally)	no	yes	no
STYP_INFO	0x4000	comment section	no	no	no
STYP_OVER	0x8000	overlay section	no	yes	no
STYP_LIB	0x10000	for .lib section (shared library), treated the same as INFO	no	no	no

Table 2-5. Flags for Type of Data Contained in Section

MNEMONIC	FLAG	DESCRIPTION
STYP_TEXT	0x20	section contains executable text
STYP_DATA	0x40	section contains initialized data
STYP_BSS	0x80	section contains only uninitialized data
STYP_MOD	0x100	section contains module table
STYP_LINK	0x200	section contains link table

Chapter 3

SECTIONS

3.1 INTRODUCTION

The section is the basic unit for defining the contents of an area of memory. Each section is described by its section header. Raw data, relocation information, and line numbers for each section occur after the section headers. Figure 1-1 shows that section headers are followed by the appropriate number of bytes of text or data. If the optional header is present, the beginning of the section aligns in the file at the alignment boundary given by the U_AL part of the optional header flags field.

Files produced by the GNX compilers, the assembler, and the linker may contain sections for code, data, and uninitialized data plus additional sections for *Series 32000* modularity. The .text section contains the instruction text (*i.e.*, executable code), the .data section contains initialized data variables, and the .bss section contains uninitialized data variables. In support of the *Series 32000* modularity features, a module table is contained in a .mod section, link tables are contained in .link sections, and static-base-relative data are in .static sections.

The linker's "SECTIONS" directive described in the *Series 32000 GNX — Version 3 Linker User's Guide* allows users to:

- Describe how input sections combine.
- Direct the placement of output sections.

If no SECTIONS directives are given, each input section appears in an output section of the same name. For example, if a number of object files from the compiler link together, each containing the three sections .text, .data, and .bss, then the output object file also contains the same three sections.

3.2 LOADING A FILE WITH MODULAR FEATURES

A GNX COFF file normally loads with the information in the section headers. The loading process may be hastened by the use of the information in the optional header. However, use of the linker command language or modular features of the *Series 32000* architecture may result in section configurations which invalidate the contents of the optional header. In these cases, the optional header information cannot load the object file.

In addition, various specializations of this general structure are possible.

- If modular software features are not used, the linker can combine files by using traditional relocation, resulting in only one module and a nearly "standard" file organization.
- The linker can link certain sections to appear consecutively in the resulting memory image. The operating system can load each such aggregate section as an undivided whole, obtaining starting addresses and lengths from the optional header. Many variations of this scheme are possible.
- When the optional header obtains loading information, its magic and flag fields discriminate among the different possibilities.

Chapter 4

RELOCATION INFORMATION

4.1 INTRODUCTION

Since a COFF section may be relocated by a linker, references to symbols of that section must also be relocatable. The relocation entries contain sufficient information to properly update each reference when the referenced section relocates.

4.2 RELOCATION ENTRY

The relocation entries describe a reference and a referenced memory location. The reference is the area in a section which contains code bytes for accessing the referenced memory location. The referenced memory location is the (relocatable) memory being accessed.

The relocation entry describes the reference and its relationship to the referenced memory location. During the link process, the reference may move, the referenced memory location may move, or (typically) both may move.

In order to implement this, each section with relocatable references contains a list of relocation entries. Each relocation entry is composed of:

- the address of the reference in memory. These addresses always fall within the boundaries of the section.
- a symbol table index. The value of this symbol defines the address of the referenced memory location.
- the addressing type of this reference.
- the relative addressing mode of the reference to the referenced memory location.
- the data format of the reference.
- the size of the reference.

4.3 COFF RELOCATION ENTRY STRUCTURE

Figure 4-1 shows the structure of the 10-byte COFF record representing the relocation entry. Item 1 is represented by the r_vaddr field. Item 2 is a r_symndx field. Items 3 through 6 are represented in the r_type field.

Bytes	Declaration	Name	Description
0-3	long int	r_vaddr	(virtual) address of reference
4-7	long int	r_symndx	symbol table index
8-9	unsigned short	r_type	relocation type (see below)
10-11	short	dummy	dummy padding bytes

Figure 4-1. Relocation Section Contents

NOTE: The C structure declaration for this file may be found in the `reloc.h` header file.

The relocation entries are actually packed one per 10-byte field in the object. Therefore, use macro definition RELSZ (which is currently 10) to determine the size of each relocation entry. Do not use sizeof (RELOC) since this returns 12 due to padding field "dummy."

In GNX COFF, r_type field is partitioned into four subfields given by the bit-mask definitions in Table 4-1.

Table 4-1. Relocation Type Flag Definitions

FIELD	MNEMONIC	MASK/VALUE	FIELD DESCRIPTION/MEANING
R_ADDRTYPE		0x000f	address type of reference
	R_NOTHING	0x0000	no relocation to be performed
	R_ADDRESS	0x0001	normal memory addressing
	R_LINKENTRY	0x0002	link table index (prescaled by 4)
	R_STATIC_SEC	0x0003	default static section base address
	R_LINK_SEC	0x0004	default link section base address
	R_TEXT_SEC	0x0005	default text section base address
R_RELTO		0x00f0	the addressing mode
	R_ABS	0x0000	absolute addressing
	R_PCREL	0x0010	pc relative addressing
	R_SBREL	0x0020	static base relative
R_FORMAT		0x0f00	the format of the address
	R_NUMBER	0x0000	a two's complement number (low order to high order)
	R_DISPL	0x0100	*Series 32000* displacement (high order to low order with Huffman encoding bits)
	R_PROCDES	0x0200	*Series 32000* procedure descriptor (16-bit module followed by 16-bit offset)
	R_IMMED	0x0300	a two's complement number (high order to low order)
R_SIZESP		0xf000	the size of the reference
	R_S_08	0x0000	1 byte long
	R_S_16	0x1000	2 bytes long
	R_S_32	0x2000	4 bytes long

4.4 SEMANTICS

The R_ADDRTYPE (0x000f) subfield specifies the type of addressing for the reference.

R_NOTHING (0x0000) flag indicates that no action is required by the linker.

R_ADDRESS (0x0001) flag is the normal value for any memory reference.

R_LINKENTRY (0x0002) flag is used when the reference is an index off of the link base. (See the modsym field of the section header.)

This instructs the linker that the reference is scaled by 4 (as is appropriate for External-addressing mode and the index provided on the CXP instruction).

If the link base of the referenced memory location changes, the linker adjusts the reference appropriately. (For R_LINKENTRY, R_RELTO is always R_ABS).

The R_RELTO (0x00f0) subfield indicates how the linker must relocate the reference when one or both of the sections involved are moved.

R_STATIC_SEC (0x0003) flag is used for the default static base of a module. The symbol is the name of the module. The reference is relocated relative to the movement of the base of the .static section.

R_LINK_SEC (0x0004) flag is used for the default link base of a module. The symbol is the name of the module. The reference is relocated relative to the movement of the base of the .link section.

R_TEXT_SEC (0x0005) flag is used for the default program base of a module. The symbol is the name of the module. The reference is relocated relative to the movement of the base of the .text section.

R_ABS (0x0000) indicates that the reference is relative to the beginning of memory. Therefore, the linker will adjust the reference (up/down) when the referenced memory location is moved (up/down).

R_PCREL (0x0010) indicates that the reference is the offset from the PC of the current instruction to the referenced memory location. In this case the linker adjusts the reference (down) as the PC of the reference moves (up). The linker also adjusts the reference (up/down) when the referenced memory location moves (up/down).

R_SBREL (0x0020) indicates that the reference is relative to the static base of the referenced memory location. The linker updates the reference when the static base of the referenced memory location changes during linking. (This occurs when two or more .static sections combine.) The static base of the reference is known from the current module associated with the referencing section. (See the modsym field of the section header.)

The R_FORMAT (0x0f00) subfield indicates the data format for this reference.

R_NUMBER (0x0000) indicates the reference is represented as a two's complement number with the low-order byte first.

R_DISPL (0x0100) indicates the reference is represented as a *Series 32000* displacement with Huffman encoding bits and a signed displacement in high to low order.

R_PROCEDES (0x0200) indicates the reference is a *Series 32000* procedure descriptor consisting of a 16-bit module number (low byte, high byte) followed by a 16-bit procedure offset (low byte, high byte). Both values are unsigned.

R_IMMEDIATE (0x0300) indicates the address is kept as a *Series 32000* immediate value with the most significant byte first.

The R_SIZE (0xf000) subfield indicates the size of reference.

R_S_08 (0x0000) flag indicates a 1-byte reference.

R_S_16 (0x1000) flag indicates a 2-byte reference.

R_S_32 (0x2000) flag indicates a 4-byte reference.

Chapter 5

LINE NUMBERS

5.1 INTRODUCTION

When invoked with the proper option, the compilers generate an entry in the object file for every source line where a breakpoint can be inserted. Users can then reference line numbers when using a software debugger. All line numbers in a section are grouped by function, as shown in Figure 5-1.

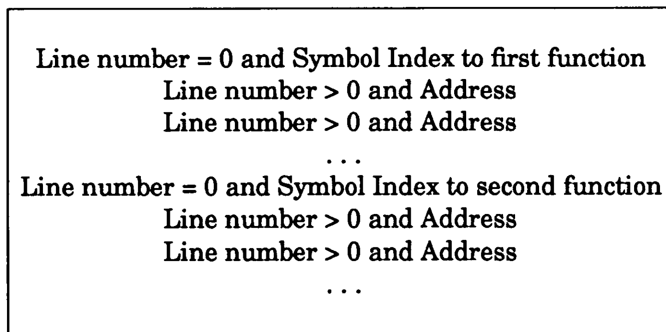


Figure 5-1. Line Number Grouping

As shown in Figure 5-2, the symbol index entry (l_symndx) occupies the same field as the address entry (l_paddr). What the field actually represents is determined by the value of the line number field (l_lnno). A line number of zero indicates that the entry is a Symbol Index. A nonzero line number indicates that the entry is the address of the beginning of that line in memory. The line numbers are relative to the beginning of the function.

NOTE: The C declaration for this structure may be found in the `linenum.h` header file. The declaration correctly maps to the structure in Figure 5-2 when it is compiled with the GNX language tools.

Bytes	Declaration	Name	Description
0-3	long	l_symndx	symbol table index of the function name (for l_lnno = 0)
0-3	long	l_paddr	address of the line number in memory (for l_lnno > 0)
4-5	unsigned short	l_lnno	line number (or 0)
6-7	short	dummy	dummy padding bytes

Figure 5-2. Line Number Structure Lineno

5.2 USING LINE NUMBERS

The line number entries appear in increasing order of address.

The size of these entries is indicated by the macro definition LINESZ (which is currently 6). Using sizeof (LINENO) returns an inappropriate value (currently 8) due to the padding of field "dummy."

The auxiliary entry for the function's .bf special symbol contains a C-source absolute line number which may be used with relative line numbers to get absolute line numbers within the function.

Chapter 6

SYMBOL TABLE

6.1 INTRODUCTION

The purpose of the symbol table is two-fold. First, the symbol table contains essential information about the object file such as names of files, names of sections, and defined and undefined global symbols. The second optional purpose is to produce the complete description of the program symbols for symbolic debugging purposes.

This chapter describes the case when the complete symbol information is generated by the compiler, when invoked by the C compiler's –g option. The compiler generates assembly code which directs the creation of the symbol table. Sections 6.1, 6.2, and 6.3 describe the overall structure of the symbol table. Sections 6.4, 6.5, and 6.6 describe the details of the entry for each symbol.

The symbol table is a sequence of symbols. Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear sequentially as shown in Figure 6-1. Note that some older tools may not adhere strictly to the standard given; the kernel, for instance is very forgiving.

The word "statics" in Figure 6-1 means static symbols defined "static" outside any function. Static symbols may be local or external. Local static symbols provide permanent storage local to that functon, whereas external static symbols allow functions from separate object files to share information without passing it explicitly. The symbol table consists of at least one fixed-length entry per symbol, with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds either the name itself (if the name is short enough) or an offset in the string table for the name, the value, and other information.

6.2 SPECIAL SYMBOLS

The symbol table contains some special symbols that are generated by the compiler, assembler and linker. These symbols are given in Table 6-1.

Six of these special symbols occur in pairs. The .bb and .eb symbols indicate the boundaries of inner blocks; a .bf and .ef pair brackets each function; and a *.x*fake and .eos pair names and defines the limit of structures, unions, and enumerations that were not named. The .eos symbol also terminates the declaration of named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler must invent a name to use in the symbol table. The name chosen for the symbol table is *.x*fake, where

x is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are ".0fake," ".1fake," and ".2fake."

Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entry.

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
...
statics for file 1
...
filename *n*
function *m*
local symbols for function *m*
...
statics for file *n*
defined global symbols
undefined global symbols

Figure 6-1. GNX COFF Symbol Table

Table 6-1. Special Symbols in the Symbol Table

SYMBOL	MEANING
.file	filename
.text	text section address
.data	data section address
.mod	module table address
.static	static section address
.link	link section address
.bss	bss section address
.bb	start of inner block address
.eb	end of inner block address
.bf	start of function address
.ef	end of function address
.target	pointer to the structure or union returned by a function
.xfake	dummy tag name for structure, union, or enumeration
.eos	end of members of structure, union, or enumeration
.sb	sb register initialization value
_etext, etext	next available address after the end of the last text output section
_edata, edata	next available address after the end of the last data output section
_end, end	next available address after the end of the last output section

6.2.1 Inner Blocks

The special symbols .bb and .eb respectively begin and end "blocks" which delineate the scope of subsequent symbol definitions. All symbol definitions following the .bb special symbol and before the matching .eb symbol are considered local to that block. For example, the C language defines a block as a compound statement that begins with a left brace ({) and ends with a right brace (}). An "inner block" is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol .bb is put in the symbol table immediately before the first local symbol of that block. In addition, a special symbol .eb is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 6-2.

.bb
local symbols for that block
.eb

Figure 6-2. Special Symbols (.bb and .eb)

Note that external functions are stored with the local symbols in order to retain local context. Because inner blocks can be nested by several levels, the .bb-.eb pairs and associated symbols may also be nested. For a relevant example in C, see Figure 6-3.

```
{                               /* block 1 */
    int i;
    char c;
    ...
    {                           /* block 2 */
            long a;
            ...
            {                   /* block 3 */
                int x;
                ...
            }                   /* block 3 */
    }                           /* block 2 */

    {                           /* block 4 */
            long i;
            ...
    }                           /* block 4 */
}                               /* block 1 */
```

Figure 6-3. Nested Blocks

An example of a symbol table is shown in Figure 6-4.

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Figure 6-4. Symbol Table

6.3 SYMBOLS AND FUNCTIONS

For each function, a special symbol .bf is put between the function name and the first local symbol of the function in the symbol table. In addition, a special symbol .ef is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 6-5.

function name
.bf
local symbols
.ef

Figure 6-5. Symbols for Functions

If the return value of the function is a structure or union, a special symbol .target is put between the function name and the .bf. The sequence is shown in Figure 6-6.

function name
.target
.bf
local symbols
.ef

Figure 6-6. The Special Symbol .target

The GNX system compilers invent .target to store the function-returned structure or union. The symbol .target is an automatic variable with "pointer" type. Its value field in the symbol table entry is always zero.

6.4 SYMBOL TABLE ENTRIES

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain the following 20 bytes of information. The meaning of each of the fields in the symbol table entry is described in Figure 6-7.

Bytes	Declaration	Name	Description
0-7	(see Section 6.4.1)	_n	these eight bytes contain either the name of a symbol or the offset of the symbol name in the string table
8-11	long int	n_value	symbol value; storage class dependent
12-13	short	n_scnum	section number of symbol
14-15	unsigned short	n_type	basic and derived type specification
16	char	n_sclass	storage class of symbol
17	char	n_numaux	number of auxiliary entries
18	char	n_env	symbol interpretation environment
19	char	n_dummy	currently unused

Figure 6-7. Symbol Table Entry Format

It should be noted that indices for symbol table entries begin at zero and count upward. Each auxiliary entry also counts as one symbol.

6.4.1 Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table.

In this case, the 8 bytes contain two long integers; the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Figure 6-8.

Bytes	Declaration	Name	Description
0-7	char	n_name	8-character null-padded symbol name
0-3	long	n_zeroes	zero in this field indicates the name is in the string table
4-7	long	n_offset	offset of the name in the string table

Figure 6-8. Name Field

Some special symbols are generated by the compiler and linker as discussed in Section 6.2. The compiler attaches an underscore (_) to all the user-defined symbols it generates.

6.4.2 Storage Classes

The following discussion of the storage class field assumes that the standard symbol interpretation environment is in effect (`n_env` == 0). In other environments the type field may be interpreted differently.

The storage class field has one of the values described in Table 6-2. These "defines" may be found in the `storclass.h` header file.

All of these storage classes except for C_ALIAS and C_HIDDEN are generated by the compiler or assembler. The storage classes C_ALIAS and C_HIDDEN are not used by the GNX language tools.

Some of these storage classes are "dummies," used only internally by the compiler and the assembler. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Table 6-2. Storage Classes

MNEMONIC	VALUE	STORAGE CLASS
C_EFCN	–1	physical end of function
C_NULL	0	
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	undefined static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARM	17	register parameter
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	filename
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

6.4.3 Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes as given in Table 6-3.

Table 6-3. Storage Class by Special Symbols

SPECIAL SYMBOL	STORAGE CLASS
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

6.4.4 Symbol Value Field

The meaning of the "value" of a symbol depends on its storage class. This relationship is summarized in Table 6-4 (note that null has a value of zero).

If a symbol has storage class C_FILE, the value of that symbol equals the symbol table entry index of the next .file symbol. That is, the .file entries form a one-way linked list in the symbol table. If there are no more .file entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the linker relocates the section, the value of these symbols changes.

Table 6-4. Storage Class and Value

STORAGE CLASS	MEANING OF VALUE
C_AUTO	stack offset in bytes
C_EXT	relocatable address
C_STAT	relocatable address
C_REG	register number
C_LABEL	relocatable address
C_MOS	offset in bytes from base of structure
C_ARG	stack offset in bytes from frame pointer
C_STRTAG	null
C_MOU	offset in bytes from base of union
C_UNTAG	null
C_TPDEF	null
C_ENTAG	null
C_MOE	enumeration value
C_REGPARM	register number
C_FIELD	bit displacement
C_BLOCK	relocatable address of executable image
C_FCN	relocatable address of executable image
C_EOS	size of structure or union which this symbol terminates
C_FILE	see Section 6.4.4
C_ALIAS	tag index
C_HIDDEN	relocatable address

6.4.5 Section Number Field

Section numbers are listed in Table 6-5. A special section number (–2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the filename. A section number of –1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and .eos symbols. The .text, .data, and .bss symbols default to section numbers are positive integers starting at 1.

Table 6-5. Section Number

MNEMONIC	SECTION NUMBER	MEANING
N_DEBUG	–2	special symbolic debugging symbol
N_ABS	–1	absolute symbol
N_UNDEF	0	undefined external symbol
N_SCNUM	1-077767	section number where symbol has been defined

With one exception, a section number of zero indicates a relocatable external symbol that is undefined in the current file. The one exception is a multiply-defined external symbol (*i.e.*, FORTRAN common or an uninitialized variable defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is zero and the value of the symbol is a positive number giving the size of the symbol. When the files are combined, the linker combines all input symbols into one symbol with the .bss section number. The maximum size of all input symbols with the same name allocates space for the symbol, and the value becomes the symbol's address. This is the only case in which a symbol has a section number of zero and a nonzero value.

6.4.6 Section Numbers and Storage Classes

Symbols with certain storage classes are also restricted to certain section numbers. They are summarized in Table 6-6.

Table 6-6. Section Number and Storage Class

STORAGE CLASS	---------SECTION NUMBER---------			
	N_ABS	N_UNDEF	N_SCNUM	N_DEBUG
C_AUTO	yes	no	no	no
C_EXT	yes	yes	yes	no
C_STAT	no	no	yes	no
C_REG	yes	no	no	no
C_LABEL	no	yes	yes	no
C_MOS	yes	no	no	no
C_ARG	yes	no	no	no
C_STRTAG	no	no	no	yes
C_MOU	yes	no	no	no
C_UNTAG	no	no	no	yes
C_TPDEF	no	no	no	yes
C_ENTAG	no	no	no	yes
C_MOE	yes	no	no	no
C_REGPARM	yes	no	no	no
C_FIELD	yes	no	no	no
C_BLOCK	no	no	yes	no
C_FCN	no	no	yes	no
C_EOS	yes	no	no	no
C_FILE	no	no	no	yes
C_ALIAS	no	no	no	yes

6.4.7 Type Entry

The type of a symbol determines the meaning of the value found in the value field for that symbol. The following discussion of the type field assumes that the standard symbol interpretation environment is in effect (n_env == 0). In other environments, the type field may be interpreted differently.

The type field in the symbol table entry contains information about the basic and derived type for the symbol. The compiler generates this information only if the option to produce additional symbol table information is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is as follows:

d6	d5	d4	d3	d2	d1	typ

Bits 0-3, called "typ," indicate one of the following fundamental types given in Table 6-7.

Table 6-7. Fundamental Types

MNEMONIC	VALUE	TYPE
T_NULL	0	type not assigned
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating-point
T_DOUBLE	7	double-word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Note that T_MOE is redundant, as C_MOE (refer to Table 6-2) will always suffice.

Bits 4-15 are arranged as six 2-bit fields marked "d1" through "d6." These d fields represent levels of the derived types given in Table 6-8.

Table 6-8. Derived Types

MNEMONIC	VALUE	TYPE
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func( );
```

Here func is the name of a function that returns a pointer to a character. The fundamental type of func is 2 (character), the d1 field is 2 (function), and the d2 field is 1 (pointer). Therefore, the type word in the symbol table for func contains the hexadecimal number 0x62, which is interpreted as "function that returns a pointer to a character."

```
short *tabptr[10][25][3];
```

Here tabptr is a three dimensional array of pointers to short integers. The fundamental type of tabptr is 3 (short integer); the d1, d2, and d3 fields each contain a 3 (array), and the d4 field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3, indicating a "three dimensional array of pointers to short integers."

6.4.8 Symbol Interpretation Environment

The meaning of symbol table entries and their auxiliaries is affected by the value of the symbol interpretation environment field. The environment designated by a zero value in this field is distinguished as the "standard" environment; the descriptions given elsewhere in this document pertain only to this environment. The standard environment is well-suited for recording symbol information from C programs. Other environments and corresponding environment-specific symbol table entry formats may be used for recording symbol information arising from other languages.

The length of symbol table entries and auxiliary entries is independent of the symbol interpretation environment. Moreover, the partitioning of "main" symbol table entries into fields is independent of the environment, although the specific meaning assigned to the value, type, and storage class fields may depend on the environment field.

6.4.9 Type Entries and Storage Classes

Table 6-9 shows the type entries that are legal for each storage class.

Table 6-9. Type Entries by Storage Class

STORAGE CLASS	"D" ENTRY FUNCTION?	ARRAY?	POINTER?	"TYP" ENTRY BASIC TYPE
C_AUTO	no	yes	yes	Any
C_EXT	yes	yes	yes	Any
C_STAT	yes	yes	yes	Any
C_REG	no	no	yes	Any
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any
C_ARG	yes	no	yes	Any
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARM	no	no	yes	Any
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Conditions for the `d` entries apply to d1 through d6, except that it is impossible to have two consecutive derived "function" types.

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have "array" as its derived type.

6.4.10 Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry may be found in the `syms.h` header file.

6.5 AUXILIARY TABLE ENTRIES

Zero or more auxiliary entries are possible as indicated by the n_numaux field of the symbol entry.* Auxiliary entries immediately follow the associated symbol table entry. Each auxiliary table entry contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type, storage class, and the symbol interpretation environment designated in the main symbol table entry. Auxiliaries for the standard environment are summarized in Table 6-10.

In Table 6-10, tagname means any symbol name including the special symbol *.xfake*, and fcname and arrname represent any symbol name.

Any symbol that satisfies more than one condition in Table 6-10 should have a union format in its auxiliary entry. Symbols that do not satisfy any of the following conditions should not have any auxiliary entry.

6.5.1 Filenames

The format for filenames is shown in Figure 6-9.

If a filename is more than 14 characters long, it has a nonzero x_foff value and is stored in the string table at the indicated offset. Otherwise, x_foff is zero and the filename resides in the x_fname field.

---

* Currently no more than one auxiliary entry is used by any tool. AT&T's COFF also includes the possibility of more than one auxiliary entry. Earlier tool sets which did not allow this possibility are considered to be in error.

Table 6-10. Auxiliary Symbol Table Entries

NAME	STORAGE CLASS	TYPE ENTRY D1	TYPE ENTRY TYP	AUXILIARY ENTRY FORMAT
.file	C_FILE	DT_NON	T_NULL	filename
.text, .data	C_STAT	DT_NON	T_NULL	section
tagname	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tagname
.eos	C_EOS	DT_NON	T_NULL	end-of-structure
fcname	C_EXT C_STAT	DT_FCN	(See Note.) any	function
arrname	(See Note.)	DT_ARY	(See Note.) any	array
.bb	C_BLOCK	DT_NON	T_NULL	beginning-of-block
.eb	C_BLOCK	DT_NON	T_NULL	end-of-block
.bf, .ef	C_FCN	DT_NON	T_NULL	beginning- and end- of- function
name related to structure, union, enumeration	(See Note.)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration
NOTE: C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF				

Bytes	Declaration	Name	Description
0-13	char[]	x_fname	filename
14-15	-	-	unused (filled with zeroes)
16-19	long	x_foff	string table offset of filename (when > 14 long)

Figure 6-9. Auxiliary Entry for Filenames

6.5.2 Sections

The auxiliary table entries for sections have the format shown in Figure 6-10.

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-5	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-11	long	x_linoptr	pointer to line number entries for this section
12-19	-	-	unused (filled with zeroes)

Figure 6-10. Auxiliary Entry for Sections

6.5.3 Tagnames

The auxiliary table entries for tagnames have the format shown in Figure 6-11.

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeros)
6-7	unsigned short	x_size	size of struct, union, and enumeration in bytes
8-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-19	-	-	unused (filled with zeroes)

Figure 6-11. Auxiliary Entry for Tagnames

6.5.4 Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, enumeration, and end-of-structure symbols have the format shown in Figure 6-12.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index (points to the symbol which names the structure)
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of struct, union, or enumeration
8-19	-	-	unused (filled with zeroes)

Figure 6-12. Auxiliary Entry for Structures, Unions and Enumerations

6.5.5 Functions

The auxiliary table entries for functions have the format shown in Figure 6-13.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index (points to a tag for the return value of the function, *e.g.*, a structure)
4-7	long int	x_fsize	size of function in bytes
8-11	long int	x_lnnoptr	file pointer to line number entries
12-15	long int	x_endndx	index of next entry beyond this function
16-17	unsigned short	x_callseq	calling sequence information
18-19	unsigned short	x_level	function nesting level

Figure 6-13. Auxiliary Entry for Functions

6.5.6 Arrays

The value of an array is a memory pointer to the 0th entry (*i.e.*, [0, 0, ..., 0]) of the array, even if the array has negative indices.

The auxiliary table entries for arrays have the format shown in Figure 6-14.

6.5.7 End of Blocks and Beginning and End of Functions

The auxiliary table entries for the end of blocks and the beginning and end of functions have the format shown in Figure 6-15.

The field x_plude is a prelude for the .bf and a postlude for the .ef special symbol. Some programming languages require code at the beginning or end of a function to manipulate the stack. During these manipulations, the contents of the stack are unintelligible to the debugger. The x_plude field allows the compiler to tell the debugger not to access the stack during this prelude or postlude.

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index (points to the tag symbol for the array, if any)
4-5	unsigned short	x_lnno	line number of declaration
6-7	unsigned short	x_size	size of the array in bytes
8-9	unsigned short	x_dimen[0]	first dimension (number of elements)
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-19	-	-	unused (filled with zeroes)

Figure 6-14. Auxiliary Entry for Arrays

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number .bf - line number within the file .ef or .eb - line number relative to the corresponding .bf or .bb
6-17	-	-	unused (filled with zeroes)
18-19	unsigned short	x_plude	prelude or postlude size (length of code for which stack is invalid)

Figure 6-15. Auxiliary Entry for Beginning of Function and End of Block/Function

6.5.8 Beginning of Blocks

The auxiliary table entries for the beginning of blocks have the format shown in Figure 6-16.

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_lnno	C-source line number
6-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-19	-	-	unused (filled with zeroes)

Figure 6-16. Auxiliary Entry for Beginning of Block

6.5.9 Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry may be found in the `syms.h` header file.

6.6 LINKED LISTS IN THE SYMBOL TABLE

The following example serves to illustrate the use of the n_value, n_endndx, and n_tagndx fields in building linked list structures in the symbol table.

The following C fragment has been compiled using the –g and –c flags. Figure 6-17 shows the resulting link list structures in the symbol table of file1.o. The C program shows examples of tagged and untagged structure declarations and a function returning a structure.

```
int global1;
struct foo_tag
    {
        char a;
        int b;
    } foo;
struct
    {
        int i;
        char ch;
    } flop;
struct foo_tag fun ()
    {
    }
```

Figure 6-17. Linked List Structures in the Symbol Table

In general, a tag index points back to a referenced structure or enumeration and an end index points around a structure or function to the next symbol. Occasionally the `n_value` field is used as a tag index or an end index (as shown with the `.file` symbol in Figure 6-17).

6.7 STRING TABLE

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; therefore, offsets into the string table are greater than or equal to four. This size value includes the 4 bytes of the size itself so that the minimum value for an empty string table is size 4.

For example, given a file containing two symbols (with names longer than eight characters, `long_name_1` and `another_one`) the string table has the format as shown in Figure 6-18.

The index of `long_name_1` in the string table is 4 and the index of `another_one` is 16.

0	1	2	3	4	5	6	7	8	9	10	11	12	13
28				l	o	n	g	—	n	a	m	e	—

14	15	16	17	18	19	20	21	22	23	24	25	26	27
1	\0	a	n	o	t	h	e	r	—	o	n	e	\0

Figure 6-18. String Table

INDEX

A

allocated 2-9
aouthdr.h 2-4, 2-5
Arrays 6-24
Auxiliary entry
  and enumerations 6-23
  end of block/function 6-25
  for arrays 6-25
  for beginning of block 6-26
  for beginning of functions 6-25
  for filenames 6-22
  for functions 6-24
  for sections 6-22
  for structure, unions 6-23
  for tagnames 6-23
Auxiliary symbol table entries, list of 6-21
Auxiliary table entries 6-20
  arrays 6-24
  blocks, beginning of 6-25
  declaration 6-26
  end of blocks 6-24
  end of structures 6-23
  filenames 6-20
  functions 6-24
  functions, beginning of 6-24
  functions, end of 6-24
  sections 6-22
  tagnames 6-22

B

Basic type entries 6-15
.bb 6-1, 6-4
.bf 5-2, 6-1, 6-6
Block 6-4
Blocks, beginning of 6-25
.bss section 3-1
.bss section header 2-10

C

.comment 1-3

D

.data section 3-1
declaration 6-26
Definitions and conventions 1-3
  physical address 1-3
  sections 1-3
  target machine 1-4
  virtual address 1-3
Derived type entries 6-15
Derived types, list of 6-17

E

.eb 6-1, 6-4
.ef 6-1, 6-6
End index 6-28
End of blocks 6-24
End of structures 6-23
Enumerations 6-23
.eos 6-1
Executable 1-3

F

File header flags
  contents of 2-2
  description of 2-2
  guidelines for use 2-2
  list of 2-3
File headers 2-1
  contents of 2-1
  fields of 2-2
Filenames 6-20
Flags 2-9
Functions 6-24
  beginning of 6-24
  end of 6-24
Fundamental type entries 6-15
Fundamental types, list of 6-16

G

GNX common object file format 1-2

H

Headers 2-1
  use of 1-3

I

Inner blocks 6-4

L

.lib 1-3
Line numbers 1-2, 5-1
  grouping of 5-1
  structure of 5-2
  using 5-2
lineno, structure of 5-2
linenum.h 5-1
LINESZ 5-2
Linker 6-13
.link section 3-1

Linked list structures in the symbol table 6-27
  figure of 6-27
Linked lists 6-26
loaded 2-9

M

Magic number 2-2, 2-5
MMU 1-3
.mod section 3-1
Modular features 3-1
  loading a file with 3-1

N

Name field, figure of 6-9
Nested blocks, figure of 6-5
n_value 6-26

O

Optional header 2-1, 2-4, 3-1
Optional header flags 2-6
  list of 2-7
Optional header magic numbers 2-5
  list of 2-6
Optional header
  contents of 2-4
  guidelines for use 2-5
Overview 1-1

P

Physical address 1-3

R

Relocatable symbols 6-11
relocated 2-9
Relocation Information 4-1
Relocation section, contents of 4-2
Relocation type flag definitions 4-3
reloc.h 4-2

S

scnhdr.h 2-8
Section header flags 2-9
  list of 2-10
Section headers 2-1, 2-8, 2-10, 3-1
  contents of 2-8
  use of 2-9
Section number and storage classes
  list of 6-14
Section number field 6-13
Section numbers
  list of 6-13
  storage classes 6-13
Sections 1-3, 3-1, 6-22
SECTIONS directive 3-1
Special symbols 6-1, 6-9, 6-11
  .bb 6-1, 6-4
  .bf 6-1, 6-6
  .eb 6-1, 6-4
  .ef 6-1, 6-6
  .eos 6-1
  in the symbol table 6-3
  sequence of 6-4, 6-7
  .target 6-7
  .xfake 6-1
.static section 3-1
statics 6-1
Storage class and value 6-12
Storage classes 6-9, 6-11
  section numbers 6-13
  special symbols of 6-11
  type entries 6-18
  value of 6-10
storclass.h 6-9
String table 1-2
  figure of 6-28
Structure for symbol table entries 6-20
Structures 6-23
Symbol interpretation environment 6-18
Symbol name 6-8
Symbol table 1-2, 6-2, 6-6
Symbol table entries 6-7, 6-20
Symbol table
  list of entry fields 6-8
  order of symbols 6-2
Symbol value field 6-11
Symbols and functions 6-6
Symbols for functions, sequence of 6-6
syms.h 6-20

T

Tag index 6-26
Tagnames 6-22
.target 6-7
Target machine 1-4
.text section 3-1
Type entries 6-15
  basic 6-15
  derived 6-17
  fundamental 6-16
  list by storage class 6-19
  storage classes 6-18
Type field 6-15

U

Unions 6-23

V
Virtual address 1-3
Vstamp 2-4

X
.xfake 6-1

National
Semiconductor

MICROCOMPUTER
SYSTEMS DIVISION

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) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada 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 3 COFF Programmer's Guide

FOLD, STAPLE, AND MAIL

424010507-003A

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

National Semiconductor Corporation
Microcomputer Systems Division
Technical Publications Dept., M/S 7C261
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-9968

Software
Problem Report

Name: ____________________
Street: ____________________
City: ____________________ State: ______ Zip: ____________
Phone: ____________________ Date: ____________

Instructions

Use this form to report bugs, or suggested enhancements. Mail the form to National Semiconductor. Technical Support may be contacted at:

(800) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada only
((0)8141) 103-330 - West Germany

Category

☐ Software Problem ☐ Request For Software Enhancement
☐ Other ☐ Documentation Problem, Publication # ____________

Software Description

National Semiconductor Product ____________
Version ____________ Registration # ____________
Host Computer Information ____________
Operating System ____________
Rev. ____________ Supplier ____________

Problem Description

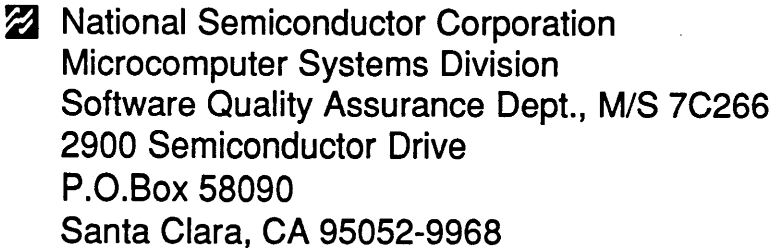
Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

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
Microcomputer Systems Division
Software Quality Assurance Dept., M/S 7C266
2900 Semiconductor Drive
P.O.Box 58090
Santa Clara, CA 95052-9968

National Semiconductor Use Only

Tech Support ____________________ Date Received ______________
Software Q.A. ____________________ Date Received ______________
Report Number ____________________
Action Taken :

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 Piedias
(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

Series 32000®

GNX — Version 3
Support Libraries
Reference Manual

Customer Order Number 424010508-003
NSC Publication Number 424010508-003B
September 1988

REVISION RECORD

REVISION	RELEASE DATE	SUMMARY OF CHANGES
A	08/88	First Release. *Series 32000®* GNX — Version 3 Support Libraries Reference Manual NSC Publication Number 424010508-003A.
B	09/88	The math library in Chapter 4 has been reorganized for readability. The calling sequences for the FORTRAN, Modula-2, and Pascal compilers have been corrected.

PREFACE

This manual describes the GNX (GENIX™ Native and Cross-Support) Libraries and library routines, which provide run-time support for the development of software for National Semiconductor's *Series 32000®* microprocessor family.

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

No part of this document may be reproduced in any form or by any means without the prior written consent of National Semiconductor Corporation.

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

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

Series 32000 is a registered trademark of National Semiconductor Corporation.

Portions of this document are derived from AT&T copyrighted material and reproduced under license from AT&T; portions are based in part on the Fourth Berkeley Software Distribution under license from The Regents of the University of California.

UNIX is a registered trademark of AT&T.

VAX, VMS, and DEC are trademarks of Digital Equipment Corporation.

CONTENTS

Chapter 1 OVERVIEW

1.1 INTRODUCTION 1-1
1.2 OPERATING SYSTEM CALL SIMULATION 1-1
1.3 MANUAL ORGANIZATION 1-2
1.4 DOCUMENTATION CONVENTIONS 1-3
1.4.1 General Conventions 1-3
1.4.2 Conventions in Syntax Descriptions 1-3
1.4.3 Example Conventions 1-4

Chapter 2 SYSTEM CALLS

2.1 INTRODUCTION 2-1
2.2 DESCRIPTION OF SYSTEM CALLS 2-3
2.2.1 Routines that Do Not Require System Calls 2-3
2.2.2 Routines that Use Simulated System Calls 2-3
2.3 SYSTEM CALL SUMMARIES 2-3
2.3.1 Close 2-7
2.3.2 Creat 2-8
2.3.3 _Exit 2-10
2.3.4 Getdtablesize 2-11
2.3.5 Lseek 2-12
2.3.6 Open 2-14
2.3.7 Read 2-16
2.3.8 Sbrk 2-17
2.3.9 Unlink 2-18
2.3.10 Write 2-20

Chapter 3 GNX DB SUPPORT LIBRARY ROUTINES

3.1 INTRODUCTION 3-1
3.2 ABORT 3-2
3.3 ABS 3-3
3.4 ATOF 3-4
3.5 BSTRING 3-5
3.6 CTIME 3-7
3.7 ECVT 3-9
3.8 EXIT 3-10

3.9 FCLOSE 3-11
3.10 FERROR 3-12
3.11 FLOOR 3-13
3.12 FOPEN 3-14
3.13 FREAD 3-16
3.14 FREXP 3-17
3.15 FSEEK 3-18
3.16 GETC 3-19
3.17 GETS 3-21
3.18 INSQUE 3-22
3.19 ISATTY 3-23
3.20 MALLOC 3-24
3.21 MEMORY 3-26
3.22 PERROR 3-28
3.23 PRINTF 3-29
3.24 PUTC 3-32
3.25 PUTS 3-34
3.26 QSORT 3-35
3.27 RANDOM 3-36
3.28 REGEX 3-38
3.29 SCANF 3-40
3.30 SETBUF 3-44
3.31 SETJMP 3-46
3.32 STRING 3-47
3.33 SWAB 3-49
3.34 UNGETC 3-50

Chapter 4 FLOATING-POINT LIBRARY

4.1 INTRODUCTION 4-1
4.2 DETAILS AND USE OF THE MATH LIBRARY 4-2
4.2.1 Number Formats 4-2
4.2.2 Integer Formats 4-2
4.2.3 Floating-point Formats 4-3
4.2.4 Reserved Operand Values and Operations 4-6
4.2.5 Not a Number (NAN) 4-7
4.2.6 Infinity 4-8
4.2.7 Denormalized Numbers 4-8

4.2.8 Math Environment Control Function 4-9
4.2.9 Using the Math Environment Functions 4-9
4.2.10 Accessing the Math Library Functions 4-10

4.3 FLOATING-POINT LIBRARY FUNCTIONS 4-12
4.3.1 Acos 4-13
4.3.2 Acosh 4-14
4.3.3 Asin 4-15
4.3.4 Asinh 4-16
4.3.5 Atan 4-17
4.3.6 Atan2 4-18
4.3.7 Atanh 4-19
4.3.8 Bessel 4-20
4.3.9 Cabs 4-21
4.3.10 Cbrt 4-23
4.3.11 Ceil 4-24
4.3.12 Compound 4-25
4.3.13 Copysign 4-26
4.3.14 Cos 4-27
4.3.15 Cosh 4-28
4.3.16 Drem 4-29
4.3.17 Erf 4-30
4.3.18 Exp 4-31
4.3.19 Exp2 4-32
4.3.20 Expm1 4-33
4.3.21 Fabs 4-34
4.3.22 Finite 4-35
4.3.23 Floor 4-36
4.3.24 Fmod 4-37
4.3.25 Fmodf 4-38
4.3.26 Fp_getexptn 4-39
4.3.27 Fp_getround 4-40
4.3.28 Fp_gettrap 4-41
4.3.29 Fp_gmathenv 4-42
4.3.30 Fpgtrpvctr 4-44
4.3.31 Fp_procentry 4-45
4.3.32 Fp_procexit 4-46
4.3.33 Fp_setexptn 4-47
4.3.34 Fp_setround 4-48
4.3.35 Fp_settrap 4-49
4.3.36 Fp_smathenv 4-50
4.3.37 Fpstrpvctr 4-52
4.3.38 Fp_testtrap 4-53
4.3.39 Fp_tstexptn 4-54
4.3.40 Gamma 4-55
4.3.41 Hypot 4-56

4.3.42 Inf 4-57
4.3.43 Log 4-58
4.3.44 Log10 4-59
4.3.45 Log1p 4-60
4.3.46 Log2 4-61
4.3.47 Neg 4-62
4.3.48 Nextfloat 4-63
4.3.49 Pi 4-64
4.3.50 Pow 4-65
4.3.51 Randomx 4-67
4.3.52 Relation 4-68
4.3.53 Rem 4-69
4.3.54 Rint 4-71
4.3.55 Sin 4-72
4.3.56 Sinh 4-74
4.3.57 Sqrt 4-75
4.3.58 Tan 4-76
4.3.59 Tanh 4-77

Chapter 5 FPEE LIBRARY

5.1 INTRODUCTION 5-1
5.2 FPEE LIBRARY CONFIGURATIONS 5-2
5.2.1 FPEE Library Creation in a Series 32000/UNIX Environment 5-2
5.2.2 Cross-development FPEE Library Creation 5-2
5.3 INTEGRATING FPEE WITH AN APPLICATION 5-3
5.3.1 Integrating FPEE with Series 32000/UNIX Applications 5-3
5.3.2 Cross Application FPEE Integration 5-3
5.3.3 FPEE Library and the Math Library Integration 5-4
5.3.4 FPEE Error Handling Routines 5-4
5.4 FPEE OPERATIONAL DETAILS 5-5
5.4.1 Operational Overview 5-5
5.4.2 FPEE Enhancements to the FPU 5-7
5.4.3 NS32081 FPU, NS32381 FPU and FPEE 5-8
5.4.4 FPEE Program Control 5-9
5.4.5 FPEE Comparisons 5-11
5.4.6 FPEE Exception Handling 5-12
5.4.7 FPEE Rounding Modes 5-14

Appendix A SERIES 32000 STANDARD CALLING CONVENTIONS

A.1 INTRODUCTION A-1
A.2 CALLING CONVENTION ELEMENTS A-1

FIGURES

Figure 4-1. Maximum and Minimum Values for Floating-point Numbers . 4-5

TABLES

Table 2-1. Routines that Do Not Require System Calls 2-4
Table 2-2. Routines that Use Simulated System Calls 2-5
Table 4-1. Minimum and Maximum Values 4-4
Table 4-2. Global Considerations 4-11
Table 5-1. Instruction Codes 5-6
Table 5-2. FPEE Library-Implemented IEEE 754 Operations 5-8
Table 5-3. Default Return Values for Overflow Exceptions 5-15

INDEX

Chapter 1

OVERVIEW

1.1 INTRODUCTION

The GNX-Version 3 Support Libraries provide run-time support for the C, Pascal, Modula-2 and FORTRAN language compilers. The GNX Development Board (DB) library also facilitates debugging by providing input/output capability with the user terminal or host system files. Programs linked with these libraries can run on a SPLICE system connected to a target or on a *Series 32000* development board with `mon16`, `mon32`, `mon332`, `mon332b`, or `mon532` monitors respectively.

The Floating-point Enhancement and Emulation (FPEE) library enhances the Floating-point Unit (FPU) by providing additional functionality (as recommended by the ANSI/IEEE task proposal 754) for binary floating-point arithmetic. The Math Library when used with the FPEE library, provides a full IEEE 754 math environment.

The location and the names of these libraries may vary with the host operating system and are discussed in the *Series 32000 GNX — Version 3 Commands and Operations Manual* provided with the GNX tools.

These libraries are similar to the standard C, Pascal, Modula-2, or FORTRAN libraries of a UNIX® operating system. The GNX libraries and the host libraries differ in that system calls, such as fork, have been removed from the GNX library because they are not executable on a development board. Some of the other host system calls have been replaced by their simulations or implemented using the virtual I/O feature of the monitor and debugger `dbg32` (such as `open`, `read`, `write`, etc.) These libraries support most of the common I/O operations.

1.2 OPERATING SYSTEM CALL SIMULATION

The libraries provide most of the common functions of C, Pascal, Modula-2, and FORTRAN. These libraries are implemented by providing a low-level simulation of some important UNIX operating system calls. This allows programs to be compiled and tested without extensive rewriting.

The system calls implemented in this release are `open`, `close`, `creat`, `read`, `write`, `_exit`, `getdtablesize`, `lseek`, `sbrk`, and `unlink`. These system calls are dependent on the development board monitors, `dbg32` and the host operating system. The user may use the routines for debugging during the program development phase (*e.g.*, writing error messages to the terminal, storing and retrieving results from files, etc.); however, programs that depend on these system calls will not work in any other target system.

Several system calls have been given dummy implementations, that is, rather than asking the host operating system to provide actual data, the calls will always return the same values. This allows existing user-developed programs to be run on the development board with less modification but there are some restrictions.

The following is a list of dummy routines:

access	geteuid	getpid	sethostid	signal	time
execl	gethostid	gettimeofday	sethostname	stat	wait
fork	gethostname	getuid	setitimer	settimeofday	
fstat	getitimer	pause	setreuid	system	

The following is a list of restrictions in the use of dummy calls:

`isatty` Always returns "1" for stdin, stdout and stderror, and a "0" for all other streams.

`timezone` Does not look for environment variable TZNAME. The development board has no concept of environment variables.

1.3 MANUAL ORGANIZATION

Chapter 1 provides an overview of the GNX support libraries, describes the operating system call simulation and provides the documentation conventions.

Chapter 2 describes system calls.

Chapter 3 describes the C library routines.

Chapter 4 describes the math library routines.

Chapter 5 describes the floating-point enhancement and emulation library.

Appendix A describes the *Series 32000* standard calling conventions.

See the *Series 32000 GNX — Version 3 Pascal Optimizing Compiler Reference Manual*, the *Series 32000 GNX — Version 3 Modula-2 Optimizing Compiler Reference Manual*, or the *Series 32000 GNX — Version 3 FORTRAN 77 Optimizing Compiler Reference Manual* for a description of the functions for Pascal, Modula-2, or FORTRAN.

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");
```

Chapter 2

SYSTEM CALLS

2.1 INTRODUCTION

This chapter describes all of the system calls. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible return value. This is almost always -1; the individual descriptions contain detailed information.

All return codes and values from functions are of type integer unless otherwise noted. An error number is also made available in the external variable *errno*, which is not cleared on successful calls. Thus, *errno* should be tested only after the program has determined that an error has occurred.

The following is a complete list of the errors and their names as given in *errno.h* and a description of each error; these errors appear as they would on a UNIX host system:

1 EPERM Not owner
: Typically, this error indicates an attempt has been made to modify a file by someone other than its owner.

2 ENOENT No such file or directory
: This error occurs when a filename is specified and the file should exist but doesn't, or when one of the directories in a pathname does not exist.

5 EIO I/O error
: A physical I/O error occurs during a `read` or `write`. This error may in some cases occur on a call following the one to which it actually applies.

6 ENXIO No such device or address
: I/O on a special file refers to a subdevice which does not exist or is beyond the limits of the device. It may also occur when, for example, an illegal tape drive unit number is selected or a disk pack is not loaded on a drive.

9 EBADF Bad file number
: Either a file descriptor refers to no open file, or a read (respectively write) request is made to a file which is open only for writing (respectively reading).

12 ENOMEM Not enough core
: During `sbrk`, a program asks for more memory than can be supplied by the development board.

13 EACCES Permission denied
: An attempt is made to access a file in a way forbidden by the protection system.

14 EFAULT Bad address
: The system encounters a hardware fault in attempting to access the arguments of a system call.

15 ENOTBLK Block device required
: A file is mentioned where a block device is required.

16 EBUSY Mount device busy
: An attempt to mount a device that is already mounted or an attempt is made to dismount a device on which there is an active file directory.

17 EEXIST File exists
: An existing file is mentioned in an inappropriate context.

19 ENODEV No such device
: An attempt is made to apply an inappropriate system call to a device; *e.g.*, read a write-only device.

20 ENOTDIR Not a directory
: A nondirectory is specified where a directory is required.

21 EISDIR Is a directory
: An attempt to write on a directory.

22 EINVAL Invalid argument
: Some invalid argument: reading or writing a file for which `lseek` has generated a negative pointer. Also set by math functions.

23 ENFILE File table overflow
: The system's table of open files is full and temporarily no more `opens` can be accepted.

24 EMFILE Too many open files
: Limit is 24.

25 ENOTTY Not a typewriter
: The file mentioned is not a terminal or one of the other devices to which these calls apply.

26 ETXTBSY Text file busy
: An attempt to execute a pure-procedure program which is currently open for writing (or reading). Also, an attempt to open for writing a pure-procedure program that is being executed.

27 EFBIG File too large
: The size of a file exceeded the maximum (about 10^9 bytes).

28 ENOSPC No space left on device
: During a `write` to an ordinary file, there is no free space left on the device.

30 EROFS Read-only file system
: An attempt to modify a file or directory is made on a device mounted read-only.

33 EDOM Math argument
: The argument of a function in the math package is out of the domain of the function.

62 ELOOP Too many levels of symbolic links
A pathname lookup involved more than 8 symbolic links.

63 ENAMETOOLONG Filename too long
A component of the pathname or an entire pathname that exceeds the host system limitations.

2.2 DESCRIPTION OF SYSTEM CALLS

As mentioned earlier, the GNX DB support library has dummy implementations of some GENIX 4.2 system calls. These calls return dummy values within the valid range for a GENIX 4.2 operating system. For example, the `getpid()` call always returns a fixed number.

The GENIX 4.2 operating system concept of process ID, group ID, user ID, etc., is maintained in the DB support library. However, this document does not attempt to define these concepts. They are relevant only if a program that runs on UNIX or GENIX operating system is ported to a development board, in which case the user should consult the corresponding development board manuals for a complete description.

2.2.1 Routines that Do Not Require System Calls

The routines listed in Table 2-1 do not require support from the debugger. They are self-contained, or at most, call routines that are self-contained.

2.2.2 Routines that Use Simulated System Calls

The routines listed in Table 2-2 use at least one simulated system call.

2.3 SYSTEM CALL SUMMARIES

This section describes the simulated system calls in the GNX DB support library. These calls provide the user with a virtual machine very much like the GENIX 4.2 or UNIX 4.2/4.3 operating systems, so that many user programs and libraries of these systems can be directly ported to a development board.

All I/O is performed via file descriptors which are small integer numbers. When a program starts, the file descriptor 0 is associated with the console terminal in read mode (*i.e.*, the keyboard) and the file descriptors 1 and 2 are associated to the console terminal in write mode (*i.e.*, the screen). All other file descriptors are undefined or closed.

Table 2-1. Routines that Do Not Require System Calls

ROUTINE	SECTION	ROUTINE	SECTION	ROUTINE	SECTION
abort	3.2	abs	3.3	asctime	3.6
atof	3.4	atoi	3.4	atol	3.4
bcopy	3.5	bcmp	3.5	bzero	3.5
ceil	3.11	clearerr	3.10	ctime	3.6
ecvt	3.7	fabs	3.11	fcvt	3.7
feof	3.10	ferror	3.10	ffs	3.5
fileno	3.10	floor	3.11	free	3.20
frexp	3.14	gcvt	3.7	gmtime	3.6
index	3.31	insque	3.18	isatty	3.19
ldepx	3.14	localtime	3.6	longjmp	3.30
memccpy	3.21	memchr	3.21	memcmp	3.21
memcpy	3.21	memset	3.21	modf	3.14
perror	3.22	qsort	3.26	random	3.27
re_comp	3.28	re_exec	3.28	remque	3.18
rindex	3.32	setbuf	3.30	setbuffer	3.30
setjmp	3.31	srandom	3.27	strcat	3.32
strchr	3.32	strrchr	3.32	strcmp	3.32
strcpy	3.32	strlen	3.32	strncat	3.32
strncmp	3.32	strncpy	3.32	swab	3.33
sys_errlist	3.22	sys_nerr	3.22		

Table 2-2. Routines that Use Simulated System Calls

ROUTINE	SECTION	ROUTINE	SECTION	ROUTINE	SECTION
calloc	3.20	exit	3.8	fclose	3.9
fdopen	3.12	fflush	3.9	fgetc	3.16
fgets	3.17	fopen	3.12	fprintf	3.21
fputc	3.22	fputs	3.23	fread	3.13
freopen	3.12	fscanf	3.27	fseek	3.15
ftell	3.15	fwrite	3.13	getchar	3.16
gets	3.17	getw	3.16	initstate	3.25
malloc	3.20	printf	3.21	putchar	3.22
puts	3.23	putw	3.22	realloc	3.20
rewind	3.15	scanf	3.27	setlinebuf	3.28
setstate	3.25	sprintf	3.21	sscanf	3.27
timezone	3.6	ungetc	3.32		

Programs open files on the host system by using the `open()` or `creat()` system calls. A file is opened with the aid of the debugger. I/O to the file goes through the debugger. File descriptors higher than 2 are used. Programs can terminate by doing an `exit()` call, which will close all files. The `exit` call communicates to the debugger, which in turn informs the user that the program has ended and waits for the next command.

The simulated system calls allow most of the commonly used C library functions to be used, though some of them have restrictions.

While these simulated system calls and the libraries built on them provide a very easy and conceptually clean interface, they may be too bulky for applications which do not require extensive I/O support. For such applications users must trim the library according to their needs.

The system calls documented here work only in conjunction with the `dbg32`, `idbg16`, and `idbg32` debuggers. The system calls use the debugger to do I/O on the host file system. For independent programs, users need to make their own routines for I/O. They can be used as guide lines for making a system-dependent set of routines for any system. The rest of the library will function correctly as long as the simulated system calls are replaced with compatible routines.

2.3.1 Close

NAME

`close` - closes a file

SYNOPSIS

```
close(fildes)
int fildes;
```

DESCRIPTION

The `close` call closes the file on the host system with a descriptor of *fildes*.

A `close` of all of the files is automatic on `exit`, but since there is a limit to the number of active files per process (the lower of the value returned by `getdtablesize` and the limitations imposed by the host operating system), `close` is necessary for programs which deal with many *fildes*.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and the global integer variable *errno* is set to indicate the error.

ERRORS

`Close` fails if:

[EBADF] *Fildes* is not an active descriptor.

SEE ALSO

open

2.3.2 Creat

NAME

`creat` - creates a new file

SYNOPSIS

```
creat(name, mode)
char *name;
```

DESCRIPTION

`Creat` creates a new file on the host system or prepares to rewrite an existing file called *name*, given as the address of a null-terminated string. If the file did not exist, it is given mode *mode*.

To construct *mode*, OR the following:

0x400	read by owner
0x200	write by owner
0x100	execute by owner
0x070	read, write, execute by group
0x007	read write, execute by others

If the file did exist, its mode and owner remain unchanged but it is truncated to 0 length. The file is also opened for writing, and its file descriptor is returned.

Syntax of the name depends on the host system, for example, on a UNIX operating system enter:

```
creat("/u/user/test/prog.c",0x777);
```

and on a VMS operating system enter:

```
creat("dr0:[user.test]prog.c",0x777):
```

RETURN VALUE

The value -1 is returned if an error occurs. Otherwise, the call returns a non-negative descriptor which permits only writing.

ERRORS

`Creat` will fail and the file will not be created or truncated if one of the following occurs:

[EPERM]	The argument contains a byte with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[EACCES]	A needed directory does not have search permission.
[EACCES]	The file does not exist and the directory in which it is to be created is not writable.
[EACCES]	The file exists, but it is unwritable.
[EISDIR]	The file is a directory.
[EMFILE]	There are already too many files open.
[EROFS]	The named file resides on a read-only file system.
[ENXIO]	The file is a character special or block special file, and the associated device does not exist.

SEE ALSO

open, *write* and *close*

2.3.3 _Exit

NAME

`_exit` - terminates a process

SYNOPSIS

```
_exit(status)
int status;
```

DESCRIPTION

`_exit` terminates a process with the following consequences:

All of the descriptors opened in the calling process are closed.

Most C programs call the library routine `exit` (see Section 3.8) which performs cleanup actions before calling `_exit`.

RETURN VALUE

This call never returns.

SEE ALSO

exit in Chapter 3

2.3.4 Getdtablesize

NAME

`getdtablesize` - gets the size of the descriptor table

SYNOPSIS

```
nds = getdtablesize()
int nds;
```

DESCRIPTION

Each process has a fixed-size descriptor table which is guaranteed to have at least 20 slots. The entries in the descriptor table are numbered with small integers starting at 0. The call `getdtablesize` returns the size of this table.

SEE ALSO

close and *open*

2.3.5 Lseek

NAME

`lseek` - moves the read/write pointer

SYNOPSIS

```
#define L_SET       0       /* set the seek pointer */
#define L_INCR  1       /* increment the seek pointer */
#define L_XTND  2       /* extend the file size */

pos = lseek(fildes, offset, whence)
int pos;
int fildes, offset, whence;
```

DESCRIPTION

The descriptor *fildes* refers to a file on the host system or device open for reading and/or writing. `Lseek` sets the file pointer of *fildes* as follows:

If *whence* is L_SET, the pointer is set to *offset* bytes.

If *whence* is L_INCR, the pointer is set to its current location plus *offset*.

If *whence* is L_XTND, the pointer is set to the size of the file plus *offset*.

Upon successful completion, the resulting pointer location as measured in bytes from the beginning of the file is returned.

NOTES

Seeking far beyond the end of a file, then writing, creates a gap or "hole," which occupies no physical space and reads as zeros.

RETURN VALUE

Upon successful completion, a non-negative integer, the current file pointer value, is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

`Lseek` fails and the file pointer remains unchanged if:

[EBADF]	*Fildes* is not an open file descriptor.
[EINVAL]	*Whence* is not a proper value.
[EINVAL]	The resulting file pointer will be negative.

SEE ALSO

open

2.3.6 Open

NAME

open - opens a file for reading or writing or creates a new file

SYNOPSIS

```
#include <sys/file.h>

open(path, flags, mode)
char *path;
int flags, mode;
```

DESCRIPTION

Open opens the file *path* for reading and/or writing on the host system, as specified by the *flags* argument, and returns a descriptor for that file. The *flags* argument may indicate that the file is to be created if it does not already exist (by specifying the O_CREAT flag), in which case the file is created with mode *mode* as described in *creat*.

Path is the address of a string of ASCII characters representing a pathname, terminated by a null character. To form the flags specified, OR the following values:

O_RDONLY	opens for reading only
O_WRONLY	opens for writing only
O_RDWR	opens for reading and writing
O_NDELAY	does not block on open
O_APPEND	appends on each write
O_CREAT	creates file if it does not exist
O_TRUNC	truncates size to 0
O_EXCL	error if create and file exists

Opening a file with O_APPEND set appends each write on the file to the end. If O_TRUNC is specified and the file exists, the file is truncated to zero length. If O_EXCL is set with O_CREAT, and the file already exists, the open returns an error. This can be used to implement a simple exclusive access-locking mechanism. If the O_NDELAY flag is specified and the open call blocks the process (*e.g.*, waiting for carrier on a dialup line), the open returns immediately.

Upon successful completion, a non-negative integer termed a "file descriptor" is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

No process may have more than `getdtablesize()` file descriptors open simultaneously.

ERRORS

The named file is opened unless one or more of the following are true:

[EPERM]	The pathname contains a character with the high-order bit set.
[ENOTDIR]	A component of the path prefix is not a directory.
[ENOENT]	O_CREAT is not set and the named file does not exist.
[EACCES]	A component of the path prefix denies search permission.
[EACCES]	The required permissions (for reading and/or writing) are denied for the named file.
[EISDIR]	The named file is a directory, and the arguments specify it is to be opened for writing.
[EROFS]	The named file resides on a read-only file system, and the file is to be modified.
[EMFILE]	Too many open files.
[ENXIO]	The named file is a character special or block special file, and the device associated with this special file does not exist.
[ETXTBSY]	The file is a pure procedure (shared text) file that is being executed, and the `open` call requests write access.
[EFAULT]	*Path* points outside the process's allocated address space.
[EEXIST]	O_EXCL has been specified and the file exists.

SEE ALSO

close, lseek, read and *write*

2.3.7 Read

NAME

read - reads input

SYNOPSIS

```
cc = read(fildes, buf, nbytes)
int cc, fildes;
char *buf;
int nbytes;
```

DESCRIPTION

Read attempts to read *nbytes* of data from the object referenced by the descriptor *fildes* into the buffer pointed to by *buf*.

The read starts at a position given by the pointer associated with *fildes*, see *lseek*. Upon return from read, the pointer is incremented by the number of bytes actually read.

Upon successful completion, read returns the number of bytes actually read and placed in the buffer. The system guarantees to read the number of bytes requested if the descriptor references a file which has that many bytes remaining before the end-of-file, but in no other cases.

If the returned value is 0, then end-of-file has been reached.

RETURN VALUE

If successful, the number of bytes actually read is returned. Otherwise, a -1 is returned and the global variable *errno* is set to indicate the error.

ERRORS

Read fails if one or more of the following are true:

[EBADF] *Fildes* is not a valid file descriptor open for reading.

[EFAULT] *Buf* points outside the allocated address space.

SEE ALSO

open

2.3.8 Sbrk

NAME

`sbrk` - allocates memory in heap

SYNOPSIS

```
char *sbrk(incr)
int incr;
```

DESCRIPTION

`Sbrk` allocates *incr* bytes of memory from the unallocated memory on the development board between the data area of the program and its stack pointer and returns the address of the lowest byte in it. `Sbrk` assumes that writable memory is continuous and the stack is at the top of memory. `Sbrk` allocates the memory block as long as it stays 1024 bytes below the current stack pointer. `Sbrk` checks to see if the memory actually exists by writing two different numbers on the highest byte and reading them back. There is no way to deallocate this memory.

RETURN VALUE

`Sbrk` returns a pointer pointing to the start of the newly allocated area. A value of -1 is returned if *incr* bytes cannot be allocated.

ERRORS

`Sbrk` fails and no additional memory allocates if the following is true:

[ENOMEM] Insufficient memory existed on the board to support the expansion.

SEE ALSO

malloc

2.3.9 Unlink

NAME

`unlink` - removes directory entry of a file

SYNOPSIS

```
unlink(path)
char *path;
```

DESCRIPTION

`Unlink` removes the file on the host system whose name is given by *path*.

RETURN VALUE

Upon successful completion, a value of 0 is returned. Otherwise, a value of -1 is returned and *errno* is set to indicate the error.

ERRORS

The unlink succeeds unless:

[EPERM] The path contains a character with the high-order bit set.

[ENOENT] The pathname is too long.

[ENOTDIR] A component of the path prefix is not a directory.

[ENOENT] The named file does not exist.

[EACCES] Search permission is denied for a component of the path prefix.

[EACCES] Write permission is denied on the directory containing the link to be removed.

[EPERM] The named file is a directory and the effective user ID of the process is not the superuser.

[EBUSY] The entry to be unlinked is the mount point for a mounted file system.

[EROFS] The named file resides on a read-only file system.

[EFAULT] *Path* points outside the process's allocated address space.

[ELOOP] Too many symbolic links have been encountered in translating the pathname.

SEE ALSO

close

2.3.10 Write

NAME

`write` - writes on a file

SYNOPSIS

```
write(fildes, buf, nbytes)
int fildes;
char *buf;
int nbytes;
```

DESCRIPTION

`Write` attempts to write *nbytes* of data to the object referenced by the descriptor *fildes* from the buffer pointed to by *buf*.

The `write` starts at a position given by the pointer associated with *fildes*, see *lseek*. Upon return from `write`, the pointer is incremented by the number of bytes actually written.

RETURN VALUE

Upon successful completion, the number of bytes actually written is returned. Otherwise, a -1 is returned and *errno* is set to indicate the error.

ERRORS

`Write` fails and the file pointer remains unchanged if one or more of the following are true:

[EBADF]	*Fildes* is not a valid descriptor open for writing.
[EFBIG]	An attempt is made to write a file that exceeds the process' file size limit or the maximum file size.

SEE ALSO

lseek and *open*

Chapter 3

GNX DB SUPPORT LIBRARY ROUTINES

3.1 INTRODUCTION

This chapter provides a summary of the GNX DB support library routines in alphabetical order. Notice that in some cases more than one routine is described in a section. The location and name of this library may vary with each host operating system. For the location and name of this library, refer to the *Series 32000 GNX — Version 3 Commands and Operations Manual.*

3.2 ABORT

NAME

`abort` - generates a fault

SYNOPSIS

`abort( )`

DESCRIPTION

`Abort` executes an instruction which is illegal in User mode. This causes a trap that normally terminates the program execution and returns control to the debugger with a message "`Flag trap (out of range)...`".

SEE ALSO

exit

3.3 ABS

NAME

abs - integer absolute value

SYNOPSIS

```
abs(i)
int i;
```

DESCRIPTION

Abs returns the absolute value of its integer operand.

SEE ALSO

fabs in Section 3.11

CAVEATS

Applying the abs function to the most negative integer generates a result which is the most negative integer. That is,

```
"abs(0x80000000)"
```

returns 0x80000000 as a result.

3.4 ATOF

NAME

`atof, atoi, atol` - convert ASCII to numbers

SYNOPSIS

```
double atof(nptr)
char *nptr;

atoi(nptr)
char *nptr;

long atol(nptr)
char *nptr;
```

DESCRIPTION

These functions convert a string pointed to by *nptr* to floating, integer, and long integer representation respectively. The first unrecognized character ends the string.

`Atof` recognizes an optional string of spaces, then an optional sign, then a string of digits optionally containing a decimal point, then an optional `e` or `E` followed by an optionally signed integer.

`Atoi` and `atol` recognize an optional string of spaces, then an optional sign, then a string of digits.

SEE ALSO

scanf

CAVEATS

There are no provisions for overflow.

3.5 BSTRING

NAME

bcopy, bcmp, bzero, ffs - bit and byte string operations

SYNOPSIS

```
bcopy(b1, b2, length)
char *b1, *b2;
int length;

bcmp(b1, b2, length)
char *b1, *b2;
int length;

bzero(b, length)
char *b;
int length;

ffs(i)
int i;
```

DESCRIPTION

The functions `bcopy`, `bcmp`, and `bzero` operate on variable length strings of bytes. They do not check for null bytes as the routines in *string* do.

`Bcopy` copies *length* bytes from string *b1* to the string *b2*.

`Bcmp` compares byte string *b1* against byte string *b2*, returning zero if they are identical, nonzero otherwise. Both strings are assumed to be *length* bytes long.

`Bzero` places *length* 0 bytes in the string *b1*.

`Ffs` finds the first bit set passed it in the argument and returns the index of that bit. Bits are numbered starting at 1. A return value of -1 indicates the value passed is zero.

CAVEATS

The `bcmp` and `bcopy` routines take parameters backwards from `strcmp` and `strcpy`.

3.6 CTIME

NAME

`ctime, localtime, gmtime, asctime, timezone` - convert date and time to ASCII

SYNOPSIS

```
char *ctime(clock)
long *clock;

#include <sys/time.h>

struct tm *localtime(clock)
long *clock;

struct tm *gmtime(clock)
long *clock;

char *asctime(tm)
struct tm *tm;

char *timezone(zone, dst)
```

DESCRIPTION

`Ctime` converts a time pointed to by *clock* such as returned by *time* into ASCII and returns a pointer to a 26-character string in the following form. All fields have constant width.

```
Sun Sep 16 01:03:52 1973\n\0
```

`Localtime` and `gmtime` return pointers to structures containing the broken-down time. `Localtime` corrects for the time zone and possible daylight-saving time; `gmtime` converts directly to Greenwich mean time (GMT), which is the time UNIX operating systems use. `Asctime` converts a broken-down time to ASCII and returns a pointer to a 26-character string.

The structure declaration from the include file is:

```
struct tm {
        int     tm_sec;
        int     tm_min;
        int     tm_hour;
        int     tm_mday;
        int     tm_mon;
        int     tm_year;
        int     tm_wday;
        int     tm_yday;
        int     tm_isdst;
};
```

These quantities give the time on a 24-hour clock, day of month (1-31), month of year (0-11), day of week (Sunday = 0), year (19*xx*), day of year (0-365), and a flag that is nonzero if daylight-saving time is in effect.

When local time is called for, the program consults the system to determine the time zone and whether the U.S.A., Australian, Eastern European, Middle European, or Western European daylight-saving time adjustment is appropriate. The program knows about various peculiarities in time conversion over the past 10-20 years; if necessary, this understanding can be extended.

`Timezone` returns the name of the time zone associated with its first argument, which is measured in minutes westward from Greenwich. If the second argument is 0, the standard name is used; otherwise, the daylight-saving version is used. If the required name does not appear in a table built into the routine, the difference from GMT is produced; *e.g.*, in Afghanistan, `timezone` (-(60*4+30),0) is appropriate because it is four hours and thirty minutes (4:30) ahead of GMT, and the string `GMT+4:30` is produced.

3.7 ECVT

NAME

`ecvt, fcvt, gcvt` - output conversion

SYNOPSIS

```
char *ecvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *fcvt(value, ndigit, decpt, sign)
double value;
int ndigit, *decpt, *sign;

char *gcvt(value, ndigit, buf)
double value;
char *buf;
```

DESCRIPTION

`Ecvt` converts the *value* to a null-terminated string of *ndigit* ASCII digits and returns a pointer to that string. The position of the decimal point relative to the beginning of the string is stored indirectly through *decpt* (negative means to the left of the returned digits). If the sign of the result is negative, the word pointed to by *sign* is nonzero, otherwise it is zero. The low-order digit is rounded.

`Fcvt` is identical to `ecvt`, except that the correct digit has been rounded for FORTRAN F format output of the number of digits specified by *ndigits*.

`Gcvt` converts the *value* to a null-terminated ASCII string in *buf* and returns a pointer to *buf*. It attempts to produce *ndigit* significant digits in FORTRAN F format if possible (otherwise E format) ready for printing. Trailing zeros may be suppressed.

SEE ALSO

printf

CAVEATS

The return values point to static data whose content is overwritten by each call.

3.8 EXIT

NAME

`exit` - terminates a process after flushing any pending output

SYNOPSIS

exit(status)
int status;

DESCRIPTION

`Exit` terminates a process after calling the library function `fflush` to flush any buffered output. `Exit` never returns.

3.9 FCLOSE

NAME

`fclose`, `fflush` - close or flush a stream

SYNOPSIS

```
#include <stdio.h>

fclose(stream)
FILE *stream;

fflush(stream)
FILE *stream;
```

DESCRIPTION

`Fclose` causes any buffers for the named *stream* to be emptied and the file to be closed. Buffers allocated by the standard input/output system are freed.

`Fclose` is performed automatically upon calling `exit`.

`Fflush` causes any buffered data for the named output *stream* to be written to that file. The stream remains open.

SEE ALSO

fopen and *setbuf*

DIAGNOSTICS

These routines return EOF if *stream* is not associated with an output file or if buffered data cannot be transferred to that file.

3.10 FERROR

NAME

`ferror, feof, clearerr, fileno` - stream status inquiries

SYNOPSIS

```
#include <stdio.h>

feof(stream)
FILE *stream;

ferror(stream)
FILE *stream

clearerr(stream)
FILE *stream

fileno(stream)
FILE *stream;
```

DESCRIPTION

`Feof` returns nonzero when end-of-file is read on the named input *stream*, otherwise it returns zero.

`Ferror` returns nonzero when an error has occurred reading or writing the named *stream*, otherwise it returns zero. Unless cleared by `clearerr`, the error indication lasts until the stream is closed.

`Clearerr` resets the error indication on the named *stream*.

`Fileno` returns the integer file descriptor associated with the *stream*, see *open*.

These functions are implemented as macros in `ldfcn.h`; they cannot be redeclared.

SEE ALSO

fopen and *open*

3.11 FLOOR

NAME

`fabs`, `floor`, `ceil` - absolute value, floor, ceiling functions

SYNOPSIS

```
#include <math.h>

double floor(x)
double x;

double ceil(x)
double x;

double fabs(x)
double x;
```

DESCRIPTION

`Fabs` returns the absolute value $|x|$.

`Floor` returns the largest integer not greater than x.

`Ceil` returns the smallest integer not less than x.

SEE ALSO

abs

3.12 FOPEN

NAME

fopen, freopen, fdopen - open a stream

SYNOPSIS

```
#include <stdio.h>

FILE *fopen(filename, type)
char *filename, *type;

FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;

FILE *fdopen(fildes, type)
char *type;
```

DESCRIPTION

Fopen opens the file named by *filename* and associates a stream with it. Fopen returns a pointer to be used to identify the stream in subsequent operations.

Type is a character string having one of the following values:

r opens for reading.

w creates for writing.

a appends: open for writing at end-of-file, or create for writing.

In addition, each *type* may be followed by a "+" to have the file opened for reading and writing. The r+ positions the stream at the beginning of the file, w+ creates or truncates it, and a+ positions it at the end. Both reads and writes may be used on read/write streams, with the limitation that an fseek, rewind, or reading an end-of-file must be used between a read and a write or vice-versa.

Freopen substitutes the named file in place of the open *stream*. It returns the original value of *stream*. The original *stream* is closed.

Freopen is typically used to attach the preopened constant names, stdin, stdout, stderr, to specified files.

Fdopen associates a stream with a file descriptor obtained from open, or creat. The *type* of the stream must agree with the mode of the open file.

SEE ALSO

fclose

DIAGNOSTICS

`Fopen` and `freopen` return the null pointer if *filename* cannot be accessed.

3.13 FREAD

NAME

`fread`, `fwrite` - buffered binary input/output

SYNOPSIS

```
#include <stdio.h>

fread(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;

fwrite(ptr, sizeof(*ptr), nitems, stream)
FILE *stream;
```

DESCRIPTION

`Fread` reads, into a block beginning at *ptr*, *nitems* of data of the type **ptr* from the named input *stream*. It returns the number of items actually read.

If *stream* is `stdin` and the standard output is line buffered, then any partial output line will be flushed before any call to `read` to satisfy the `fread`.

`Fwrite` appends at most *nitems* of data of the type **ptr* beginning at *ptr* to the named output *stream*. It returns the number of items actually written.

SEE ALSO

fopen, *getc*, *putc*, *gets*, *puts*, *printf*, *scanf*

DIAGNOSTICS

`Fread` and `fwrite` return 0 upon end-of-file or error.

3.14 FREXP

NAME

`frexp`, `ldexp`, `modf` - split into mantissa and exponent

SYNOPSIS

```
double frexp(value, eptr)
double value;
int *eptr;

double ldexp(value, exp)
double value;

double modf(value, iptr)
double value, *iptr;
```

DESCRIPTION

`Frexp` returns the mantissa of a double *value* as a double quantity (x) of magnitude less than 1 and stores an integer n such that `value` $= x * 2^n$ indirectly through *eptr*.

`Ldexp` returns the quantity `value` $* 2^{exp}$.

`Modf` returns the positive fractional part of *value* and stores the integer part indirectly through *iptr*.

3.15 FSEEK

NAME

fseek, ftell, rewind - reposition a stream

SYNOPSIS

```
#include <stdio.h>

fseek(stream, offset, ptrname)
FILE *stream;
long offset;

long ftell(stream)
FILE *stream;

rewind(stream)
FILE *stream;
```

DESCRIPTION

Fseek sets the position of the next input or output operation on the *stream*. The new position is at the signed distance *offset* bytes from the beginning, the current position, or the end of the file, if *ptrname* has the value 0, 1, or 2.

Fseek undoes any effects of ungetc.

Ftell returns the current value of the offset relative to the beginning of the file associated with the named *stream*. It is measured in bytes.

Rewind(*stream*) is equivalent to fseek(*stream*, 0L, 0).

SEE ALSO

fopen

DIAGNOSTICS

Fseek returns -1 for improper seeks.

3.16 GETC

NAME

getc, getchar, fgetc, getw - get character or word from stream

SYNOPSIS

```
#include <stdio.h>

int getc(stream)
FILE *stream;

int getchar( )

int fgetc(stream)
FILE *stream;

int getw(stream)
FILE *stream;
```

DESCRIPTION

Getc returns the next character from the named input *stream*.

Getchar() is identical to getc(*stdin*).

Fgetc behaves like getc but is a genuine function, not a macro; it may be used to save object text.

Getw returns the next word (in a 32-bit integer) from the named input *stream*. It returns the constant EOF upon end-of-file or error, but since that is a good integer value, feof should be used to check the success of getw. Getw assumes no special alignment in the file.

SEE ALSO

fopen, *putc*, *scanf*, *fread*, *ungetc*

DIAGNOSTICS

These functions return the integer constant EOF at end-of-file or upon read error.

CAVEATS

The end-of-file return from `getchar` is incompatible with that in UNIX editions 1 through 6.

Because it is implemented as a macro, `getc` treats a *stream* argument with side effects incorrectly. In particular, the `getc(*f++);` expression is not equivalent to the `ch=*f++;getc(ch)` expression.

3.17 GETS

NAME

`gets`, `fgets` - get a string from a stream

SYNOPSIS

```
#include <stdio.h>

char *gets(s)
char *s;

char *fgets(s, n, stream)
char *s;
FILE *stream;
```

DESCRIPTION

`Gets` reads a string into *s* from the standard input stream `stdin`. The string is terminated by a newline character, which is replaced in *s* by a null character. `Gets` returns its argument.

`Fgets` reads *n*-1 characters or up to a newline character, whichever comes first, from the *stream* into the string *s*. The last character read into *s* is followed by a null character. `Fgets` returns its first argument.

SEE ALSO

puts, *getc*, *scanf*, and *fread*

DIAGNOSTICS

`Gets` and `fgets` return the constant null pointer upon end-of-file or error.

CAVEATS

`Gets` deletes a newline, `fgets` keeps it.

3.18 INSQUE

NAME

insque, remque - insert/remove element from a queue

SYNOPSIS

```
struct qelem {
     struct     qelem *q_forw;
     struct     qelem *q_back;
     char       q_data[];
};

insque(elem, pred)
struct qelem *elem, *pred;

remque(elem)
struct qelem *elem;
```

DESCRIPTION

Insque and remque manipulate queues built from double-linked lists. Each element in the queue must be in the form of struct qelem. Insque inserts *elem* in a queue immediately after *pred*; remque removes an entry *elem* from a queue.

3.19 ISATTY

NAME

`isatty` - finds name of a terminal

SYNOPSIS

`isatty(filedes)`

DESCRIPTION

`Isatty` returns 1 if *filedes* is associated with a stdin, stdout or stderr; otherwise, it returns 0.

3.20 MALLOC

NAME

malloc, free, realloc, calloc - memory allocator

SYNOPSIS

```
char *malloc(size)
unsigned size;

free(ptr)
char *ptr;

char *realloc(ptr, size)
char *ptr;
unsigned size;

char *calloc(nelem, elsize)
unsigned nelem, elsize;
```

DESCRIPTION

Malloc and free provide a simple general-purpose memory allocation package. Malloc returns a pointer to a block of at least *size* bytes beginning on a word boundary.

The argument to free is a pointer to a block previously allocated by malloc; this space is made available for further allocation, but its contents are left undisturbed. (Severe disorder will result if the space assigned by malloc is overrun or if some random number is handed to free.)

Malloc maintains multiple lists of free blocks according to size, allocating space from the appropriate list. It calls sbrk (see sbrk) to get more memory from the system when there is no suitable space already free.

Realloc changes the size of the block pointed to by *ptr* to *size* bytes and returns a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

For compatibility with older versions, realloc also works if *ptr* points to a block freed since the last call of malloc, realloc, or calloc.

Calloc allocates space for an array of *nelem* elements of size *elsize.* The space is initialized to zeros.

Each of the allocation routines returns a pointer to a space suitably aligned (after possible pointer coercion) for storage of any type of object.

DIAGNOSTICS

`Malloc`, `realloc`, and `calloc` return a null pointer (0) if there is no available memory or if the arena has been detectably corrupted by storing outside the bounds of a block. `Malloc` may be recompiled to check the arena very stringently on every transaction; those sites with a source code license may check the source code to see how this can be performed.

CAVEATS

When `realloc` returns 0, the block pointed to by *ptr* may be destroyed.

3.21 MEMORY

NAME

memccpy, memchr, memcmp, memcpy, memset - memory operations

SYNOPSIS

```
#include <memory.h>

char *memccpy (s1, s2, c, n)
char *s1, *s2;
int c, n;

char *memchr (s, c, n)
char *s;
int c, n;

int memcmp (s1, s2, n)
char *s1, *s2;
int n;

char *memcpy (s1, s2, n)
char *s1, *s2;
int n;

char *memset (s, c, n)
char *s;
int c, n;
```

DESCRIPTION

These functions operate as efficiently as possible on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Memccpy copies characters from memory area s2 into s1, stopping after the first occurrence of character c has been copied, or after n characters have been copied, whichever comes first. It returns a pointer to the character after the copy of c in s1, or a null pointer if c has not been found in the first n characters of s2.

Memchr returns a pointer to the first occurrence of character c in the first n characters of memory area s, or a null pointer if c does not occur.

`Memcmp` compares its arguments, looking at the first `n` characters only, and returns an integer less than, equal to, or greater than 0, if an `s1` is lexicographically less than, equal to, or greater than `s2`.

`Memcpy` copies `n` characters from memory area `s2` to `s1`. It returns `s1`.

`Memset` sets the first `n` characters in memory area `s` to the value of character `c`. It returns `s`.

For user convenience, all these functions are declared in the optional `<memory.h>` header file.

3.22 PERROR

NAME

`perror, sys_errlist, sys_nerr` - system error messages

SYNOPSIS

```
perror(s)
char *s;

int sys_nerr;
char *sys_errlist[];
```

DESCRIPTION

On the standard error file, `perror` produces a short error message describing the last error encountered during a call to the system from a C program. First the argument string *s* is printed, then a colon, then the message and a new-line. The argument string is the name of the program which incurred the error. The error number is taken from the external variable *errno*, which is set when errors occur but not cleared when non-erroneous calls are made.

To simplify variant formatting of messages, the vector of message strings *sys_errlist* is provided; *errno* can be used as an index in this table to get the message string without the new-line. *Sys_nerr* is the number of messages provided for in the table; it should be checked because new error codes may be added to the system before they are added to the table.

3.23 PRINTF

NAME

`printf, fprintf, sprintf` - formatted output conversion

SYNOPSIS

```
#include <stdio.h>

printf(format [ , arg ] ...)
char *format;

fprintf(stream, format [ , arg ] ...)
FILE *stream;
char *format;

sprintf(s, format  [ , arg ] ...)
char *s, format;
```

DESCRIPTION

`Printf` places output on the standard output stream `stdout`. `Fprintf` places output on the named output `stream`. `Sprintf` places "output" in the string `s`, followed by the "\0" character.

Each of these functions converts, formats, and prints its arguments after the first argument under control of the format argument. The format argument is a character string which contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which causes conversion and printing of the next successive *arg*.

Each conversion specification is introduced by the character `%`. Following the `%`, there may be

- An optional minus sign "-" which specifies *left adjustment* of the converted value in the indicated field.
- An optional digit string specifying a *field width;* if the converted value has fewer characters than the field width, it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero-padding will be performed instead of blank-padding.

In no case does a non-existent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width. Characters generated by `printf` are printed by `putc`.

- An optional period "." which serves to separate the field width from the next digit string.
- An optional digit string specifying a *precision* which specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string.
- An optional "#" character specifying that the value should be converted to an alternate form. For `c`, `d`, `s`, and `u` conversions, this option has no effect. For `o` conversions, the precision of the number is increased to force the first character of the output string to a zero. For `x(X)` conversion, a nonzero result has the string `0x(0X)` added to the front. For `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal point, even if no digits follow the point (normally, a decimal point appears only in the results of those conversions if a digit follows the decimal point). For `g` and `G` conversions, trailing zeros are not removed from the result as they would otherwise be.
- The character `l` specifying that a following `d`, `o`, `x`, or `u` corresponds to a long integer *arg*.
- A character which indicates the type of conversion to be applied.

A field width or precision may be "*" instead of a digit string. In this case, an integer *arg* supplies the field width or precision.

The conversion characters and their meanings are:

dox The integer *arg* is converted to decimal, octal, or hexadecimal notation respectively.

f The float or double *arg* is converted to decimal notation in the style "`[-]ddd.ddd`" where the number of d's after the decimal point is equal to the precision specification for the argument. If the precision is missing, 6 digits are given; if the precision is explicitly 0, no digits and no decimal point are printed.

e The float or double *arg* is converted in the style "`[-]d.ddde±dd`" where there is one digit before the decimal point and the number after is equal to the precision specification for the argument; when the precision is missing, 6 digits are produced.

g The float or double *arg* is printed in style `d`, in style `f`, or in style `e`, whichever gives full precision in minimum space.

- **c** The character *arg* is printed.
- **s** *Arg* is taken to be a string (character pointer), and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is 0 or missing, all characters up to a null are printed.
- **u** The unsigned integer *arg* is converted to decimal and printed (the result will be in the range 0 through MAXUINT, where MAXUINT equals 4294967295 on a VAX-11 and 65535 on a PDP-11).
- **%** Print a "%" percent sign; no argument is converted.

Example: To print a date and time in the form `Sunday, July 3, 10:02`, where *weekday* and *month* are pointers to null-terminated strings:

```
printf("%s, %s %d, %02d:%02d", weekday, month, day, hour, min);
```

To print π to 5 decimals:

```
printf("pi = %.5f", 4*atan(1.0));
```

SEE ALSO

putc, *scanf*, *ecvt*

CAVEATS

Very wide fields (>128 characters) fail.

3.24 PUTC

NAME

putc, putchar, fputc, putw - put character or word on a stream

SYNOPSIS

```
#include <stdio.h>

int putc(c, stream)
char c;
FILE *stream;

putchar(c)

fputc(c, stream)
FILE *stream;

putw(w, stream)
int w;
FILE *stream;
```

DESCRIPTION

Putc appends the character *c* to the named output *stream*. It returns the character written.

Putchar(c) is defined as putc(c, *stdout*).

Fputc behaves like putc, but is a genuine function rather than a macro.

Putw appends word (that is, int) w to the output *stream*. It returns the word written. Putw neither assumes nor causes special alignment in the file.

SEE ALSO

fopen, *fclose*, *getc*, *puts*, *printf*, *fread*

DIAGNOSTICS

These functions return the constant EOF upon error.

CAVEATS

Because it is implemented as a macro, `putc` improperly treats a *stream* argument with side effects. In particular,

```
putc(c, *f++);
```

doesn't work logically.

Errors can occur long after the call to `putc`.

3.25 PUTS

NAME

`puts`, `fputs` - put a string on a stream

SYNOPSIS

```
#include <stdio.h>

puts(s)
char *s;

fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION

`Puts` copies the null-terminated string `s` to the standard output stream *stdout* and appends a newline character.

`Fputs` copies the null-terminated string *s* to the named output *stream*.

Neither routine copies the terminal null character.

SEE ALSO

fopen, *gets*, *putc*, *printf*, and *fwrite*

CAVEATS

`Puts` appends a newline, `fputs` does not.

3.26 QSORT

NAME

qsort - quicker sort

SYNOPSIS

```
qsort(base, nel, width, compar)
char *base;
int nel,width;
int (*compar)();
```

DESCRIPTION

Qsort is an implementation of the quicker sort algorithm. The first argument is a pointer to the base of the data, the second is the number of elements, the third is the width of an element in bytes, and the last is the name of the comparison routine to be called. Qsort contains two arguments which are pointers to the elements being compared. The routine must return an integer less than, equal to, or greater than 0 according to whether the first argument is less than, equal to, or greater than the second.

3.27 RANDOM

NAME

random, srandom, initstate, setstate - random number generator; routines for changing generators

SYNOPSIS

```
long  random()

srandom(seed)
int  seed;

char  *initstate(seed, state, n)
unsigned  seed;
char  *state;
int  n;

char  *setstate(state)
char  *state;
```

DESCRIPTION

Random uses a nonlinear additive feedback random number generator employing a default table of size 31 long integers to return successive pseudo-random numbers in the range of 0 to 2^{31}-1. The period of this random number generator is very large, approximately 16*(2^{31}-1).

All the bits generated by random are usable. For example, **random()&01** will produce a random binary value.

Random will by default produce a sequence of numbers that can be duplicated by calling srandom with *1* as the *seed*.

The *initstate* routine allows a state array, passed in as an argument, to be initialized for future use. The size of the state array (in bytes) is used by initstate to decide how sophisticated a random number generator it should use -- the more state, the better the random numbers will be. (Current "optimal" values for the amount of state information are 8, 32, 64, 128, and 256 bytes; other amounts will be rounded down to the nearest known amount. Using less than 8 bytes will cause an error.) The seed for the initialization (which specifies a starting point for the random number sequence and provides for restarting at the same point) is also an argument. Initstate returns a pointer to the previous state information array.

Once a state has been initialized, the `setstate` routine provides for rapid switching between states. `Setstate` returns a pointer to the previous *state* array; its argument state array is used for further random number generation until the next call to `initstate` or `setstate`.

Once a state array has been initialized, it may be restarted at a different point either by calling `initstate` (with the desired seed, the state array, and its size), or by calling both `setstate` (with the state array) and `srandom` (with the desired seed). The advantage of calling both `setstate` and `srandom` is that the size of the state array does not have to be remembered after it is initialized.

With 256 bytes of state information, the period of the random number generator is greater than 2^{69}, which should be sufficient for most purposes.

DIAGNOSTICS

If `initstate` is called with less than 8 bytes of state information, or if `setstate` detects that the state information has been garbled, error messages are printed on the standard error output.

3.28 REGEX

NAME

re_comp, re_exec - regular expression handler

SYNOPSIS

```
char *re_comp(s)
char *s;

re_exec(s)
char *s;
```

DESCRIPTION

Re_comp compiles a string into an internal form suitable for pattern matching. Re_exec checks the argument string against the last string passed to re_comp.

Re_comp returns 0 if the string s is compiled successfully; otherwise a string containing an error message is returned. If re_comp is passed 0 or a null string, it returns without changing the currently compiled regular expression.

Re_exec returns 1 if the string *s* matches the last compiled regular expression, 0 if the string *s* failed to match the last compiled regular expression, and –1 if the compiled regular expression is invalid (indicating an internal error).

The strings passed to both re_comp and re_exec may have trailing or embedded newline characters; they are terminated by nulls.

DIAGNOSTICS

Re_exec returns –1 for an internal error.

`Re_comp` returns one of the following strings if an error occurs:

```
No previous regular expression,
Regular expression too long,
unmatched \(,
missing ],
too many \(\) pairs,
unmatched \).
```

3.29 SCANF

NAME

scanf, fscanf, sscanf - formatted input conversion

SYNOPSIS

```
#include <stdio.h>

scanf(format [ , pointer ] ... )
char *format;

fscanf(stream, format [ , pointer ] ... )
FILE *stream;
char *format;

sscanf(s, format [ , pointer ] ... )
char *s, *format;
```

DESCRIPTION

Scanf reads from the standard input stream stdin. Fscanf reads from the named input stream. Sscanf reads from the character string s. Each function reads characters, interprets them according to a format, and stores the results in its arguments. Each expects arguments as a control string *format*, described below, and a set of *pointer* arguments indicating where the converted input should be stored.

The control string normally contains conversion specifications, which are used to direct interpretation of input sequences. The control string may contain:

1. Blanks, tabs, or newlines, which match optional white space in the input.
2. An ordinary character (not %) which must match the next character of the input stream.
3. Conversion specifications, consisting of the character %, an optional assignment suppressing character *, an optional numerical maximum field width, and a conversion character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding argument, unless assignment suppression has been indicated by `*`. An input field is defined as a string of nonspace characters; it extends to the next inappropriate character or until the field width, if specified, is exhausted.

The conversion character indicates the interpretation of the input field; the corresponding pointer argument must normally be of a restricted type. The following conversion characters are legal:

% a single "%" is expected in the input at this point; no assignment is performed.

d a decimal integer is expected; the corresponding argument should be an integer pointer.

o an octal integer is expected; the corresponding argument should be an integer pointer.

x a hexadecimal integer is expected; the corresponding argument should be an integer pointer.

s a character string is expected; the corresponding argument should be a character pointer pointing to an array of characters large enough to accept the string and a terminating "\0," which will be added. The input field is terminated by a space character or a newline.

c a character is expected; the corresponding argument should be a character pointer. The normal skip over space characters is suppressed in this case; to read the next nonspace character, try "%1s." If a field width is given, the corresponding argument should refer to a character array, and the indicated number of characters is read.

f or **e**
a floating-point number is expected; the next field is converted accordingly and stored through the corresponding argument, which should be a pointer to a *float*. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of an `E` or `e` followed by an optionally signed integer.

[indicates a string not to be delimited by space characters. The left bracket is followed by a set of characters and a right bracket; the characters between the brackets define a set of characters making up the string. If the first character is not circumflex (^), the input field is all characters until the first character not in the set between the brackets. If the first character after the left bracket is ^, the input field is all characters until the first character which is in the

remaining set of characters between the brackets. The corresponding argument must point to a character array.

The conversion characters d, o, and x may be capitalized or preceded by l to indicate that a pointer to long rather than to int is in the argument list. Similarly, the conversion characters e or f may be capitalized or preceded by l to indicate a pointer to double rather than to float. The conversion characters d, o, and x may be preceded by h to indicate a pointer to short rather than to int.

The scanf functions return the number of successfully matched and assigned input items. This can be used to decide how many input items have been found. The constant EOF is returned upon end-of-input; note that this is different from 0, which means that no conversion has been performed; if conversion had been intended, it has been frustrated by an inappropriate character in the input.

For example, the call

```
int i; float x; char name[50];
scanf("%d%f%s", &i, &x, name);
```

with the input line

```
25   54.32E-1  thompson
```

will assign to *i* the value 25, *x* the value 5.432, and *name* will contain thompson\0. Or,

```
int i; float x; char name[50];
scanf("%2d%f%*d%[1234567890]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to *i*, 789.0 to *x*, skip "0123," and place the string 56\0 in *name*. The next call to getchar will return a.

SEE ALSO

atof, *getc*, *printf*

DIAGNOSTICS

The `scanf` functions return EOF on end-of-input and a short count for missing or illegal data items.

CAVEATS

The success of literal matches and suppressed assignments is not directly determinable.

3.30 SETBUF

NAME

`setbuf, setbuffer, setlinebuf` - assign buffering to a stream

SYNOPSIS

```
#include <stdio.h>

setbuf(stream, buf)
FILE *stream;
char *buf;

setbuffer(stream, buf, size)
FILE *stream;
char *buf;
int size;

setlinebuf(stream)
FILE *stream;
```

DESCRIPTION

The three types of buffering available are unbuffered, block buffered, and line buffered. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written; when it is block buffered, many characters are saved up and written as a block; when it is line buffered, characters are saved up until a newline is encountered or input is read from `stdin`. `Fflush` (see `fclose`) may be used to force the block out early. Normally all files are block buffered. A buffer is obtained from `malloc` upon the first `getc` or `putc` on the file. If the standard stream `stdout` refers to a terminal, it is line buffered. The standard stream `stderr` is always unbuffered.

`Setbuf` is used after a stream has been opened but before it is read or written. The character array *buf* is used instead of an automatically allocated buffer. If *buf* is the constant null pointer, input/output will be completely unbuffered. A manifest constant `BUFSIZ` tells how big an array is needed:

```
char buf[BUFSIZ];
```

`Setbuffer`, an alternate form of `setbuf`, is used after a stream has been opened but before it is read or written. The character array *buf* whose size is determined by the *size* argument is used instead of an automatically allocated buffer. If *buf* is the constant null pointer, input/output will be completely unbuffered.

`Setlinebuf` is used to change *stdout* or *stderr* from block buffered or unbuffered to line buffered. Unlike `setbuf` and `setbuffer`, it can be used at any time that the file descriptor is active.

A file can be changed from unbuffered or line buffered to block buffered by using `freopen` (see `fopen`). A file can be changed from block buffered or line buffered to unbuffered by using `freopen` followed by `setbuf` with a buffer argument of null.

SEE ALSO

fopen, *getc*, *putc*, *malloc*, *fclose*, *puts*, *printf*, *fread*

CAVEATS

The standard error stream should be line buffered by default.

3.31 SETJMP

NAME

`setjmp`, `longjmp` - nonlocal goto

SYNOPSIS

```
#include <setjmp.h>
setjmp(env)
jmp_buf env;

longjmp(env, val)
jmp_buf env;
```

DESCRIPTION

These routines are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

`Setjmp` saves its stack environment in `env` for later use by `longjmp`. It returns value 0.

`Longjmp` restores the environment saved by the last call of `setjmp`. It then returns in such a way that execution continues as if the call of `setjmp` had just returned the value `val` to the function that invoked `setjmp`. (`Setjmp` must not have returned in the interim.) All accessible data have values as soon as `longjmp` is called.

CAVEATS

`Setjmp` does not save current notion of whether the process is executing on the user stack or interrupt stack. If `setjmp` and `longjmp` are performed while the process is executing on different stacks, the result will be unpredictable.

3.32 STRING

NAME

`index`, `rindex`, `strcat`, `strncat`, `strcmp`, `strncmp`, `strcpy`, `strncpy`, `strlen`, `strchr`, `strrchr` - string operations

SYNOPSIS

```
#include <strings.h>

char *strcat(s1, s2)
char *s1, *s2;

char *strncat(s1, s2, n)
char *s1, *s2;

strcmp(s1, s2)
char *s1, *s2;

strncmp(s1, s2, n)
char *s1, *s2;

char *strcpy(s1, s2)
char *s1, *s2;

char *rindex(s, c)
char *s, c;

char *strncpy(s1, s2, n)
char *s1, *s2;

strlen(s)
char *s;

char *strchr(s, c)
char *s, c;

char *strrchr(s, c)
char *s, c;

char *index(s, c)
char *s, c;
```

DESCRIPTION

These functions operate on null-terminated strings. They do not check for overflow of any receiving string.

`Strcat` appends a copy of string *s2* to the end of string *s1*. `Strncat` copies at most `n` characters. Both return a pointer to the null-terminated result.

`Strcmp` compares its arguments and returns an integer greater than, equal to, or less than 0, if an *s1* is lexicographically greater than, equal to, or less than *s2*. `Strncmp` makes the same comparison but looks at most *n* characters.

`Strcpy` copies string *s2* to *s1*, stopping after the null character has been moved. `Strncpy` copies exactly *n* characters, truncating or null-padding *s2;* the target may not be null-terminated if the length of *s2* is *n* or more. Both return *s1*.

`Strlen` returns the number of non-null characters in *s*.

`Strchr` (*strrchr*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string. The null character terminating a string is considered to be part of the string.

`Index` (*rindex*) returns a pointer to the first (last) occurrence of character *c* in string *s*, or zero if *c* does not occur in the string.

3.33 SWAB

NAME

swab - swaps bytes

SYNOPSIS

```
swab(from, to, nbytes)
char *from, *to;
```

DESCRIPTION

Swab copies *nbytes* bytes pointed to by *from* to the position pointed to by *to*, exchanging adjacent even and odd bytes. It is useful for carrying binary data between PDP11's and other machines. *Nbytes* should be even.

3.34 UNGETC

NAME

`ungetc` - pushes character back into input stream

SYNOPSIS

```
#include <stdio.h>

ungetc(c, stream)
FILE *stream;
```

DESCRIPTION

`Ungetc` pushes the character `c` back on an input stream. That character will be returned by the next `getc` call on that stream. `Ungetc` returns `c`.

One character of pushback is guaranteed provided something has been read from the stream and the stream is actually buffered. Attempts to push EOF are rejected.

`Fseek` erases all memory of pushed back characters.

SEE ALSO

getc, *setbuf*, *fseek*

DIAGNOSTICS

`Ungetc` returns EOF if it can't push a character back.

Chapter 4

FLOATING-POINT LIBRARY

4.1 INTRODUCTION

This chapter describes the single-precision and double-precision math library functions for the NS32081 and NS32381 floating-point units. The math libraries, `libm.a`, `libXm.a`, `lib381m.a`, and `libX381m.a` contain the same standard math functions. The functions in `libm.a` and `libXm.a` support the NS32081 floating-point unit. The functions in `lib381m.a` and `libX381m.a` support the NS32381 floating-point unit. Throughout this manual, the term "math library" refers to `libm.a`, `libXm.a`, `lib381m.a`, and `libX381m.a`.

There are separate implementations for single-precision and double-precision floating-point arithmetic. The names of the double-precision functions are listed below; the names of the single-precision functions are the same as the double-precision functions prefixed with an `f`. For example, the single-precision version of `sin` is `fsin`. There is one exception to this naming convention, `nextdouble` (double-precision) and `nextfloat` (single-precision).

acos	cabs	drem	floor	log1p	rint
acosh	cbrt	erf	fmod	log2	sin
asin	ceil	exp	fmodf	neg	sinh
asinh	compound	exp2	hypot	pi	sqrt
atan	copysign	expm1	inf	pow	tan
atan2	cos	fabs	log	relation	tanh
atanh	cosh	finite	log10	rem	

The following functions are common to both the single- and double-precision libraries:

gamma bessel randomx

The following environment access functions are also common to both the single- and double-precision arithmetic:

fp_getexptn	fp_procentry	fp_smathenv
fp_getround	fp_procexit	fpstrpvctr
fp_gettrap	fp_setexptn	fp_testtrap
fp_gmathenv	fp_setround	fp_tstexptn
fpgtrpvctr	fp_settrap	

See Sections 4.2.10 and 4.3, describing the use of these functions from a program written in C, Pascal, FORTRAN or Modula-2.

The standard calling conventions as described in Appendix A are used to call math library functions. This protocol includes the convention of passing only double-precision floating-point arguments in external procedure and function calls. Because of this, when a single-precision procedure or function is called, a hardware instruction is invoked whenever it is necessary to convert an argument from single-precision to double-precision. If this instruction is executed with a reserved operand, the result is an immediate invalid-operation trap. It is not possible for the user to disable this trap; therefore, with the combination of the math library and the floating-point emulation library, the user may achieve compliance with only the IEEE 754 Standard for Floating-Point Arithmetic for double-precision arithmetic.

Major problems result when the user is unable to effectively use the single-precision version of the `relation` function; `frelation` returns "unordered" when passed a quiet NAN as an argument, and `ffinite` returns a zero when passed an infinity or a NAN as an argument. These routines, if the source code is available, can be included in a program as local routines to avoid the conversion problem.

4.2 DETAILS AND USE OF THE MATH LIBRARY

This section describes integer and floating-point number formats, reserved operand values and conditions, and techniques for handling floating-point error situations according to the ANSI/IEEE "Standard for Binary Floating-point Arithmetic" (ANSI/IEEE Std 754-1985).

4.2.1 Number Formats

The *Series 32000* architecture implements three lengths for integers and two lengths for floating-point numbers. Reserved operand values are floating-point numbers that represent values outside the *Series 32000* architecturally possible range.

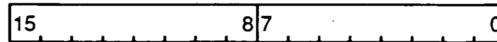
4.2.2 Integer Formats

The most significant bit in the integer format is a sign bit used to implement negative integers in two's-complement representation. The math library operates on integers in three formats.

Byte Format:

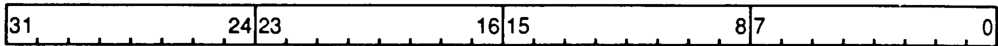
Byte format represents values from negative 128 through positive 127.

Word Format:



Word format represents values from negative 32768 through positive 32767.

Double-word Format:



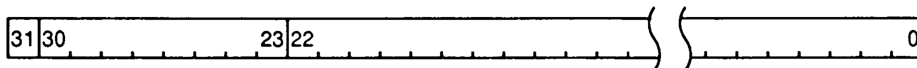
Double-word format represents values from negative 2147483648 through positive 2147483647.

4.2.3 Floating-point Formats

The math library operates on single-precision and double-precision floating-point numbers. Single-precision and double-precision formats have three parts:

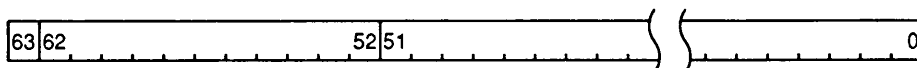
- Sign
- Exponent
- Fraction

Single-precision numbers have a 1-bit sign, an 8-bit exponent (between -126 and +127), and a 23-bit fraction, as follows:



Single-precision math functions return a single-precision number as the result.

Double-precision numbers have a 1-bit sign, an 11-bit exponent (between -1022 and +1023), and a 52-bit fraction, as follows:



Double-precision math functions return a double-precision number as the result.

The math library operates on the valid range of floating-point numbers. All valid floating-point numbers are normalized numbers. Normalized numbers have two characteristics which distinguish them from invalid floating-point range numbers (reserved operands). Normalized numbers have an assumed leading 1 in the fraction part of the format and the exponent is neither all 0's nor all 1's. The mantissa of a

floating-point number is formed by prefixing a 1 to the fraction. For example, a single-precision floating-point fraction 11000000111011111010010 after prefixing a 1 to the mantissa becomes 1.11000000111011111010010. The binary point is between the assumed first bit and the most significant bit of the fraction. A bias value is added to the exponent before it is stored in the exponent field of the floating-point number. The bias value added to each exponent is:

127 for single-precision
1023 for double-precision

The minimum and maximum decimal and hexadecimal values for single- and double-precision floating-point numbers are given in Table 4-1 and shown on a number line in Figure 4-1.

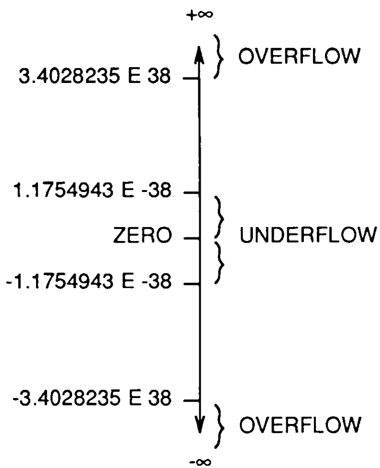
Though zero is a valid floating-point number, it is not a normalized number. A zero is represented by all 0's in the exponent and fraction. The sign bit can be either 0 (positive zero) or 1 (negative zero). Normally, positive and negative zero are equivalent, but in special cases, such as divide by zero, they are distinguishable.

Table 4-1. Minimum and Maximum Values

	HEXADECIMAL VALUE	DECIMAL NUMBER VALUE
DOUBLE-PRECISION		
Max(normal)	7FEFFFFF FFFFFFFF	1.797693134862316 E 308
Min(normal)	00100000 00000000	2.225073858507201 E -308
Min(denormal)	00000000 00000001	4.940656458412465 E -324
SINGLE-PRECISION		
Max(normal)	7F7FFFFF	~3.4028235 E 38
Min(normal)	00800000	~1.1754943 E -38
Min(denormal)	00000001	~1.4012984 E -45

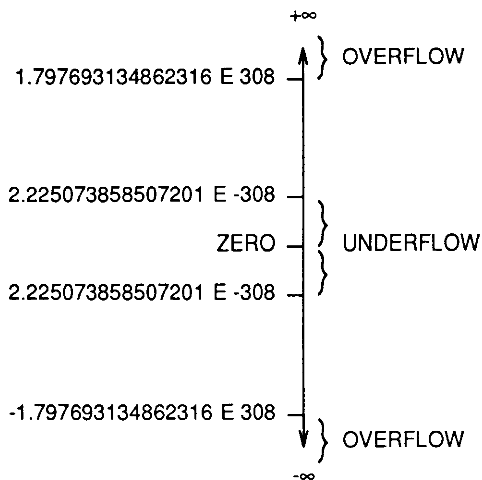
NOTE: 1. These values are positive. The sign bit does not affect the absolute magnitude of the limits. The negative limits can be determined by adding a sign bit to the hexadecimal value and affixing a negative sign in front of the floating-point value.

2. The binary exponent is the value of the exponent within the given hexadecimal value. The maximum value is represented by all bits set within the exponent field. For both single-precision and double-precision formats, this value indicates not-a-number or infinity.



PRECISION = 2^{-23} = 1.1920929 E -7

SINGLE-PRECISION



PRECISION = 2^{-52} = 2.2204460492503132 E -52

DOUBLE-PRECISION

HM-01-0-U

Figure 4-1. Maximum and Minimum Values for Floating-point Numbers

4.2.4 Reserved Operand Values and Operations

Reserved operand values represent values and situations outside the architecturally legal range of floating-point numbers. The architecturally legal range is a function of the size of both the exponent field and the fraction field. There are three types of reserved operands:

- Not-a-Number (NAN)
- Infinity (plus or minus)
- Denormalized number

For double-precision arithmetic, the math library implements some functions from the ANSI/IEEE-754 Standard for handling reserved operand situations. Full IEEE 754 functionality for double-precision is achieved only when the math library is used in conjunction with *Series 32000* Floating-point Enhancement and Emulation Library (FPEE Libraries). The name of the FPEE library varies with host system; see Chapter 5 for the name of the FPEE library for each specific host and to gain a full understanding of GNX floating-point support. The key distinctions and differences between the math library and the FPEE libraries follow the brief description of the IEEE 754 floating-point system.

IEEE 754 requires that exceptions (arithmetic operations on reserved operands) cause a signal. The signal may be either the setting of an exception status flag, or taking a trap, or both. The exact action must be under control of the application program. For example, the application program can specify setting the exception status flag, but no trapping, for a specific type of exception. In this case, program execution continues despite the exception, and the numerical result of the operation causing the exception is the appropriate IEEE 754 recommended value, typically either a NAN, or a signed infinity. If trapping is disabled, the application program can look at the exception status flags to determine if an exception occurred. The propagation of numerically meaningless values (*i.e.*, NANs or infinities) is strictly for retrospective diagnostic reasons and rarely serves any meaningful purpose for the real world. A finished application most likely runs with all traps enabled since any trap is cause for concern, and program execution stops as soon as possible after the exception. The GNX floating-point support meets these requirements for double-precision arithmetic.

IEEE 754 defines five types of exceptions: underflow, overflow, divide by zero, inexact result, and invalid operation. IEEE 754 requires functions to enable/disable each of the five traps and functions to read/clear each of the five exception status flags. These traps and exceptions called math environment variables are controlled by a series of functions provided in the math library. These functions are named fp_*function_name*, where *function_name* is a descriptive phrase.

The *Series 32000* FPUs (*i.e.*, NS32081 and NS32381) provide flags only for underflow, inexact trap enable, and inexact trap status. The FPU always traps for overflow, divide by zero, and invalid operation exceptions. In no case do the *Series 32000* FPUs handle a trap. A trap halts application program execution.

The math library alone provides the functions to access and control the IEEE 754 math environment variables, but it is the FPEE libraries which implement the trap handling functionality. An application that uses only the math library cannot rightfully be considered in compliance with IEEE 754. An application that uses both the math library

and an FPEE library for double-precision arithmetic is in compliance with IEEE 754 math environment requirements.

The remainder of this section describes the reserved operand format and pertinent information.

4.2.5 Not a Number (NAN)

Not a Number (NAN) is the result of an invalid operation. Invalid operations include zero multiplied by infinity, division of infinity by infinity, and any arithmetic operation on a NAN.

There are two types of NANs: Quiet and Signaling. The quiet NAN (QNAN) does not cause a trap or set the invalid operation status flag. The QNAN is propagated quietly through floating-point operations and is useful for retrospective diagnosis.

The quiet NAN verses signaling NAN (SNAN) distinction is implemented in the FPEE library. Many of the math library mathematical functions can return QNANs, but this value propagates through subsequent calculations only if the FPEE library is used. The FPEE library implements the trap handler which processes QNANs and resumes application program execution after a QNAN causes an FPU trap. QNANs always cause an FPU trap since the *Series 32000* FPUs do not distinguish between QNANs and SNANs.

	Signaling	Quiet
sign	0 or 1	0 or 1
exponent	All 1s	All 1s
fraction	1 followed by any combination of 0s and 1s	0 followed by any combination of 0s and 1s where at least one of the following bits must be a 1

4.2.6 Infinity

Some operations (*i.e.*, those which cause overflow) produce a value representing infinity. When the FPEE library is used, it allows infinity to be used in operations such as comparison and multiplication. Infinity has the following formats:

Positive Infinity	Negative Infinity	
sign	0	1
exponent	All 1s	All 1s
fraction	All 0s	All 0s

The FPEE library supports a number system in which two infinities exist: a positive infinity at the positive end of the number line and a negative infinity at the negative end of the number line.

4.2.7 Denormalized Numbers

A denormalized number is a value for numbers which are too small to be correctly represented in standard single- or double-precision format. These numbers are produced to avoid underflow (they actually allow a gradual underflow which decreases the region shown in Figure 4-1). Denormalized numbers are characterized by an assumed 0 instead of 1 at the beginning of the fraction. Operations such as division can generate denormalized numbers. Denormalized numbers have the following format:

sign	0 or 1
exponent	All 0s
fraction	Any combination of 0s and 1s but it is read as less than 1

Any operation that causes underflow creates a denormalized number and sets the underflow status flag. The underflow does not cause a trap unless the underflow trap enable flag is set. A subsequent operation using the denormalized number causes an invalid operation trap because it is an operation on a reserved operand. The FPEE library exception trap handler normalizes the underflowed number by shifting the fraction to the left and setting the exponent to its minimum, and if the invalid operation exception trap is disabled, program execution continues with a very small value that is not a reserved operand and therefore is suitable for floating-point numerical operations. It is the responsibility of the application program to check the exception flags in the floating-point status register to determine if underflow and an operation on a denormalized value (invalid operation) occurred. The FPUs return 0.0 if the underflow trap enable flag is not set.

4.2.8 Math Environment Control Function

The math library provides a number of math environment control functions which when used in conjunction with the FPEE library, provide the full range of IEEE 754 features. All of the math environment control functions begin with "fp_" followed by a descriptive phrase. These functions provide control over the three basic set of math environment variables:

- Exception trap enable/disable
- Exception status flags
- Rounding mode

The exception trap enable/disable functions provide application program access to the FPU's floating-point status register fields which enable or disable traps on exceptions. These functions control trapping for the five exceptions: overflow, underflow, divide by zero, inexact result, and invalid operation. If the FPEE library is not used, only underflow and inexact result traps are meaningful, but only in a minimal sense since no trap handler is available to handle invalid operation exceptions. Without the FPEE trap handler, the first underflow or inexact result effectively prevents further program execution irrespective of the trap enable/disable setting, since a subsequent operation in the application program that uses the underflowed or inexact result value causes an invalid operation trap.

The exception status flag functions provide application program access to the FPU's floating-point status register fields which report whether an exception has occurred. These functions can either report the value of the field (*i.e.* flag status), or set it to either a 1 or a 0. There are five status flags: one for each of the five exceptions. Once an exception sets its status flag, the flag stays set until explicitly reset by the application program.

The FPEE library implements most of the functionality associated with the exception traps and status fields. The FPEE library uses the software field of the FPU's floating-point status register to implement the trap enable/disable and exception status for overflow, divide by zero, and invalid result exceptions.

The functions for rounding mode are applicable whether the FPEE library is used or not.

4.2.9 Using the Math Environment Functions

Using math environment functions is not mandatory. If the math environment functions are not used, the application program runs with whichever default conditions the run-time system provides. The default conditions are system dependent and may vary. Typically, these are minimal and not IEEE 754-based, but are adequate for many applications.

Mathematically sophisticated applications do require the discipline provided by the math environment functions. An IEEE 754 math-environment-based application begins execution by first calling `fp_procentry()` before any application calculations. The `fp_procentry()` function saves the current math environment (exception status flags, Rounding mode, and trap flags) and sets the FPU's floating-point status register

to the IEEE 754 default (clears all exception status flags, sets Rounding mode to nearest, and disables all traps). The saved math environment is kept for restoration when the application program completes (`fp_procexit()` is used as the last statement in the application program). At critical points along the application program's execution, checking for exceptions is performed using an appropriate function such as `fp_getexptn()`. If an exception is found, application program error functions provide whatever service is necessary.

The `fp_procentry()` and `fp_procexit()` functions surround any atomic region of code, and the pair may be used as often as required to simplify or implement any special error handling functions. Though `fp_procentry()`s and `fp_procexit()`s may be linearly nested, this normally complicates tracking the last saved math environment; therefore, this practice is not recommended.

The command summaries provide specific information on all the math environment functions.

4.2.10 Accessing the Math Library Functions

High-Level Languages (HLL) access the math library functions in one of two ways.

In languages like C or Modula-2, that do not have a predefined list of math function names, programs call the math library functions directly. C and Modula-2 programs treat a math library function as just another external function. C programmers should include the `math.h` file, which declares the functions and constants mentioned in the following paragraphs.

Languages like FORTRAN or Pascal have predefined (intrinsic) lists of math function names and in certain cases must access the math library through the interface library provided with the compiler. The interface library maps the HLL predefined name to the corresponding math function name in the math library. In FORTRAN's case, the compiler and/or the interface library also implement the parameter-passing convention expected by the math library function. (FORTRAN normally passes all parameters by address, but recognizes the intrinsic routines and passes their parameters by value, as is expected by the math library).

The following sections provide the Pascal, FORTRAN, Modula-2, and C calling sequence for the math library functions. In the case of FORTRAN intrinsic routines, only the generic function name is mentioned. Refer to the *Series 32000 GNX — Version 3 FORTRAN 77 Optimizing Compiler Reference Manual* for a detailed description of the alternative ways to call these intrinsic functions. Table 4-2 describes global considerations, which must be observed in order to use any of the routines described in this chapter. This information will not be repeated in each sub-section.

Table 4-2. Global Considerations

LANGUAGE	INSERT THE FOLLOWING TO USE THE MATH LIBRARY	EXAMPLE PROGRAM

C	#include <math.h>	#include <math.h>
		main() {
		  double x = 0.0;
		  printf(sin(x));
		}

FORTRAN		double precision x x = 0.0 print *, sin(x) end

Pascal		program try(output);
		var x: longreal;
		begin
		  x := 0.0;
		  writeln(sin(x));
		end.

Modula-2	FROM libm IMPORT *name*;	MODULE TRY; FROM libm IMPORT sin; FROM InOut IMPORT WriteLn; FROM RealInOut IMPORT WriteReal; VAR X: LONGREAL; BEGIN X := 0.0; WriteReal(sin(X),10); WriteLn; END TRY.

To access nonstandard library routines, refer to Appendix B of the various language reference manuals.

4.3 FLOATING-POINT LIBRARY FUNCTIONS

This section describes all the floating-point library functions. These descriptions include calling sequence, accuracy, and handling for special case requirements. The following notations are used in the description of the library functions:

ULP Unit in the last place of a floating-point number.

NAN Not a Number value. This is any floating-point number with all 1's in its exponent field and a fraction field not equal to 0.

SNAN Signaling NAN. By convention this is a NAN with the bit after the exponent bit (bit 22 for single-precision, bit 51 for double-precision) set to 1.

QNAN Quiet NAN. By convention this is a NAN with the bit after the exponent bit (bit 22 for single-precision, bit 51 for double-precision) set to 0.

∞ machine infinity. This is a floating-point number with all 1's in its exponent field and a fraction field of 0. It is either $+\infty$ or $-\infty$, depending on the sign bit.

π 3.14159265358979323...

4.3.1 Acos

This section describes the double-precision and single-precision arccosine functions, `acos` and `facos`, that return in radians the inverse cosine of `x` in the range of 0 to π.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = facos(x);`	`double x, result;` `result = acos(x);`
FORTRAN	`real x, result` `result = acos(x)`	`double precision x, result` `result = acos(x)`

Pascal	`function facos(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := facos(x);`
		`function acos(n: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := acos(x);`

Modula-2	`VAR x, result: REAL;` `result := facos(x);`	`VAR x, result: LONGREAL;` `result := acos(x);`

ACCURACY

The `acos` and `facos` functions are accurate to within 3 *ulps*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\|x\| > 1$	QNAN with invalid signal
± 0.0	± π/2
+ 1.0	0.0
− 1.0	π

4.3.2 Acosh

This section describes the double-precision and single-precision arc-hyperbolic cosine functions, `acosh` and `facosh`, that return in radians the inverse hyperbolic cosine of argument `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = facosh(x);`	`double x, result;` `result = acosh(x);`
FORTRAN	`real x, result, acosh` `result = acosh(x)`	`double precision x, result, dacosh` `result = dacosh(x)`

Pascal	`function facosh(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := facosh(x);`
		`function acosh(n: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := acosh(x);`

Modula-2	`VAR x, result: REAL;` `result = facosh(x);`	`VAR x, result: LONGREAL;` `result = acosh(x);`

ACCURACY

The `acosh` and `facosh` functions inherit much of their error from `log1p` described in `exp`. The `acosh` and `facosh` functions are accurate to about 3 *ulps*.

DIAGNOSTICS

The `acosh` and `facosh` functions return a reserved operand if the argument is less than 1.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$x < 1$	QNAN with invalid signal
$+\infty$	$+\infty$
+ 1.0	+ 0.0

4.3.3 Asin

This section describes the double-precision and single-precision arc-sine functions, `asin` and `fasin`, that return in radians the arc-sine of argument `x` in the range of $-\pi/2$ to $\pi/2$.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fasin(x);`	`double x, result;` `result = asin(x);`
FORTRAN	`real x, result` `result = asin(x)`	`double precision x, result` `result = asin(x)`

Pascal	`function fasin(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := fasin(x);`
		`function asin(n: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := asin(x);`

Modula-2	`VAR x, result: REAL;` `result := fasin(x);`	`VAR x, result: LONGREAL;` `result := asin(x);`

ACCURACY

The `asin` and `fasin` functions are accurate to within 3 *ulps*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\|x\| > 1$	QNAN with invalid signal
± 1.0	$\pm \pi/2$
± 0.0	± 0.0

4.3.4 Asinh

This section describes the double-precision and single-precision functions, `asinh` and `fasinh`, that return in radians the arc-hyperbolic sine of argument `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fasinh(x);`	`double x, result;` `result = asinh(x);`
FORTRAN	`real x, result, asinh` `result = asinh(x)`	`double precision x, result, dasinh` `result = dasinh(x)`

Pascal	`function fasinh(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := fasinh(x);`
		`function asinh(n: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := asinh(x);`

Modula-2	`VAR x, result: REAL;` `result := fasinh(x);`	`VAR x, result: LONGREAL;` `result := asinh(x);`

ACCURACY

The `asinh` and `fasinh` functions inherit much of their error from `log1p` described in `exp`. The `asinh` and `fasinh` functions are accurate to about 3 *ulps*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$\pm\infty$
± 0.0	± 0.0

4.3.5 Atan

This section describes the double-precision and single-precision functions, `atan` and `fatan`, that return the arc tangent of `x` in the range of $-\pi/2$ to $\pi/2$.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fatan(x);`	`double x, result;` `result = atan(x);`
FORTRAN	`real x, result` `result = atan(x)`	`double precision x, result` `result = atan(x)`
Pascal	`var x, result: real;` `result := arctan(x);`	`var x, result: longreal;` `result := arctan(x);`
Modula-2	`VAR x, result: REAL;` `result := fatan(x);`	`VAR x, result: LONGREAL;` `result := atan(x);`

ACCURACY

The `atan` and `fatan` functions are accurate to within 1 *ulp*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$\pm\pi/2$
± 0.0	± 0.0

4.3.6 Atan2

This section describes the double-precision and single-precision functions, `atan2` and `fatan2`, that return the arc tangent of `y/x` in the range of $-\pi$ to π.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, y, result;` `result = fatan2(y, x);`	`double x, y, result;` `result = atan2(y, x);`
FORTRAN	`real x, y, result` `result = atan2(y, x);`	`double precision x, y, result` `result = atan2(y, x)`

Pascal	`function fatan2(n, m: real):`
	`    real; external;`
	`var y, x, result: real;`
	`result := fatan2(y, x);`
		`function atan2(n, m: longreal):`
		`    longreal; external;`
		`var y, x, result: longreal;`
		`result := atan2(y, x);`

Modula-2	`VAR x, y, result: REAL;` `result := fatan2(y, x);`	`VAR x, y, result: LONGREAL;` `result := atan2(y, x);`

ACCURACY

The `atan2` and `fatan2` functions are accurate to within 3 *ulps*.

SPECIAL CASES

y	x	result
(anything)	SNAN	QNAN with invalid signal
SNAN	(anything)	QNAN with invalid signal
(anything but SNAN)	QNAN	QNAN
QNAN	(anything but SNAN)	QNAN
± 0.0	+(anything but NAN)	± 0.0
± 0.0	−(anything but NAN)	$\pm \pi$
±(anything but 0 and NAN)	0	$\pm \pi/2$
±(anything but ∞ and NAN)	+∞	± 0
±(anything but ∞ and NAN)	−∞	$\pm \pi$
± ∞	+∞	$\pm \pi/4$
± ∞	−∞	$\pm 3\pi/4$
± ∞	(anything but 0, NAN, and ∞)	$\pm\pi/2$

4.3.7 Atanh

This section describes the double-precision and single-precision functions, `atanh` and `fatanh`, that compute the designated arc-hyperbolic tangent functions for real arguments.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fatanh(x);`	`double x, result;` `result = atanh(x);`
FORTRAN	`real x, result, atanh` `result = atanh(x)`	`double precision x, result, datanh` `result = datanh(x)`

Pascal	`function fatanh(n: real):`
	`    real; external;`
	`var x, result: real;`
	`result := fatanh(x);`
		`function atanh(n: longreal):`
		`    longreal; external;`
		`var x, result: longreal;`
		`result := atanh(x);`

Modula-2	`VAR x, result: REAL;` `result := fatanh(x);`	`VAR x, result: LONGREAL;` `result := atanh(x);`

ACCURACY

The `atanh` and `fatanh` functions inherit much of their errors from `log1p` described in `exp`. The `atanh` and `fatanh` functions are accurate to within 3 *ulps*.

DIAGNOSTICS

The `atanh` and `fatanh` functions return a reserved operand if the argument has an absolute value larger than or equal to 1.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\|x\| > 1$	QNAN with invalid signal
± 1.0	$\pm \infty$
± 0.0	± 0.0

4.3.8 Bessel

This section describes the bessel functions, `j0`, `j1`, `jn`, `y0`, `y1` and `yn`. These functions calculate bessel functions of the first and second kinds for real arguments and integer orders. Only double-precision functions are available.

CALLING SEQUENCES

COMPILER	*y* = j0, j1, y0 or y1	*y* = jn or yn

C	`double x, result;`
	`result = y(x);`
		`double x, result;`
		`int n;`
		`result = y(n, x);`

FORTRAN	`double precision x, result`
	`double precision y`
	`result = y(x)`
		`double precision x, result, y`
		`integer n`
		`result = y(n, x)`

Pascal	`function y(z: longreal):`
	`        longreal; external;`
	`var x, result: longreal;`
	`result := y(x);`
		`function y(f:integer; g:longreal):`
		`    longreal; external;`
		`var x, result: longreal;`
		`n: integer;`
		`result := y(n, x);`

Modula-2	`VAR x, result: LONGREAL;`
	`result := y(x);`
		`VAR x, result: LONGREAL;`
		`n: INTEGER;`
		`result := y(n, x);`

DIAGNOSTICS

Negative arguments cause the `y0`, `y1`, and `yn` functions to return an erroneous negative value and set `errno` to `EDOM`.

4.3.9 Cabs

This section describes the double-precision and single-precision complex absolute value functions, `cabs` and `fcabs`, that return $\sqrt{z.x^2+z.y^2}$. These functions return the correct result if $z.x^2$ or $z.y^2$ is out of range, as long as the result is within range.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION

C	`struct {`
	`   float x, y;`
	`} z;`
	`float result;`
	`result = fcabs(z);`
		`struct {`
		`   double x, y;`
		`} z;`
		`double result;`
		`result = cabs(z);`

FORTRAN	`complex z`
	`real result`
	`result = abs(z)`
		`double complex z`
		`double result`
		`result = abs(z)`

Pascal	`type complex = record`
	`  x,y : real;`
	`end;`
	`function fcabs(c: complex):`
	`   real; external;`
	`var z: complex;`
	`  result: real;`
	`result:= fcabs(z);`
		`type complex = record`
		`     x,y : longreal;`
		`end;`
		`function cabs(c: complex):`
		`   longreal; external;`
		`var z: complex;`
		`  result: longreal;`
		`result:= cabs(z);`

Modula-2	`TYPE complex = RECORD`
	`  x,y : REAL;`
	`END;`
	`FUNCTION fcabs(c: complex):`
	`   REAL; EXTERNAL;`
	`VAR z: complex;`
	`  r: REAL;`
	`r:= fcabs(z);`
		`TYPE complex = RECORD`
		`     x,y : LONGREAL;`
		`END;`
		`FUNCTION cabs(c: complex):`
		`   LONGREAL; EXTERNAL;`
		`VAR z: complex;`
		`  r: LONGREAL;`
		`r:= cabs(z);`

ACCURACY

The `cabs` and `fcabs` functions are accurate to within 1 *ulp*. In general, these functions return an integer whenever an integer might be expected.

SPECIAL CASES

x	y	result
$\pm\infty$	(anything)	$+\infty$
(anything)	$\pm\infty$	$+\infty$
SNAN	(anything but ∞)	QNAN with invalid signal
(anything but ∞)	SNAN	QNAN with invalid signal
QNAN	(anything but ∞ and SNAN	QNAN
(anything but ∞ and SNAN	QNAN	QNAN

4.3.10 Cbrt

This section describes the double-precision and single-precision cube root functions, `cbrt` and `fcbrt`, that return $\sqrt[3]{x}$.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fcbrt(x);`	`double x, result;` `result = cbrt(x);`
FORTRAN	`real x, result, cbrt` `result = cbrt(x)`	`double precision x, result, dcbrt` `result = dcbrt(x)`

Pascal	`function fcbrt(m: real):`
	`real; external;`
	`var x, result: real;`
	`result := fcbrt(x);`
		`function cbrt(m: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := cbrt(x);`

Modula-2	`VAR x, result: REAL;` `result := fcbrt(x);`	`VAR x, result: LONGREAL;` `result := cbrt(x);`

ACCURACY

The `cbrt` and `fcbrt` functions are accurate to within 1 *ulp*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$\pm\infty$
± 0.0	± 0.0

4.3.11 Ceil

This section describes the double-precision and single-precision ceiling functions, `ceil` and `fceil`, that return the smallest integer value not less than `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fceil(x);`	`double x, result;` `result = ceil(x);`
FORTRAN	`real x, result, ceil` `result = ceil(x)`	`double precision x, result, dceil` `result = dceil(x)`

Pascal	`function fceil(n: real):`
	`   real; external;`
	`var x, result: real;`
	`result := fceil(x);`
		`function ceil(n: longreal):`
		`   longreal; external;`
		`var x, result: longreal;`
		`result := ceil(x);`

Modula-2	`VAR x, result: REAL;` `result := fceil(x);`	`VAR x, result: LONGREAL;` `result := ceil(x);`

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$\pm\infty$
± 0.0	± 0.0

4.3.12 Compound

This section describes the functions, `compound` and `fcompound`, that return the compound interest factor, $(1+r)^n$.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION

C	`float r, result;`
	`int n;`
	`result = fcompound(r, n);`
		`double r, result;`
		`int n;`
		`result = compound(r, n);`

FORTRAN	`real r, result, compound`
	`integer n`
	`result = compound(r, n)`
		`double precision r, result, dcompound`
		`integer n`
		`result = dcompound(r, n)`

Pascal	`function fcompound(x: real;`
	`i: integer): real; external;`
	`var r, result: real;`
	`n: integer;`
	`result := fcompound(r, n);`
		`function compound(x: longreal;`
		`i: integer): longreal; external;`
		`var r, result: longreal;`
		`n: integer;`
		`result := compound(r, n);`

Modula-2	`VAR r, result: REAL;`
	`n: INTEGER;`
	`result := fcompound(r, n);`
		`VAR r, result: LONGREAL;`
		`n: INTEGER;`
		`result := compound(r, n);`

ACCURACY

The `compound` and `fcompound` functions are accurate to within 3 *ulps*.

SPECIAL CASES

r	n	result
(anything)	0	1
QNAN	anything	QNAN
SNAN	anything	QNAN with invalid signal
– 1.0	n < 0	+∞ with divide by zero signal
+∞	n > 0	+∞
+∞	n < 0	+0.0
–∞	n > 0, even	+∞ with divide by zero signal
–∞	n > 0, odd	–∞ with divide by zero signal
–∞	n < 0, even	+0.0
–∞	n < 0, odd	–0.0

4.3.13 Copysign

This section describes the double-precision and single-precision copy sign functions, `copysign` and `fcopysign`. The returned value for `copysign(x, y)` and `fcopysign(x, y)` has the magnitude of `x` and the sign of `y`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, y, result;` `result = fcopysign(x, y);`	`double x, y, result;` `result = copysign(x, y);`
FORTRAN	`real x, y, result` `result = sign(x, y)`	`double precision x, y, result` `result = sign(x, y)`

Pascal	`function fcopysign(m, n: real):`
	`real; external;`
	`var x, y, result: real;`
	`result := fcopysign(x, y);`
		`function copysign(m, n: longreal):`
		`longreal; external;`
		`var x, y, result: longreal;`
		`result := copysign(x, y);`

Modula-2	`VAR x, y, result: REAL;` `result := fcopysign(x, y);`	`VAR x, y, result: LONGREAL;` `result := copysign(x, y);`

4.3.14 Cos

This section describes the double-precision and single-precision cosine functions, `cos` and `fcos`, that return the trigonometric cosine function of a radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful. For very large arguments the result may have no significance.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fcos(x);`	`double x, result;` `result = cos(x);`
FORTRAN	`real x, result` `result = cos(x)`	`double precision x, result` `result = cos(x)`
Pascal	`var x, result: real;` `result := cos(x);`	`var x, result: longreal;` `result := cos(x);`
Modula-2	`VAR x, result: REAL;` `result := fcos(x);`	`VAR x, result: LONGREAL;` `result := cos(x);`

ACCURACY

The `cos` and `fcos` functions are accurate to within 1 *ulp*. The return value is never greater than 1.0.

DIAGNOSTICS

Extremely large arguments, such that $|x|+\frac{\pi}{2}>(2^{31}-1)\pi$, cause `fcos` to return value 0; `errno` is set to `EDOM`.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	QNAN with invalid signal
0.0	1.0

4.3.15 Cosh

This section describes the double-precision and single-precision hyperbolic cosine functions, `cosh` and `fcosh`, that return the hyperbolic cosine of argument `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fcosh(x);`	`double x, result;` `result = cosh(x);`
FORTRAN	`real x, result` `result = cosh(x)`	`double precision x, result` `result = cosh(x)`

Pascal	`function fcosh(n: real):`
	`    real; external;`
	`var x, result: real;`
	`result := fcosh(x);`
		`function cosh(n: longreal):`
		`    longreal; external;`
		`var x, result: longreal;`
		`result := cosh(x);`

Modula-2	`VAR x, result: REAL;` `result := fcosh(x);`	`VAR x, result: LONGREAL;` `result := cosh(x);`

ACCURACY

The `cosh` and `fcosh` functions are accurate to within 3 *ulps*.

DIAGNOSTICS

If the correct result overflows, an erroneous number of the appropriate sign returns and `errno` is set to `ERANGE`.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$+\infty$
± 0.0	$+ 1.0$

4.3.16 Drem

This section describes the double-precision and single-precision remainder functions, `drem` and `fdrem`, that return the remainder when `x` is divided by `y`. The returned value is $x - \left[\frac{m}{n}\right] * y$, where $\left[\frac{m}{n}\right]$ is the nearest integer less than $\frac{x}{y}$.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, y, result;` `result = fdrem(x, y);`	`double x, y, result;` `result = drem(x, y);`
FORTRAN	`real x, y, result, drem` `result = drem(x, y)`	`double precision x, y, result, ddrem` `result = ddrem(x, y)`

Pascal	`function fdrem(m, n:real):`
	`   real; external;`
	`var x, y, result: real;`
	`result := fdrem(x, y);`
		`function drem(m, n:longreal):`
		`   longreal; external;`
		`var x, y, result: longreal;`
		`result := drem(x, y);`

Modula-2	`VAR x, y, result: REAL;` `result := fdrem(x, y);`	`VAR x, y, result: LONGREAL;` `result := drem(x, y);`

SPECIAL CASES

x	y	result
SNAN	(anything)	QNAN with invalid signal
(anything)	SNAN	QNAN with invalid signal
∞	(anything but NAN)	QNAN with invalid signal
(anything but NAN)	0.0	QNAN with invalid signal
QNAN	(anything but SNAN)	QNAN
(anything but SNAN)	QNAN	QNAN

SEE ALSO

rem

4.3.17 Erf

This section describes the C program for floating-point error functions, `erf` and `ferf`. When called, `erf(x)` returns the error function of its argument; `erf(x)` is defined by $\frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt$.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = ferf(x);`	`double x, result;` `result = erf(x);`
FORTRAN	`real x, result, erf` `result = erf(x)`	`double precision x, result, derf` `result = derf(x)`

Pascal	`function ferf(m: real):`
	`   real; external;`
	`var x, result: real;`
	`result := ferf(x);`
		`function erf(m: longreal):`
		`   longreal; external;`
		`var x, result: longreal;`
		`result := erf(x);`

Modula-2	`VAR x, result: REAL;` `result := ferf(x);`	`VAR x, result: LONGREAL;` `result := erf(x);`

4.3.18 Exp

This section describes the double-precision and single-precision functions, `exp` and `fexp`, that return the natural exponential of `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fexp(x);`	`double x, result;` `result = exp(x);`
FORTRAN	`real x, result` `result = exp(x)`	`double precision x, result` `result = exp(x)`
Pascal	`var x, result: real;` `result := exp(x);`	`var x, result: longreal;` `result := exp(x);`
Modula-2	`VAR x, result: REAL;` `result := fexp(x);`	`VAR x, result: LONGREAL;` `result := exp(x);`

ACCURACY

The `exp` and `fexp` functions are accurate to within 1 *ulp*.

DIAGNOSTICS

If the correct result overflows, an erroneous number returns, `errno` is set to `ERANGE`, and 0.0 is returned.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$+\infty$	$+\infty$
$-\infty$	+0.0
± 0.0	1.0

4.3.19 Exp2

This section describes the double-precision and single-precision base 2 exponential functions, `exp2` and `fexp2`, that compute 2^x.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fexp2(x);`	`double x, result;` `result = exp2(x);`
FORTRAN	`real x, result, exp2` `result = exp2(x)`	`double precision x, result, dexp2` `result = dexp2(x)`

Pascal	`function fexp2(n: real):`
	`    real; external;`
	`var x, result: real;`
	`result := fexp2(x);`
		`function exp2(n: longreal):`
		`    longreal; external;`
		`var x, result: longreal;`
		`result := exp2(x);`

Modula-2	`VAR x, result: REAL;` `result := fexp2(x);`	`VAR x, result: LONGREAL;` `result := exp2(x);`

ACCURACY

The `exp2` and `fexp2` functions are accurate to within 3 *ulps* and `exp2(`*N*`)`, where *N* is an integer, is exact.

DIAGNOSTICS

If the correct result overflows, an erroneous number returns, `errno` is set to `ERANGE`, and 0.0 is returned.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
+∞	+∞
−∞	+0.0
±0.0	1.0

4.3.20 Expm1

This section describes the double-precision and single-precision functions, `expm1` and `fexpm1`, that return e^x-1. These functions exist so that the user can deal with numbers such as 1.00000000031597331, without losing significance when biasing by ±1.0.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fexpm1(x);`	`double x, result;` `result = expm1(x);`
FORTRAN	`real x, result, expm1` `result = expm1(x)`	`double precision x, result, dexpm1` `result = dexpm1(x)`

Pascal	`function fexpm1(n: real):`
	`    real; external;`
	`var x, result: real;`
	`result := fexpm1(x);`
		`function expm1(n: longreal):`
		`    longreal; external;`
		`var x, result: longreal;`
		`result := expm1(x);`

Modula-2	`VAR x, result: REAL;` `result := fexpm1(x);`	`VAR x, result: LONGREAL;` `result := expm1(x);`

ACCURACY

The `expm1` and `fexpm1` functions are accurate to within 3 *ulps*.

DIAGNOSTICS

If the correct result overflows, an erroneous number returns, `_errno` is set to `ERANGE`, and -1.0 is returned.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
+∞	+∞
−∞	−1.0
± 0.0	± 0.0

4.3.21 Fabs

This section describes the double-precision and single-precision functions, `fabs` and `ffabs`, that return the absolute value of `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = ffabs(x);`	`double x, result;` `result = fabs(x);`
FORTRAN	`real x, result` `result = abs(x)`	`double precision x, result` `result = abs(x)`
Pascal	`var x, result: real;` `result := abs(x);`	`var x, result: longreal;` `result := abs(x);`
Modula-2	`VAR x, result: REAL;` `result := ABS(x);`	`VAR x, result: LONGREAL;` `result := ABS(x);`

4.3.22 Finite

This section describes the double-precision and single-precision finite predicate functions, `finite` and `ffinite`, that are recommended by the IEEE standard 754 for floating-point arithmetic. These functions return a 1 if x is finite and a 0 if x is ±∞ or NAN.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION

C	`float x;`
	`int result;`
	`result = ffinite(x);`
		`double x;`
		`int result;`
		`result = finite(x);`

FORTRAN	`real x`
	`integer result, finite`
	`result = finite(x)`
		`double precision x`
		`integer result, dfinite`
		`result = dfinite(x)`

Pascal	`function ffinite(n: real):`
	`integer; external;`
	`var x: real;`
	`result: integer;`
	`result := ffinite(x);`
		`function finite(n: longreal):`
		`integer; external;`
		`var x: longreal;`
		`result: integer;`
		`result := finite(x);`

Modula-2	`VAR x: REAL;`
	`result: INTEGER;`
	`result := ffinite(x);`
		`VAR x: LONGREAL;`
		`result: INTEGER;`
		`result := finite(x);`

4.3.23 Floor

This section describes the double-precision and single-precision floor functions, `floor` and `ffloor`, that return the largest integral value no greater than `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = ffloor(x);`	`double x, result;` `result = floor(x);`
FORTRAN	`real x, result, floor` `result = floor(x)`	`double precision x, result, dfloor` `result = dfloor(x)`

Pascal	`function ffloor(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := ffloor(x);`
		`function floor(n: longreal):`
		`longreal ;external;`
		`var x, result: longreal;`
		`result := floor(x);`

Modula-2	`VAR x, result: REAL;` `result := ffloor(x);`	`VAR x, result: LONGREAL;` `result := floor(x);`

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$\pm\infty$
± 0.0	± 0.0

4.3.24 Fmod

This section describes the `fmod` and `ffmod` functions, that return the remainder of `x` on division by `y` with the same sign as `x`, except that if | `y` | << | `x` |, it returns 0.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, y, result;` `result = ffmod(x, y);`	`double x, y, result;` `result = fmod(x, y);`
FORTRAN	`real x, y, result` `result = mod(x, y)`	`double precision x, y, result` `result = mod(x, y)`

Pascal	`function ffmod(m, n: real):`
	`real; external;`
	`var x, y, result: real;`
	`result := ffmod(x, y);`
		`function fmod(m, n: longreal):`
		`longreal; external;`
		`var x, y, result: longreal;`
		`result := fmod(x, y);`

Modula-2	`VAR x, y, result: REAL;` `result := ffmod(x, y);`	`VAR x, y, result: LONGREAL;` `result := fmod(x, y);`

4.3.25 Fmodf

This section describes the single-precision function, `fmodf`, that returns the signed fractional part of `value` and stores the integer part indirectly through `whole`.

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`float x, whole, fract;` `fract := fmodf(x, &whole);`
FORTRAN	`real x, whole, fract, modf` `fract := modf(x, whole)`

Pascal	`function fmodf(value:real; var whole:real):`
	`real; external;`
	`var x, fract, whole:real;`
	`fract := fmodf(x, whole);`

Modula-2	`VAR x, fract, whole:REAL;` `fract := fmodf(x, whole);`

4.3.26 Fp_getexptn

This section describes the get floating-point exception status flag function, `fp_getexptn`, that returns a value indicating which floating-point exception status flags are set. The returned value is the OR'ed value representing the following exceptions:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int result;` `result = fp_getexptn();`
FORTRAN	`integer result, fp_getexptn` `result = fp_getexptn()`

Pascal	`function fp_getexptn:integer; external;`
	`var result:integer;`
	`result := fp_getexptn;`

Modula-2	`VAR result:INTEGER;` `result := fp_getexptn();`

SEE ALSO

fp_setexptn, *fp_tstexptn*

4.3.27 Fp_getround

This section describes the function, `fp_getround`, that returns the floating-point rounding direction from the rounding mode field of the Floating-point Status Register. The rounding direction is one of the following:

0 = TO_NEAREST
1 = TO_ZERO
2 = UPWARD
3 = DOWNWARD

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int result;` `result = fp_getround();`
FORTRAN	`integer result, fp_getround` `result = fp_getround()`

Pascal	`function fp_getround:integer; external;`
	`var result:integer;`
	`result := fp_getround;`

Modula-2	`VAR result:INTEGER;` `result := fp_getround();`

SEE ALSO

fp_setround

4.3.28 Fp_gettrap

This section describes the get floating-point trap enable flag function, `fp_gettrap`, that returns a value indicating which floating-point exception trap enable flags are set. The returned value is the OR'ed value representing the following exceptions:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int result;` `result = fp_gettrap();`
FORTRAN	`integer result, fp_gettrap` `result = fp_gettrap()`

Pascal	`function fp_gettrap:integer; external;`
	`var result:integer;`
	`result := fp_gettrap;`

Modula-2	`VAR result:INTEGER;` `result := fp_gettrap();`

SEE ALSO

fp_settrap, *fp_testtrap*

4.3.29 Fp_gmathenv

This section describes the function, `fp_gmathenv`, that gets the math environment from the Floating-point Status Register (FSR). Input argument `e` is a structure to receive the information from the FSR.

The math environment is defined as a record with the following three fields:

first field: rounding mode (`rm`)

0 TO_NEAREST
1 TO_ZERO
2 UPWARD
3 DOWNWARD

second field: trap enable flags (`tenable`)

third field: exception status flags (`estatus`)

`Tenable` and `estatus` may be any OR'ed value representing the following exceptions in the FSR:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

CALLING SEQUENCES

<table>
<tr><th>LANGUAGE</th><th>CALLING SEQUENCE</th></tr>
<tr><td>C</td><td><pre>struct environment {
  int rm, tenable, estatus;
} e;
void fp_gmathenv();
fp_gmathenv(&e);</pre></td></tr>
<tr><td>FORTRAN</td><td><pre>integer e(3)
integer rm, tenable, estatus

equivalence (e(1),rm), (e(2),tenable), (e(3),estatus)
call fp_gmathenv(e)</pre></td></tr>
<tr><td>Pascal</td><td><pre>type fp_status_rec=record
  rm:integer;
  tenable:integer;
  estate:integer;
end;
procedure fp_gmathenv(var fps:fp_status_rec);external;
var e:fp_status_rec;
fp_gmathenv(e);</pre></td></tr>
<tr><td>Modula-2</td><td><pre>TYPE fp_status=RECORD
  rm:INTEGER;
  tenable:INTEGER;
  estate:INTEGER;
END;
VAR e:fp_status;
fp_gmathenv(e);</pre></td></tr>
</table>

SEE ALSO

fp_smathenv

4.3.30 Fpgtrpvctr

This section describes the function, `fpgtrpvctr`, that returns the address of the FPU trap handler function. This function is for execution on a development board only.

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`void (*v) ();` `v = fpgtrpvctr();`
FORTRAN	Cannot get user defined trap function in FORTRAN
Pascal	Cannot get user defined trap function in Pascal

Modula-2	`FROM libm IMPORT procedure_ptr, fpgtrpvctr;`
	`VAR trap_handler:procedure_ptr;`
	`trap_handler := fpgtrpvctr();`

SEE ALSO

fpstrpvctr

4.3.31 Fp_procentry

This section describes the process entry (save FSR and install IEEE defaults) function, `fp_procentry`, that returns the current floating-point math environment. This is the contents of the Floating-point Status Register (FSR).

The `fp_procentry` function clears the exception status fields in the FSR and sets the FSR control fields to IEEE defaults. (The Rounding mode is set to "TO_NEAREST." All traps are disabled.)

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int result;` `result = fp_procentry();`
FORTRAN	`integer result, fp_procentry` `result = fp_procentry()`

Pascal	`function fp_procentry:integer; external;`
	`var result:integer;`
	`result := fp_procentry;`

Modula-2	`VAR result:INTEGER;` `result := fp_procentry();`

SEE ALSO

fp_procexit

4.3.32 Fp_procexit

This section describes the process exit (restore FSR and signal exceptions) function, `fp_procexit`, that replaces the current FSR contents with an `e`. (An `e` will generally be the output of an earlier fp_procentry call.) It then signals any exceptions (underflow or inexact) that are flagged in the stored FSR.

(Surround atomic code with `fp_procentry` and `fp_procexit` calls. `Fp_procentry` installs IEEE default modes; `fp_procexit` reinstates old modes and then signals any exception which occurred since the last `fp_procentry` call.)

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int e;` `fp_procexit(e);`
FORTRAN	`integer e` `call fp_procexit(e)`

Pascal	`procedure fp_procexit(x:integer); external;`
	`var e:integer;`
	`fp_procexit(e);`

Modula-2	`VAR e:INTEGER;` `fp_procexit(e);`

SEE ALSO

fp_procentry

4.3.33 Fp_setexptn

This section describes the function, `fp_setexptn`, that sets or clears exception status flags in the Floating-point Status Register. An `e` may be any OR'ed value representing the following exceptions:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

If `s` is 0, the FSR status flags corresponding to the exceptions encoded in `e` are cleared.

If `s` is not 0, the FSR status flags for the exceptions encoded in `e` are set.

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int e, s;` `fp_setexptn(e, s);`
FORTRAN	`integer e, s` `call fp_setexptn(e, s)`

Pascal	`procedure fp_setexptn(x, y:integer); external;`
	`var e, s:integer;`
	`fp_setexptn(e, s);`

Modula-2	`VAR e, s:INTEGER;` `fp_setexptn(e, s);`

SEE ALSO

fp_getexptn, *fp_tstexptn*

4.3.34 Fp_setround

This section describes the function, `fp_setround`, that sets the floating-point rounding mode in the Floating-point Status Register to `r`, where `r` is one of the following rounding directions:

0 = TO_NEAREST
1 = TO_ZERO
2 = UPWARD
3 = DOWNWARD

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int r;` `fp_setround(r);`
FORTRAN	`integer r` `call fp_setround(r)`

Pascal	`procedure fp_setround(x:integer); external;`
	`var r:integer;`
	`fp_setround(r);`

Modula-2	`VAR r:INTEGER;` `fp_setround(r);`

SEE ALSO

fp_getround

4.3.35 Fp_settrap

This section describes the function, `fp_settrap`, that sets or clears trap enable flags in the Floating-point Status Register. An `e` may be any OR'ed value representing the following traps:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

If `s` is 0, the FSR trap enable flags corresponding to the exceptions encoded in `e` are cleared.

If `s` is not 0, the FSR trap enable flags for the exceptions encoded in `e` are set.

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int e, s;` `fp_settrap(e,s);`
FORTRAN	`integer e, s` `call fp_settrap(e,s)`

Pascal	`procedure fp_settrap(e,s:integer); external;`
	`var e,s:integer;`
	`fp_settrap(e,s);`

Modula-2	`VAR e,s:INTEGER;` `fp_settrap(e,s);`

SEE ALSO

fp_gettrap, *fp_testtrap*

4.3.36 Fp_smathenv

This section describes the function, `fp_smathenv`, that sets the math environment in the Floating-point Status Register (FSR). Input argument `e` is a pointer to a structure containing the information to be written to the FSR.

The math environment is defined as a record with the following three fields:

first field: rounding mode (`rm`)

0 TO_NEAREST
1 TO_ZERO
2 UPWARD
3 DOWNWARD

second field: trap enable flags (`tenable`)

third field: exception status flags (`status`)

`Tenable` and `estatus` may be any OR'ed value representing the following exceptions:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE

C	`struct environment {`
	`  int rm, tenable, estatus;`
	` } e;`
	`void fp_smathenv();`
	`fp_smathenv(&e);`

FORTRAN	`integer e(3)`
	`integer rm, tenable, estatus`
	
	`equivalence (e(1),rm), (e(2),tenable), (e(3),estatus)`
	`call fp_smathenv(e)`

Pascal	`type fp_status_rec=record`
	`  rm:integer;`
	`  tenable:integer;`
	`  estate:integer;`
	`end;`
	`procedure fp_smathenv(var fps:fp_status_rec); external;`
	`var e:fp_status_rec;`
	`fp_smathenv(e);`

Modula-2	`TYPE fp_status=RECORD`
	`  rm:INTEGER;`
	`  tenable:INTEGER;`
	`  estate:INTEGER;`
	`END;`
	`VAR e:fp_status;`
	`fp_smathenv(e);`

SEE ALSO

fp_gmathenv

4.3.37 Fpstrpvctr

This section describes the function, `fpstrpvctr`, that replaces the FPU trap handler function with function `v`. The `v` function is called when any FPU trap occurs. This function is for execution on a development board only. This routine assumes the caller is in user mode and reinstates user mode before returning.

CALLING SEQUENCES

<table>
<tr><th>LANGUAGE</th><th>CALLING SEQUENCE</th></tr>
<tr><td>C</td><td><pre>void v();
fpstrpvctr(v);</pre></td></tr>
<tr><td>FORTRAN</td><td><pre> external v
call fpstrpvctr(v);</pre></td></tr>
<tr><td>Pascal</td><td><pre>procedure fpstrpvctr(procedure p); external;
procedure new_handler;
begin
   .
   .
   .
end;
fpstrpvctr(new_handler);</pre></td></tr>
<tr><td>Modula-2</td><td><pre>FROM libm IMPORT procedure_ptr, fpstrpvctr;
PROCEDURE new_handler;
BEGIN
   .
   .
   .
END new_handler;
fpstrpvctr(ADDR(new_handler));</pre></td></tr>
<tr><td colspan="2">NOTE: Alternatively, v may be declared "void (*v) ()" and set to the address of the handler function (e.g., v = fpgtrpvctr()), before the call to fpstrpvctr.</td></tr>
</table>

SEE ALSO

fpgtrpvctr

4.3.38 Fp_testtrap

This section describes the test floating-point trap enable flag function, `fp_testtrap`, that tests the enable flags in the Floating-point Status Register corresponding to the traps encoded in `e`. An `e` may be any OR'ed value representing the following exceptions:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

If the FSR trap enable flag for any of the exceptions encoded in `e` is set, 1 is returned, otherwise 0 is returned.

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int e, result;` `result = fp_testtrap(e);`
FORTRAN	`integer e, result, fp_testtrap` `result = fp_testtrap(e)`

Pascal	`function fp_testtrap(e:integer):integer; external;`
	`var x, result:integer;`
	`result:=fp_testtrap(x);`

Modula-2	`VAR result:INTEGER;`
	`e:INTEGER;`
	`result := fp_testtrap(e);`

SEE ALSO

fp_gettrap, *fp_settrap*

4.3.39 Fp_tstexptn

This section describes the test floating-point exception status flag function, `fp_tstexptn`, that tests the status flags in the Floating-point Status Register corresponding to the exceptions encoded in `e`. An `e` may be any OR'ed value representing the following exceptions:

1 = UNDERFLOW
2 = INEXACT
4 = INVALID
8 = DIVIDE_BY_ZERO
16 = OVERFLOW

If the FSR status flag for any of the exceptions encoded in `e` is set, 1 is returned, otherwise 0 is returned.

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`int e, result;` `result = fp_tstexptn(e);`
FORTRAN	`integer e, result, fp_tstexptn` `result = fp_tstexptn(e)`

Pascal	`function fp_tstexptn(e:integer):integer; external;`
	`var e, result:integer;`
	`result:=fp_tstexptn(e);`

Modula-2	`VAR e, result:INTEGER;` `result := fp_tstexptn(e);`

SEE ALSO

fp_getexptn, *fp_setexptn*

4.3.40 Gamma

This section describes the double-precision `gamma` function (ln gamma function). `Gamma(x)` returns ln $|\Gamma(|x|)|$. The sign of $\Gamma(|x|)$ is returned in the external integer `signgam`. The following C program might be used to calculate Γ:

```
y = gamma(x);
if (y > 88.0) error();
y = exp(y);
if (signgam) y = -y;
```

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE
C	`double x, result;` `result = gamma(x);`

FORTRAN	`common/signgam/signgam`
	`double precision x, result, gamma`
	`result = gamma(x)`

Pascal	`#include "signgam.h"`
	`function gamma(x: longreal): longreal;`
	`external;`
	`var m, result: longreal;`
	`result := gamma(m);`

Modula-2	`VAR x, result: LONGREAL;` `result := gamma(x);`

NOTE: In FORTRAN, to use `signgam`, the user must declare it in a named common block in the file, *e.g.*, `/signgam/signgam`.

In Pascal, the user must declare `signgam` as an external integer variable by placing its declaration in an "include" file and referencing the file in the source file, *e.g.*, `#include "signgam.h"`.

DIAGNOSTICS

A large value is returned for negative integer arguments.

4.3.41 Hypot

This section describes the double-precision and single-precision euclidean distance functions, `hypot` and `fhypot`, that return $\sqrt{x^2+y^2}$. These functions return the correct result if x^2 or y^2 is out of range, as long as the results are within range.

ACCURACY

The `hypot` and `fhypot` functions are accurate to within 1 *ulp*.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, y, result;` `result = fhypot(x, y);`	`double x, y, result;` `result = hypot(x, y);`

FORTRAN	`real x, y, result, hypot`
	`result = hypot(x, y)`
		`double precision x, y, result`
		`double precision dhypot`
		`result = dhypot(x, y)`

Pascal	`function fhypot(m, n: real):`
	`real; external;`
	`var x, y, result: real;`
	`result := fhypot(x, y);`
		`function hypot(m, n: longreal):`
		`longreal; external;`
		`var x, y, result: longreal;`
		`result := hypot(x, y);`

Modula-2	`VAR x, y, result: REAL;` `result := fhypot(x, y);`	`VAR x, y, result: LONGREAL;` `result := hypot(x, y);`

SPECIAL CASES

x	y	result
± ∞	(anything)	+ ∞
(anything)	± ∞	+ ∞
SNAN	(anything but ∞)	QNAN with invalid signal
(anything but ∞)	SNAN	QNAN with invalid signal
QNAN	(anything but ∞ and SNAN)	QNAN
(anything but ∞ and SNAN)	QNAN	QNAN

4.3.42 Inf

This section describes the double-precision and single-precision functions, `inf` and `finf`, that return machine infinity.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float result;` `result = finf();`	`double result;` `result = inf();`
FORTRAN	`real result, inf` `result = inf()`	`double precision result, dinf` `result = dinf()`

Pascal	`function finf: real;`
	`    external;`
	`var result: real;`
	`result := finf;`
		`function inf: longreal;`
		`    external;`
		`var result: longreal;`
		`result := inf;`

Modula-2	`VAR result: REAL;` `result := finf;`	`VAR result: LONGREAL;` `result := inf;`

4.3.43 Log

This section describes the double-precision and single-precision functions, `log` and `flog`, that return the natural logarithm of `x`.

ACCURACY

The `log` and `flog` functions are accurate to within 1 *ulp*.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = flog(x);`	`double x, result;` `result = log(x);`
FORTRAN	`real x, result` `result = log(x)`	`double precision x, result` `result = log(x)`
Pascal	`var x, result: real;` `result := ln(x);`	`var x, result: longreal;` `result := ln(x);`
Modula-2	`VAR x, result: REAL;` `result := flog(x);`	`VAR x, result: LONGREAL;` `result := log(x);`

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$x < 0.0$	QNAN with invalid signal
± 0.0	$-\infty$ with divide by zero signal

4.3.44 Log10

This section describes the double-precision and single-precision common logarithm functions, `log10` and `flog10`, that return the base 10 logarithm of `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = flog10(x);`	`double x, result;` `result = log10(x);`
FORTRAN	`real x, result` `result = log10(x)`	`double precision x, result` `result = log10(x)`
Pascal	`var x, result: real;` `result := log10(x);`	`var x, result: longreal;` `result := log10(x);`
Modula-2	`VAR x, result: REAL;` `result := flog10(x);`	`VAR x, result: LONGREAL;` `result := log10(x);`

ACCURACY

The `log10` and `flog10` functions are accurate to within 3 *ulp* and $\log_{10}(10^{+N})$ should be equal to +N.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$x < 0.0$	QNAN with invalid signal
± 0.0	$-\infty$ with divide by zero signal

4.3.45 Loglp

This section describes the double-precision and single-precision functions, `log1p` and `flog1p`, that return the natural logarithm of (1 + x). These functions exist so that the user can deal with numbers, such as 1.00000000031597331, without losing significance when biasing by ±1.0.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = flog1p(x);`	`double x, result;` `result = log1p(x);`
FORTRAN	`real x, result, log1p` `result = log1p(x)`	`double precision x, result, dlog1p` `result = dlog1p(x)`

Pascal	`function flog1p(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := flog1p(x);`
		`function log1p(n: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := log1p(x);`

Modula-2	`VAR x, result: REAL;` `result := flog1p(x);`	`VAR x, result: LONGREAL;` `result := log1p(x);`

ACCURACY

The `log1p` and `flog1p` functions are accurate to within 3 *ulps*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
x < −1.0	QNAN with invalid signal
−1.0	−∞ with divide by zero signal
+∞	+∞
±0.0	±0.0

4.3.46 Log2

This section describes the double-precision and single-precision functions, `log2` and `flog2`, that return the base 2 logarithm of `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = flog2(x);`	`double x, result;` `result = log2(x);`
FORTRAN	`real x, result, log2` `result = log2(x)`	`double precision x, result, dlog2` `result = dlog2(x)`

Pascal	`function flog2(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := flog2(x);`
		`function log2(n: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := log2(x);`

Modula-2	`VAR x, result: REAL;` `result := flog2(x);`	`VAR x, result: LONGREAL;` `result := log2(x);`

ACCURACY

The `log2` and `flog2` functions are accurate within 3 *ulps*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
x < 0.0	QNAN with invalid signal
± 0.0	$-\infty$ with divide by zero signal

4.3.47 Neg

This section describes the double-precision and single-precision negation functions, `neg` and `fneg`, that return the negative of `x`. This is identical to `-x`, but is guaranteed not to generate a machine exception in case of special operands.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fneg(x);`	`double x, result;` `result = neg(x);`
FORTRAN	`real x, result, neg` `result = neg(x)`	`double precision x, result, dneg` `result = dneg(x)`

Pascal	`function fneg(n: real):`
	`real; external;`
	`var x, result: real;`
	`result := fneg(x);`
		`function neg(n: longreal):`
		`longreal; external;`
		`var x, result: longreal;`
		`result := neg(x);`

Modula-2	`VAR x, result: REAL;` `result := fneg(x);`	`VAR x, result: LONGREAL;` `result := neg(x);`

4.3.48 Nextfloat

This section describes the double-precision neighbor function, `nextdouble`, and the single-precision neighbor function, `nextfloat`, that returns the nearest representable neighbor of `x` in the direction of `y`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, y, result;` `result = nextfloat(x, y);`	`double x, y, result;` `result = nextdouble(x, y);`

FORTRAN	`real x, y, result, nextfloat`
	`result = nextfloat(x, y)`
		`double precision x, y, result`
		`double precision nextdouble`
		`result = nextdouble(x, y)`

Pascal	`function nextfloat(m, n: real):`
	`real; external;`
	`var x, y, result: real;`
	`result := nextfloat(x, y);`
		`function nextdouble(m, n: longreal):`
		`longreal; external;`
		`var x, y, result: longreal;`
		`result := nextdouble(x, y);`

Modula-2	`VAR x, y, result: REAL;` `result := nextfloat(x, y);`	`VAR x, y, result: LONGREAL;` `result := nextdouble(x, y);`

SPECIAL CASES

x	y	result
SNAN	anything	QNAN with invalid signal
QNAN	anything but SNAN	QNAN
anything	SNAN	QNAN with invalid signal
anything but SNAN	QNAN	QNAN
x	x	x

4.3.49 Pi

This section describes the double-precision and single-precision pi functions, `pi` and `fpi`, that return machine π.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float result;` `result = fpi();`	`double result;` `result = pi();`
FORTRAN	`real result, pi` `result = pi()`	`double precision result, dpi` `result = dpi()`

Pascal	`function fpi: real;`
	`external;`
	`var result: real;`
	`result := fpi;`
		`function pi: longreal;`
		`external;`
		`var result: longreal;`
		`result := pi;`

Modula-2	`VAR result: REAL;` `result := fpi;`	`VAR result: LONGREAL;` `result := pi;`

4.3.50 Pow

This section describes the double-precision and single-precision power functions, `pow` and `fpow`, that return x^y.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, y, result;` `result = fpow(x, y);`	`double x, y, result;` `result = pow(x, y);`
FORTRAN	use the power operator (`**`)	use the power operator (`**`)
Pascal	`var x, y, result: real;` `result := power(x, y);`	`var x, y, result: longreal;` `result := power(x, y);`
Modula-2	`VAR x, y, result: REAL;` `result := fpow(x, y);`	`VAR x, y, result: LONGREAL;` `result := pow(x, y);`

ACCURACY

The `pow` and `fpow` functions are accurate to within 3 *ulps*.

DIAGNOSTICS

If the correct result overflows, an erroneous number of appropriate sign is returned and `errno` is set to `ERANGE`.

SPECIAL CASES

x	y	result
(anything)	0	1
(anything but SNAN)	1	x
SNAN	1	QNAN with invalid signal
(anything)	SNAN	QNAN with invalid signal
(anything but SNAN)	QNAN	QNAN
SNAN	(anything but 0)	QNAN with invalid signal
QNAN	(anything but 0 and SNAN)	QNAN
$\|x\| > 1$	$+\infty$	$+\infty$
$\|x\| > 1$	$-\infty$	+0
$\|x\| < 1$	$+\infty$	+0
$\|x\| < 1$	$-\infty$	$+\infty$
±1	$\pm\infty$	QNAN with invalid signal
+0	+(anything but 0 and NAN)	+0
−0	+(anything but 0, NAN, odd integer)	+0
+0	−(anything but 0 and NAN)	$+\infty$ with divide by zero signal
−0	+(anything but 0, NAN, odd integer)	$+\infty$ with divide by zero signal
−0	(odd integer)	−(+0 ** (odd integer))
$+\infty$	+(anything but 0 and NAN)	$+\infty$
$+\infty$	−(anything but 0 and NAN)	+0
$-\infty$	(anything)	−0 ** (−anything)
−(anything)	(integer)	(−1)**(integer)* (+anything**integer)
−(anything except 0)	(non-integer)	QNAN with invalid signal

4.3.51 Randomx

This section discusses the random number generators, `randomx` and `initrand`.

The `randomx` function generates a sequence of pseudo-random numbers, uniformly distributed over the interval, 0 to 2^{31} -1. One number in the sequence is generated at each call to `randomx`.

The `initrand` function initializes `randomx` with a user-supplied seed. The default seed (if `randomx` is called without first calling `initrand`) is 19414. To re-initialize `randomx` and produce a different sequence of random numbers, call initrand with a new seed.

The `randomx` function uses an algorithm derived from the Data Encryption Standard developed by the National Security Agency. The generated values repeat with a cycle length of approximately 16 * 2^{31}. This cycle length is suitable for most purposes. Moreover, all bits of the generated values are random, uniformly distributed, and uncorrelated.

CALLING SEQUENCES

LANGUAGE	CALLING SEQUENCE

C	`int s, result;`
	`initrand(s);`
	`result = randomx();`

FORTRAN	`integer s, result, randomx, initrand`
	`call initrand(s)`
	`result = randomx()`

Pascal	`procedure initrand(seed: integer); external;`
	`function randomx: integer; external;`
	`var s, result: integer;`
	`initrand(s);`
	`result := randomx;`

Modula-2	`VAR s, result: INTEGER;`
	`initrand(s);`
	`result := randomx();`

4.3.52 Relation

This section describes the double-precision and single-precision relation functions, `relation` and `frelation`, that return the relationship between numbers `x` and `y`. The returned value is one of the following:

LT	=0	if $x < y$,
EQ	=1	if $x = y$,
GT	=2	if $x > y$,
UNORDERED	=3	if either x or y is NAN

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION

C	`float x, y;`
	`int result;`
	`result = frelation(x, y);`
		`double x, y;`
		`int result;`
		`result = relation(x, y);`

FORTRAN	`real x, y`
	`integer result, relation`
	`result = relation(x, y)`
		`double precision x, y`
		`integer result, drelation`
		`result = drelation(x, y)`

Pascal	`function frelation(m, n: real):`
	`integer; external;`
	`var x, y: real;`
	`result: integer;`
	`result := frelation(x, y);`
		`function relation(m, n: longreal):`
		`integer; external;`
		`var x, y: longreal;`
		`result: integer;`
		`result := relation(x, y);`

Modula-2	`VAR x, y: REAL;`
	`result: INTEGER;`
	`result := frelation(x, y);`
		`VAR x, y: LONGREAL;`
		`result: INTEGER;`
		`result := relation(x, y);`

SPECIAL CASES

x	y	result
NAN	anything	UNORDERED
anything	NAN	UNORDERED

4.3.53 Rem

This section describes the double-precision and single-precision remainder functions, `rem` and `frem`.

The `rem` and `frem` functions return the remainder when `x` is divided by `y`. The returned value is $x - \left[\frac{m}{n}\right] * y$, where $\left[\frac{m}{n}\right]$ is the nearest integer less than $\frac{x}{y}$. The `rem` and `frem` functions also return as `quo`, the low-order 7 bits of the integer part of the quotient.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION

C	`float  x, y, result;`
	`int quo;`
	`result = frem(x, y, &quo);`
		`double x, y, result;`
		`int quo;`
		`result = rem(x, y, &quo);`

FORTRAN	`real x, y, result, remainder`
	`integer quo`
	`result = remainder(x, y, quo)`
		`double precision x, y, result`
		`double precision dremainder`
		`integer quo`
		`result = dremainder(x, y, quo)`

Pascal	`function frem(m, n: real;`
	`  var q: integer): real;`
	`  external;`
	`var x, y, result: real;`
	`  quo: integer;`
	`result := frem(x, y, quo);`
		`function rem(m, n: longreal;`
		`  var q: integer): longreal;`
		`  external;`
		`var x, y, result: longreal;`
		`  quo: integer;`
		`result := rem(x, y, quo);`

Modula-2	`VAR x, y, result: REAL;`
	`  quo: INTEGER;`
	`result := frem(x, y, quo);`
		`VAR x, y, result: LONGREAL;`
		`  quo: INTEGER;`
		`result := rem(x, y, quo);`

SPECIAL CASES

x	y	result
SNAN	(anything)	QNAN with invalid signal
(anything)	SNAN	QNAN with invalid signal
∞	(anything but NAN)	QNAN with invalid signal
(anything but NAN)	0.0	QNAN with invalid signal
QNAN	(anything but SNAN)	QNAN
(anything but SNAN)	QNAN	QNAN

NOTE: Quo is undefined in the exception cases.

SEE ALSO

drem

4.3.54 Rint

This section describes the double-precision and single-precision integral value functions, `rint` and `frint`.

The `rint` and `frint` functions convert `x` to the nearest integral value in the direction indicated by the current rounding mode.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = frint(x);`	`double x, result;` `result = rint(x);`
FORTRAN	`real x, result, rint` `result = rint(x)`	`double precision x, result, drint` `result = drint(x)`

Pascal	`function frint(n: real):`
	`   real; external;`
	`var x, result: real;`
	`result := frint(x);`
		`function rint(n: longreal):`
		`   longreal; external;`
		`var x, result: longreal;`
		`result := rint(x);`

Modula-2	`VAR x, result: REAL;` `result := frint(x);`	`VAR x, result: LONGREAL;` `result := rint(x);`

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$\pm\infty$
± 0.0	± 0.0

4.3.55 Sin

This section describes the double-precision and single-precision sine functions, `sin` and `fsin`.

The `sin` and `fsin` functions return the trigonometric sine function of a radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful. For very large arguments, the result may have no significance.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fsin(x);`	`double x, result;` `result = sin(x);`
FORTRAN	`real x, result` `result = sin(x)`	`double precision x, result` `result = sin(x)`
Pascal	`var x, result: real;` `result := sin(x);`	`var x, result: longreal;` `result := sin(x);`
Modula-2	`VAR x, result: REAL;` `result := fsin(x);`	`VAR x, result: LONGREAL;` `result := sin(x);`

ACCURACY

The `sin` and `fsin` functions are accurate to within 1 *ulp*. The return value is never greater than 1.0.

DIAGNOSTICS

Extremely large arguments, such that abs(x) is greater than $(2^{31}-1)*\pi$, cause `fsin` to return value 0; `errno` is set to `EDOM`.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	QNAN with invalid signal
± 0.0	± 0.0

4.3.56 Sinh

This section describes the double-precision and single-precision functions, `sinh` and `fsinh`, that return the hyperbolic sine of argument `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fsinh(x);`	`double x, result;` `result = sinh(x);`
FORTRAN	`real x, result` `result = sinh(x)`	`double precision x, result` `result = sinh(x)`

Pascal	`function fsinh(n: real):`
	`    real; external;`
	`var x, result: real;`
	`result := fsinh(x);`
		`function sinh(n: longreal):`
		`    longreal; external;`
		`var x, result: longreal;`
		`result := sinh(x);`

Modula-2	`VAR x, result: REAL;` `result := fsinh(x);`	`VAR x, result: LONGREAL;` `result := sinh(x);`

ACCURACY

The `sinh` and `fsinh` functions are accurate to within 3 *ulps*.

DIAGNOSTICS

If the correct result overflows, an erroneous number of the appropriate sign returns and `errno` sets to `ERANGE`.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	$\pm\infty$
± 0.0	± 0.0

4.3.57 Sqrt

This section describes the double-precision and single-precision square root functions, `sqrt` and `fsqrt`, that return $\sqrt{x}$.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = fsqrt(x);`	`double x, result;` `result = sqrt(x);`
FORTRAN	`real x, result` `result = sqrt(x)`	`double precision x, result` `result = sqrt(x)`
Pascal	`var x, result: real;` `result := sqrt(x);`	`var x, result: longreal;` `result := sqrt(x);`
Modula-2	`VAR x, result: REAL;` `result := fsqrt(x);`	`VAR x, result: LONGREAL;` `result := sqrt(x);`

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$x < 0$	QNAN with invalid signal
$+\infty$	$+\infty$
± 0.0	± 0.0

4.3.58 Tan

This section describes the double-precision and single-precision tangent functions, `tan` and `ftan`, that return the trigonometric tangent function of a radian argument. The magnitude of the argument should be checked by the caller to make sure the result is meaningful. For very large arguments the result may have no significance.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = ftan(x);`	`double x, result;` `result = tan(x);`
FORTRAN	`real x, result` `result = tan(x)`	`double precision x, result` `result = tan(x)`
Pascal	`var x, result: real;` `result := tan(x);`	`var x, result: longreal;` `result := tan(x);`
Modula-2	`VAR x, result: REAL;` `result := ftan(x);`	`VAR x, result: LONGREAL;` `result := tan(x);`

ACCURACY

The `tan` and `ftan` functions are accurate to within 2 *ulps*.

DIAGNOSTICS

At tangent's singular points $x = n * \frac{\pi}{2}$ for n odd, `errno` to `ERANGE`.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	QNAN with invalid signal
± 0.0	± 0.0

4.3.59 Tanh

This section describes the double-precision and single-precision functions, `tanh` and `ftanh`, that return the hyperbolic tangent of argument `x`.

CALLING SEQUENCES

LANGUAGE	SINGLE-PRECISION	DOUBLE-PRECISION
C	`float x, result;` `result = ftanh(x);`	`double x, result;` `result = tanh(x);`
FORTRAN	`real x, result` `result = tanh(x)`	`double precision x, result` `result = tanh(x)`

Pascal	`function ftanh(n: real):`
	`    real; external;`
	`var x, result: real;`
	`result := ftanh(x);`
		`function tanh(n: longreal):`
		`    longreal; external;`
		`var x, result: longreal;`
		`result := tanh(x);`

Modula-2	`VAR x, result: REAL;` `result := ftanh(x);`	`VAR x, result: LONGREAL;` `result := tanh(x);`

ACCURACY

The `tanh` and `ftanh` functions are accurate to within 3 *ulps*.

SPECIAL CASES

x	result
QNAN	QNAN
SNAN	QNAN with invalid signal
$\pm\infty$	± 1.0
± 0.0	± 0.0

Chapter 5

FPEE LIBRARY

5.1 INTRODUCTION

When a Floating-Point Unit (FPU) is not present, the Floating-Point Enhancement and Emulation (FPEE) library provides low-cost floating-point support by emulating the *Series 32000* FPU instructions. When an FPU is present, FPEE enhances the *Series 32000* FPU by providing additional functionality as recommended by Draft 10 of the ANSI/IEEE Task 754 Proposal for Binary Floating-point Arithmetic (IEEE 754). FPEE meets the IEEE 754 standard for double-precision arithmetic.

To maximize the efficiency of execution of external procedures and function calls, we have adopted the convention of passing only double-precision floating-point arguments and results. Because of this, when a single-precision procedure or function is called, a hardware instruction is invoked whenever it is necessary to convert an argument from single-precision to double-precision. If this instruction is executed with a reserved operand, the result is an immediate invalid-operation trap. It is not possible for the user to disable this trap; therefore, with the combination of the math library and the floating-point emulation library, the user may achieve compliance only with the IEEE 754 Standard for Floating-Point Arithmetic for double-precision arithmetic.

Major problems result when the user is unable to effectively use the `frelation` function; `frelation` returns "unordered" when passed a quiet NAN as an argument, and `ffinite` returns a zero when passed an infinity or a NAN as an argument. These routines, if the source code is available, can be included in a program as local routines to avoid the conversion problem.

The FPEE library is provided in source form and as a binary library suitable for its particular GNX tool-set environment. The source includes a makefile to build the FPEE library. The FPEE library can be configured to enhance/emulate either the NS32081 FPU or the NS32381 FPU for either native applications or cross-development applications.

This chapter describes the FPEE library's interaction with the NS32081 FPU and the NS32381 FPU, how to use and integrate the FPEE library with an application program, and the basic FPEE library operational details.

Before proceeding, the information presented in Section 4.2 should be reviewed. This information describes the *Series 32000* floating-point number formats and special values and defines floating-point arithmetic terminology.

5.2 FPEE LIBRARY CONFIGURATIONS

All FPEE binary libraries provided with the GNX development tool-set are configured for the NS32381 FPU by default.

5.2.1 FPEE Library Creation in a Series 32000/UNIX Environment

In a *Series 32000*/UNIX environment, the GNX tool-set can create either native or cross-development versions of the FPEE library. To build a native version of the FPEE library, execute the `make` command while in the FPEE source directory. `Make` creates a native FPEE library named `libfpe.a` that enhances/emulates the NS32381 FPU. `Make` creates a cross FPEE library named `libdb_fpe.a` that enhances/emulates the NS32381 FPU for target board applications.

To create a NS32081 version of the FPEE library, for either native or cross-development applications, edit the makefile by commenting the line that defines the makescript variable PPFLAGS=-DFPU381 and assembler flags to reflect NS32081.

The FPU type does not affect the name of the library.

5.2.2 Cross-development FPEE Library Creation

The GNX cross-development tool-set on VAX/VMS or VAX/UNIX 4.3 systems creates only cross-development versions of the FPEE library.

To build an FPEE library on a VAX/UNIX system, execute the `make` command (**`make all`**) while in the FPEE source directory. This creates an FPEE library named `libfpe.a` that enhances and emulates the NS32381 FPU.

To create an NS32081 version of the FPEE library, edit the `makefile` by uncommenting the line that defines the makescript variable PPFLAGS = -DFPU381 and change the assembler flags to reflect NS32081.

The FPU type does not affect the name of the library.

The FPEE library built in VAX/VMS environment is implemented in a DCL command file (*i.e.*, a file named `make.com`). To build the FPEE library, execute the `make.com` file (*i.e.*, **`@make all`**) while in the FPEE source directory. To create an NS32081 version of the FPEE library, edit the file `make.com` and comment the line that invokes the assembler with the /DEFINE=(FPU381) command line option and uncomment with /TARGET=(FPU081).

5.3 INTEGRATING FPEE WITH AN APPLICATION

The integration of the FPEE library imposes two mandatory requirements upon the application.

First, the application must initialize the FPU's status register (FSR). See Section 5.3.1 and Section 5.3.2 for details. This is especially critical if the FPEE library is enhancing the FPU. Initializing the FSR (Floating-point Status Register) synchronizes the FPU's hardware FSR with that of the FSR's software image in the FPEE library.

Second, CPU exception dispatch-table trap-descriptors for FPU (slave) and undefined instructions must be set to their corresponding entry points in the FPEE library. Use the `fpgtrpvctr` and `fpstrpvctr` functions to fetch and set the FPU trap handler (see Sections 4.3.30 and 4.3.37).

Only applications that require full FPU emulation (no FPU present) use the undefined instruction trap. Those applications that use the FPEE library to enhance the FPU need only the FPU trap, and the undefined trap initialization code may be removed from the source.

5.3.1 Integrating FPEE with Series 32000/UNIX Applications

The user can link an application program with the FPEE library and execute code in a GENIX V development environment. This may be of use to customers that want to do some initial checkup of their application.

Native applications call a special initialization routine provided with the FPEE library `libfpe.a`. `Libfpe.a` must be installed in `/lib` before linking.

In the application program just after declarations, a call is made to `fpinit_`, an FPEE initialization routine which sets the GENIX V signals (traps) for both the undefined instruction and the FPU trap. Upon return from this routine, the FSR is initialized and then the application program is called. (The source to `fpinit_` is in the `fpinitn.c` file of the FPEE sources.) Normally, `fpinit_` is called with an assembly instruction (*e.g.*, `asm("bsr _fpinit_");`).

5.3.2 Cross Application FPEE Integration

In cross-development mode, the FPEE library is supported by several functions from the *Series 32000* Development Board Monitors.

On a VAX/UNIX development host, a cross-application must either include a call to the `INIT__` routine (in source file `fpinitx.s`) prior to any floating-point operations or use the –f flag on the compiler invocation line to link with FPEE. The following two examples link FPEE to an application program in the file `yourprog.c`:

```
nmcc (mif yourprog.c
```

or

```
nmcc (mic yourprog.c
nmeld GNXDIR/lib/fcrt0.o yourprog.o -lfpe -lc
```

On a VAX/VMS development host, the linking is done in the following two steps:

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

On a *Series 32000*/UNIX system, cross-application linking to FPEE must be explicitly requested. For example,

```
cc -c yourprog.c
ld GNXDIR/lib/db_fcrt0.o yourprog.o -ldb_fpe -ldb_c
```

5.3.3 FPEE Library and the Math Library Integration

The math library routines, when used with the FPEE library, provide a full IEEE 754 math environment. The math library provides many routines that control the FPEE library actions by providing high-level language routines to manipulate the FPU's FSR. Section 4.2.10 completely details the IEEE 754 math environment requirements and its relationship to the FPEE library.

An important difference between the math library and the FPEE library is the initial value of the FSR. The FPEE library initialization routine (*i.e.* `INIT__` for cross-development; `fpinit` for execution under GENIX V) initializes the FSR to a value that does not assume presence of the FPEE software. This FSR value does not enable the complete IEEE 754 math environment functionality. To initialize the FSR to the IEEE 754 specified default, use the the `fp_procentry` function in the math library. This function assumes the presence of the FPEE software but does not require it for operation. If FPEE is not used, the only effect is the loss of the FPEE software-supported features.

5.3.4 FPEE Error Handling Routines

The FPEE library provides the application with five FPU trap-exception routines. There are routines for the following FPU traps: underflow, overflow, inexact result, invalid operation, and divide by zero. Application program execution is transferred to the appropriate routine when the application performs an operation which results in an exception and that exception's FSR trap-enable flag is set.

As provided with the FPEE library, these routines simply output an error message and then halt the application program execution. This is the minimum, generic IEEE 754 requirement; elaboration of these routines is application-specific and the responsibility

of the application program. Typically, an application program elaborates error routine after determining which type of floating-point operation caused the exception and then returns a value which allows the application program to continue execution.

The FPEE library implements a technique that allows an application program to quickly determine the error-causing floating-point instruction. Upon entry to one of these routines, a coded integer value is available which identifies the offending floating-point instruction. (Table 5-1 provides the value-mapping code). From this information, the application program can determine the type of error causing operand (*i.e.* byte, word, double-word, single- or double-precision floating-point) and, therefore, return the correct type of result.

This FPEE error mechanism is implemented in a generic fashion and requires modification before integration with any special application needs. The default error routines for native applications are in the source file `fperrn.c`; the error routines for cross applications are in the source file `fperrx.s`.

5.4 FPEE OPERATIONAL DETAILS

Floating-point operations for the *Series 32000* family may be implemented with the *Series 32000* FPUs alone or with the FPEE library alone; however, the fastest and greatest variety of operations are provided when both the FPEE library and the *Series 32000* FPUs are present in a system. The *Series 32000* FPUs provide fast execution but do not fully meet the IEEE 754 requirements. The FPEE library does not have the speed of the *Series 32000* FPUs, but the library does provide additional functionality necessary to fulfill IEEE 754 requirements. Complete IEEE 754 conformance is achieved for double-precision arithmetic when the application program uses both the FPEE library and the math library (the math library provides the interface routines to control the IEEE 754 specified math environment).

5.4.1 Operational Overview

The FPEE library interfaces with the *Series 32000* FPU (when present) and a *Series 32000* CPU to execute or enhance floating-point operations. When the CPU encounters a floating-point instruction, it checks the Configuration register (CFG) and if the FPU is present, it transfers control to the FPU. If the FPU is not present, control transfers to the undefined instruction trap handler in the FPEE library. The FPEE library undefined instruction trap handler emulates the floating-point instruction.

If an FPU is present and a floating-point exception occurs (such as a floating-point divide-by-zero operation), the CPU generates a floating-point trap and control is transferred to the floating-point (FPU) trap handler in the FPEE library. The FPEE FPU trap handler takes appropriate action, such as returning a NAN or infinity as the result or halting execution at a specified error routine.

The transfer of control between the FPEE library, the *Series 32000* CPU, and the *Series 32000* FPU is completely application-program transparent.

Table 5-1. Instruction Codes

INSTRUCTION	CODE	INSTRUCTION	CODE
addf	33	movlf	18
addl	32	movwf	17
absf	15	movwl	16
absl	14	mulf	43
cmpf	27	mull	42
cmpl	26	negf	13
divf	29	negl	12
divl	28	polyf	59
dotf	61	polyl	58
dotl	60	roundfb	5
floorfb	9	roundlb	4
floorlb	8	roundfw	21
floorfw	25	roundlw	20
floorlw	24	roundfd	35
floorfd	39	roundld	34
floorld	38	sfsr	3
lfsr	2	scalbf	63
logbf	65	scalbl	62
logbl	64	subf	41
movbf	1	subl	40
movbl	0	truncfb	7
movdf	31	trunclb	6
movdl	30	truncfw	23
movf	11	trunclw	22
movfl	19	truncfd	37
movl	10	truncld	36

5.4.2 FPEE Enhancements to the FPU

IEEE 754 requires that exceptions (arithmetic operations on reserved operands) cause a signal. The signal may be either setting a status flag, or taking a trap, or both. The exact action must be under control of the application program. For example, the application program can specify setting a flag, but no trapping, for a specific type of exception. In this case, program execution continues despite the exception, and the numerical result of the operation causing the exception is the appropriate IEEE 754 recommended value, typically either a NAN or a signed infinity.

IEEE 754 defines five types of exceptions: underflow, overflow, divide by zero, inexact result, and invalid operation. The NS32081 and NS32381 FPUs provide status flags only for underflow and inexact result but traps for the other exceptions. In no case does the FPU allow continued execution after an exception trap.

The FPEE library implements status flags for overflow, invalid operation, and divide by zero and allows the application program to enable or disable trapping for these exceptions by using routines provided in the math library. The implementation is transparent to the application program because the *Series 32000* FPU's floating-point status register (FSR) contains bits which are under the FPEE library's software control (the FSR's Software Field Bits). The application program need only consult the value of the FSR to determine the status of FPEE software-supported flags and FPU hardware-supported flags.

The IEEE 754 enhancements to the FPU are implemented in the FPU trap handler in the FPEE library. The FPEE library FPU trap handler examines the trap enable flags to determine whether application program execution should continue. If the trap for a specific exception is disabled, the trap handler simply sets the appropriate FSR flag signaling the exception, makes sure that the correct special value is returned as the result (typically NAN or a signed infinity), and resumes execution of the application program.

Table 5-2 lists the functions implemented by the FPEE library.

Table 5-2. FPEE Library-Implemented IEEE 754 Operations

FPEE library implements these required IEEE Standard operations for double-precision arithmetic:	
Special Values	
	Plus and minus zero
	Denormalized numbers
	Plus and minus infinity
	Signaling and quiet NANs
Special Operations	
	Infinities
	NANs
	Denormalized values
Comparisons	
	Unordered
Exception Handling	
	Underflow
	Overflow
	Divide by Zero
	Invalid Operand
	Inexact Result

5.4.3 NS32081 FPU, NS32381 FPU and FPEE

There are a few differences between the NS32081 FPU and the NS32381 FPU which require consideration when using the FPEE library. The NS32381 FPU implements four additional floating-point instructions (scalb, logb, dot, and poly) and a floating-point register modified bit (RMB) in the FSR. The NS32381 FPU has eight 64-bit floating-point registers instead of eight 32-bit floating-point registers.

The FPEE library does distinguish between NS32081 FPU instruction emulation and NS32381 instruction emulation. The distinction is specified when the FPEE library is created. The FPEE library can be created to enhance/emulate either the NS32081 FPU or the NS32381 FPU instruction set. Applications must be compiled specifying the exact FPU that the FPEE library enhances/emulates.

Applications must be compiled for the correct FPEE library because the NS32381 FPU FPEE library implements eight 64-bit registers and supports the RMB bit of the FSR. The FSR RMB is supported only by the NS32381 FPEE, and applications using this bit do not work with the NS32081 version of the FPEE library. The 64-bit registers might cause some problems for assembly language routines written for the NS32081 FPU that move a 64-bit value from register to memory using two 32-bit move instructions, rather than the appropriate single 64-bit instruction. This technique does not work with the NS32381 because the NS32381 does not concatenate two adjacent 32-bit registers to form a 64-bit register; all eight NS32381 registers are 64-bit. A single 64-bit move instruction must be used to transfer register contents to memory.

5.4.4 FPEE Program Control

The FPEE software implements the full IEEE 754 math environment by using the software field in the FSR. Between the FPEE-implemented FSR bits and those of the FPU, an application can enable or disable any of the five traps (overflow, underflow, inexact result, invalid operation, and divide by zero) and check any of the five exception status flags. The FPU maintains the lower nine bits of the FSR while seven higher bits are implemented by the FPEE software. The remainder of the bits 17-31 are reserved, bit 16 is used only by the NS32381.

The FPEE library implements the software field FSR bits (9-15) for exception trap enable and exception status.

The FPEE software-implemented and supported FSR contains:

Bit:	Purpose:
0-2	Trap type
3	Underflow trap-enable flag
4	Underflow status flag
5	Inexact-result trap-enable flag
6	Inexact-result status flag
7-8	Rounding mode
9	FPU
10	Invalid-operation trap-enable flag
11	Invalid-operation status flag
12	Division-by-zero trap-enable flag
13	Division-by-zero status flag
14	Overflow trap-enable flag
15	Overflow status flag
16	Register Modified Bit (NS32381 Only)
17-31	Reserved for future use

Trap Type

The Trap type bits indicate the type of floating-point exception which occurred:

000	No trap
001	Underflow
010	Overflow
011	Division-by-zero
100	Illegal-instruction
101	Invalid-operation
110	Inexact-result
111	Reserved for future use

Rounding Mode

Rounding mode bits indicate how floating-point operations are rounded:

00	Toward nearest *
01	Toward zero
10	Toward positive infinity
11	Toward negative infinity

* if two values are equally near, towards the even value

The exception status flags, once set, remain set until explicitly cleared by writing a 0.

The FPU bit selects either FPU (NS32081 or NS32381) compatible mode of operation or IEEE 754 mode of operation. If the FPU bit is 1, the library emulates the FPU chip exactly. In IEEE 754 mode (FPU bit is 0) for double-precision arithmetic, the library operates according to the IEEE 754 Standard. Results of operations and exceptions when the FPU bit is set or cleared are given in the following paragraphs. In each case, the value of the FSR is presented with significant bits shown as either 1 or 0; "don't care" bits are shown as X.

Underflow exception:

XXXXXX0X XXX1XXXX	Return a denormalized number
XXXXXX0X XXX11XXX	Underflow trap
XXXXXX1X XXX10XXX	Return zero (non-IEEE 754 standard)
XXXXXX1X XXX11XXX	Underflow trap

Inexact result exception:

XXXXXXXX X10XXXXX	Return an inexact result
XXXXXXXX X11XXXXX	Inexact result trap

Invalid operation exception:

```
XXXX1X1X XXXXXXXX   Invalid Operation trap
XXXX100X XXXXXXXX   Return NAN **
XXXX110X XXXXXXXX   Invalid-Operation trap
```

** If the invalid operand is a denormalized number, the FPEE software returns a normalized value.

Division by zero exception:

```
XX1XXX1X XXXXXXXX   Division by zero trap
XX10XX0X XXXXXXXX   Return infinity
XX11XX0X XXXXXXXX   Division by zero trap
```

Overflow signaled:

```
1XXXXX1X XXXXXXXX   Overflow trap
10XXXX0X XXXXXXXX   Result according rounding mode
11XXXX0X XXXXXXXX   Overflow trap
```

See Section 5.4.7 on rounding mode for results.

Overflow on conversion from float to integer:

```
XXXXXX1X XXXXXXXX   Overflow trap (non-IEEE 754 Standard)
XXXXX00X XXXXXXXX   Return -1
XXXXX10X XXXXXXXX   Invalid-operation trap
```

5.4.5 FPEE Comparisons

Floating-point comparisons differ from integer comparisons because there are four possible results: unordered result, greater than, equal to, and less than. The unordered result occurs from comparisons of operands such as NANs.

The FPEE software sets bits in the Processor Status Register (PSR) of the *Series 32000* CPU to indicate the result of a floating-point comparison. The FPEE library uses the N, Z, and L bits:

Comparison Result	Bit Set	Bits Cleared
Operands are equal	Z	N and L
Operand1 is less than Operand2	None	Z, N, and L
Operand2 is less than Operand1	N	Z and L
Unordered	L	Z and N

All comparisons with an unordered result use the FPEE library since the NS32081 and NS32381 FPUs generate an FPU trap when one of the operands of a comparison is a reserved operand.

5.4.6 FPEE Exception Handling

The FPEE library implements six exception handling routines:

- Invalid-operation
- Division-by-zero
- Overflow
- Underflow
- Inexact-result
- Illegal-instruction

Library handling of these exceptions is internal and transparent to the application.

These floating-point exceptions lead to a run-time error or to results specified by the IEEE 754 Standard. Note that the NS32081 and NS32381 FPUs (and emulation in FPU mode) do not handle exceptions for underflow according to the IEEE standards. For underflow, the FPUs return zero.

If an exception occurs and its trap enable flag in the FSR is set, application program execution is transferred to the appropriate error handling routine.

If an exception occurs and its trap enable flag in the FSR is not set, application program execution continues after the FPU trap handler services the exception by setting the exception status flag and returning the IEEE 754 specified result. It is the application program's responsibility to check for set exception status flags in the FSR.

Invalid operation exceptions occur when a floating-point operation (other than a move) is attempted on a reserved operand. The following are operations which cause an invalid operation exception:

- An operand which is a NAN
- A result of a remainder operation, x REM y (remainder of x divided by y), where y is zero or x is infinity
- Infinity plus negative infinity or infinity minus infinity
- Multiplying zero by infinity
- Dividing zero by zero
- Dividing infinity by infinity
- The operand is a denormalized number. If the invalid operation trap is disabled, the FPEE software returns a normalized number.
- Comparing with "<" or ">" when the relation is unordered

The Division-by-zero exception occurs when the divisor of a floating-point operation is zero and the dividend is a finite nonzero number.

The Overflow exception occurs when the result of a floating-point operation is finite but too large to be represented in the given format. Any decimal value whose magnitude is larger than the following causes the Overflow exception:

- 3.4028235 E 38 for single-precision
- 1.797693134862316 E 308 for double-precision

The Underflow exception occurs when the result of a floating-point operation is not zero and the exponent is too small to be represented in the given format. This exception may also occur for denormalized numbers. Any decimal value whose magnitude is smaller than the following causes the Underflow exception:

- 1.1754943 E -38 for single-precision
- 2.225073858507201 E -308 for double-precision

The Inexact-result exception occurs when the rounded result of a floating-point is not exact or when an overflow occurs and the overflow trap is not enabled.

Non-implemented operation codes cause the Illegal-Instruction exception.

5.4.7 FPEE Rounding Modes

The rounding modes affect normal calculations which require rounding and the returned result for the overflow exception.

The FPEE library implements overflow-exception-returned results. If the overflow exception trap is disabled, the results returned are shown in Table 5-3.

Table 5-3. Default Return Values for Overflow Exceptions

ROUNDING MODE	SIGN OF THE INTERMEDIATE RESULT	RESULT RETURNED BY THE FPEE SUPPORT LIBRARY
Toward Nearest	+	Positive Infinity
	–	Negative Infinity
Toward Zero	+	+3.4028235 E 38 (single-precision) +1.797693134862316 E 308 (double-precision)
	–	–3.4028235 E 38 (single-precision) –1.797693134862316 E 308 (double-precision)
Toward Negative Infinity	+	+3.4028235 E 38 (single-precision) +1.797693134862316 E 308 (double-precision)
	–	Negative Infinity
Toward Positive Infinity	+	Positive Infinity
	–	–3.4028235 E 38 (single-precision) –1.797693134862316 E 308 (double-precision)

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 rightmost argument is 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 neccessary), floating-point values use eight bytes and are represented as *long* values; structures (records) use a multiple of double-words.

NOTE: Stack alignment is maintained by all GNX — Version 3 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 FIXED_FRAME (-OF) 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 value may be changed by a called routine. Also included in this list of scratch registers is the long register L1 of the NS32381 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 NS32381) are used, their values should be saved (onto the stack or in temps) 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.

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

A long floating-point value that returns from a function, returns in register pair F0-F1. A float-returning function returns the value in register F0.

If a function returns a structure, 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 during execution of the return statement. Note that functions that return structures must be correctly declared as such even if the return value is ignored.

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

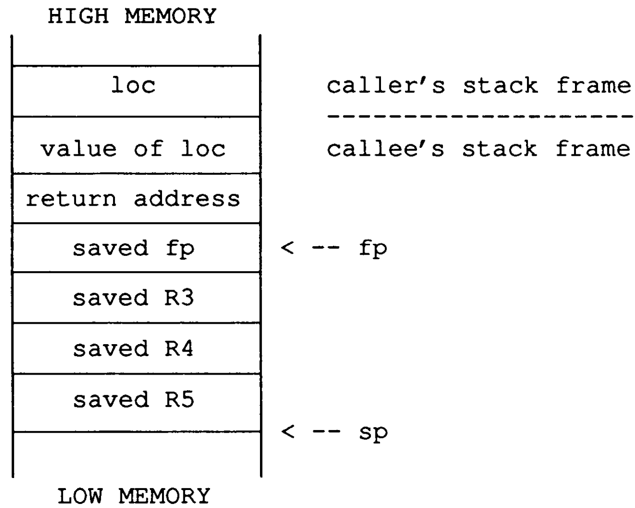
ifunc(p1)
int p1;
{
    int i, j, k;
    j = 0;
    for (i = 1; i <= p1; i++)
        j = j + f(i);
    return(j);
}
```

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:



INDEX

** function, see pow 4-65
1 EPERM 2-1
2 ENOENT 2-1
5 EIO 2-1
6 ENXIO 2-1
9 EBADF 2-1
12 ENOMEN 2-1
13 EACCES 2-1
14 EFAULT 2-2
15 ENOTBLK 2-2
16 EBUSY 2-2
17 EEXIST 2-2
19 ENODEV 2-2
20 ENOTDIR 2-2
21 EISDIR 2-2
22 EINVAL 2-2
23 ENFILE 2-2
24 EMFILE 2-2
25 ENOTTY 2-2
26 ETXTBSY 2-2
27 EFBIG 2-2
28 ENOSPC 2-2
30 EROFS 2-2
32 – bit move instruction 5-9
33 EDOM 2-3
62 ELOOP 2-3
63 ENAMETOOLONG 2-3
64 – bit move instruction 5-9

A

abort routine 3-2
abs function, see cabs 4-21
abs function, see fabs 4-34
abs routine 3-3
Absolute ceiling 3-13
Absolute floor 3-13
Absolute value 3-13
Absolute value function
  cabs 4-21
  fabs 4-34
  fcabs 4-21
  ffabs 4-34
Accessing math library functions 4-10
acos function 4-13
acosh function 4-14
Allocate memory in heap 2-17
Arccosine function
  acos 4-13
  facos 4-13
Arc-hyperbolic cosine function
  acosh 4-14
  facosh 4-14
Arc-hyperbolic sine function
  asinh 4-16
  fasinh 4-16
Arc-hyperbolic tangent function
  atanh 4-19
  fatanh 4-19
Arc-sine function
  asin 4-15
  fasin 4-15
arctan function, see atan 4-17
Arctangent function
  atan 4-17
  atan2 4-18
  fatan 4-17
  fatan2 4-18
asctime 3-7
asin function 4-15
asinh function 4-16
Assign buffering to a stream 3-44
atan function 4-17
atan2 function 4-18
atanh function 4-19
atof routine 3-4
  atoi 3-4
  atol 3-4
atoi routine 3-4
atol routine 3-4

B

Bad address 2-2
Bad file number 2-1
bcmp routine 3-5
bcopy routine 3-5
Bessel function
  j0 4-20
  j1 4-20
  jn 4-20
  y0 4-20
  y1 4-20
  yn 4-20
Bias 4-4
Bias value 4-4
Bit and byte string operations 3-5
Block device required 2-2
bstring routines 3-5
  bcmp 3-5
  bcopy 3-5
  bzero 3-5
  ffs 3-5
Buffered binary I/O 3-16
Byte format 4-2
bzero routine 3-5

C

cabs function 4-21
Calling sequence A-1
calloc routine 3-24
cbrt function 4-23
ceil function 4-24
ceil routine 3-13
Ceiling function
  ceil 4-24
  fceil 4-24
clearerr 3-12
Close a file 2-7
Close or flush a stream 3-11
close system call 2-7
compound function 4-25
Convert ASCII to numbers 3-4
Convert date and time to ASCII 3-7
copysign function 4-26
Copysign function
  copysign 4-26
  fcopysign 4-26
cos function 4-27
cosh function 4-28
Cosine function
  cos 4-27
  cosh 4-28
  fcos 4-27
  fcosh 4-28
CPU generates 5-5
creat call 2-6
creat system call 2-8
Create a new file 2-8
Cross application FPEE integration 5-3
ctime routine 3-7
  asctime 3-7
  gmtime 3-7
  localtime 3-7
  timezone 3-7
Cube root function
  cbrt 4-23
  fcbrt 4-23

D

dacosh function, see acosh 4-14
dasinh function, see asinh 4-16
datanh function, see atanh 4-19
DB library 1-1, 2-3, 3-1
dcbrt function, see cbrt 4-23
dceil function, see ceil 4-24
dcompound function, see compound 4-25
ddrem function, see drem 4-29
Denormalized numbers 4-8
derf function, see erf 4-30
Description of system calls 2-3
dexp2 function, see exp2 4-32
dexpm1 function, see expm1 4-33
dfinite function, see finite 4-35
dfloor function, see floor 4-36
dhypot function, see hypot 4-56
dinf function, see inf 4-57
divide by zero 4-6
Divide by zero exception 5-13
Division by zero exception 5-11
dlog1p function, see log1p 4-60
dlog2 function, see log2 4-61
dneg function, see neg 4-62
Documentation conventions 1-2
Double-precision function
  acos 4-13
  acosh 4-14
  asin 4-15
  asinh 4-16
  atan 4-17
  atan2 4-18
  atanh 4-19
  cabs 4-21
  cbrt 4-23
  ceil 4-24
  compound 4-25
  copysign 4-26
  cos 4-27
  cosh 4-28
  drem 4-29
  exp 4-31
  exp2 4-32
  expm1 4-33
  fabs 4-34
  ffinite 4-35
  finite 4-35
  floor 4-36
  hypot 4-56
  inf 4-57
  log 4-58
  log10 4-59
  log1p 4-60
  log2 4-61
  neg 4-62
  nextdouble 4-63
  pi 4-64
  pow 4-65
  relation 4-68
  rem 4-69
  rint 4-71
  sin 4-72
  sinh 4-74
  sqrt 4-75
  tan 4-76
  tanh 4-77
Double-precision numbers 4-3
Double-word format 4-3
dpi function, see pi 4-64
drelation function, see relation 4-68
drem function 4-29
dremainder function, see rem 4-69
drint function, see rint 4-71
dtanh function, see atanh 4-19
Dummy call restrictions 1-2

Dummy implementations 1-2, 2-3
Dummy routines, list of 1-2

E

ecvt routine 3-9
  fcvt 3-9
  gcvt 3-9
Environment control functions 4-9
erf function 4-30
errno 2-1
Errof function
  ferf 4-30
Error function
  erf 4-30
Errors, list of 2-1
Errors
  1 EPERM 2-1
  2 ENOENT 2-1
  5 EIO 2-1
  6 ENXIO 2-1
  9 EBADF 2-1
  12 ENOMEN 2-1
  13 EACCES 2-1
  14 EFAULT 2-2
  15 ENOTBLK 2-2
  16 EBUSY 2-2
  17 EEXIST 2-2
  19 ENODEV 2-2
  20 ENOTDIR 2-2
  21 EISDIR 2-2
  22 EINVAL 2-2
  23 ENFILE 2-2
  24 EMFILE 2-2
  25 ENOTTY 2-2
  26 ETXTBSY 2-2
  27 EFBIG 2-2
  28 ENOSPC 2-2
  30 EROFS 2-2
  33 EDOM 2-3
  62 ELOOP 2-3
  63 ENAMETOOLONG 2-3
Euclidean distance function
  fhypot 4-56
  hypot 4-56
Exception status flag function
  fp_setexptn 4-47
  fp_tstexptn 4-54
Exception status flag functions 4-9
Exception status flags 5-10
Exception trap functions 4-9
Exception traps 4-9
Exceptions 4-6
  checking 4-10
exit() call 2-6
exit routine 3-10
_exit system call 2-10
exp function 4-31
exp2 function 4-32
expm1 function 4-33
Exponential function
  exp 4-31
  exp2 4-32
  expm1 4-33
  fexp 4-31
  fexp2 4-32
  fexpm1 4-33

F

fabs function 4-34
fabs routine 3-13
facos function 4-13
fasin function 4-15
fasinh function 4-16
fatan function 4-17
fatan2 function 4-18
fatanh function 4-19
fcabs function 4-21
fcbrt function 4-23
fceil function 4-24
fclose routine 3-11
  fflush 3-11
fcompound function 4-25
fcopysign function 4-26
fcos function 4-27
fcosh function 4-14, 4-28
fcvt routine 3-9
fdopen routine 3-14
fdrem function 4-29
feof routine 3-12
ferf function 4-30
ferror routine 3-12
  clearerr 3-12
  feof 3-12
  fileno 3-12
fexp function 4-31
fexp2 function 4-32
fexpm1 function 4-33
ffabs function 4-34
ffinite function 4-35
ffloor function 4-36
fflush routine 3-11
ffmod function 4-37
ffs routine 3-5
fgetc routine 3-19
fgets routine 3-21
fhypot function 4-56
file descriptor 2-3
File exists 2-2
File name too long 2-3
File table overflow 2-2
File too large 2-2
fileno 3-12
Find name of a terminal 3-23
finf function 4-57
finite function 4-35
Floating-point comparisons 5-12

Floating-point emulation library 1-1
Floating-point enhancement and emulation 5-1
Floating-point exception handling 5-12
Floating-point exceptions 5-12
Floating-point format 4-3
Floating-point library 4-1
Floating-point numbers 4-6
  Reserved operand values 4-6
Floating-point range 4-4
Floating-point
  divide by zero 4-6
  inexact result 4-6
  invalid operation 4-6
  overflow 4-6
  underflow 4-6
flog function 4-58
flog10 function 4-59
flog1p function 4-60
flog2 function 4-61
floor function 4-36
floor routine 3-13
  ceil 3-13
  fabs 3-13
fmod function 4-37
  ffmod 4-37
fmodf function 4-38
fneg function 4-62
fopen routine 3-14
  fdopen 3-14
  freopen 3-14
Formatted input conversion 3-40
Formatted output conversion 3-29
FPEE 5-1
FPEE and math library integration 5-4
FPEE enhancements to FPU 5-7
FPEE error handling 5-4
FPEE error mechanism 5-5
FPEE libraries 4-6
FPEE libraries implement 4-6
FPEE library configurations 5-2
FPEE library creation
  cross-development 5-2
  native 5-2
FPEE library implements 5-7, 5-9, 5-14
FPEE library supports 4-8
FPEE library
  cross application 4-6
  native application 4-6
FPEE operation, overview of 5-5
FPEE operations 5-5
FPEE program control 5-9
FPEE rounding modes 5-14
FPEE source 5-1
fperrn.c 5-5
fperrx.s 5-5
fp_getexptn() function 4-10
fp_getexptn function 4-39
fp_getround function 4-40
fp_gettrap function 4-41
fp_gmathenv function 4-42
fpgtrpvctr function 4-44
fpi function 4-64
fpinit 5-3
fpinit routine 5-3
fpow function 4-65
fp_procentry() function 4-9
fp_procentry function 4-45, 5-4
fp_procexit function 4-46
fprintf routine 3-29
fp_setexptn function 4-47
fp_setround function 4-48
fp_settrap function 4-49
fp_smathenv function 4-50
fpstrpvctr function 4-52
fp_testtrap function 4-53
fp_tstexptn function 4-54
FPU bit selects 5-10
FPU provides 5-7
FPU Trap 4-6
FPU trap handler 5-5
FPU traps for 4-6
fputc routine 3-32
fputs routine 3-34
Fraction 4-3
fread routine 3-16
  fwrite 3-16
free routine 3-24
frelation function 4-68
frem function 4-69
freopen routine 3-14
frexp routine 3-17
  ldexp 3-17
  modf 3-17
frint function 4-71
fscanf routine 3-40
fseek routine 3-18
  ftell 3-18
  rewind 3-18
fsin function 4-72
fsinh function 4-74
fsqrt function 4-75
ftan function 4-76
ftanh function 4-77
ftell routine 3-18
Functions
  acos 4-13
  acosh 4-14
  asin 4-15
  asinh 4-16
  atan 4-17
  atan2 4-18
  atanh 4-19
  cabs 4-21
  cbrt 4-23
  ceil 4-24
  compound 4-25
  copysign 4-26
  cos 4-27
  cosh 4-28
  drem 4-29

erf 4-30
exp 4-31
exp2 4-32
expm1 4-33
fabs 4-34
facos 4-13
fasin 4-15
fasinh 4-16
fatan 4-17
fatan2 4-18
fatanh 4-19
fcabs 4-21
fcbrt 4-23
fceil 4-24
fcompound 4-25
fcopysign 4-26
fcos 4-27
fcosh 4-14, 4-28
fdrem 4-29
ferf 4-30
fexp 4-31
fexp2 4-32
fexpm1 4-33
ffabs 4-34
ffinite 4-35
ffloor 4-36
fhypot 4-56
finf 4-57
finite 4-35
flog 4-58
flog10 4-59
flog1p 4-60
flog2 4-61
floor 4-36
fmod 4-37
fmodf 4-38
fneg 4-62
fp_getexptn 4-39
fp_getround 4-40
fp_gettrap 4-41
fp_gmathenv 4-42
fpgtrpvctr 4-44
fpi 4-64
fpow 4-65
fp_procentry 4-45
fp_procexit 4-46
fp_setexptn 4-47
fp_setround 4-48
fp_settrap 4-49
fp_smathenv 4-50
fpstrpvctr 4-52
fp_testtrap 4-53
fp_tstexptn 4-54
frelation 4-68
frem 4-69
frint 4-71
fsin 4-72
fsinh 4-74
fsqrt 4-75
ftan 4-76
ftanh 4-77
gamma 4-55
hypot 4-56
inf 4-57
j0 4-20
j1 4-20
jn 4-20
log 4-58
log10 4-59
log1p 4-60
log2 4-61
neg 4-62
nextdouble 4-63
nextfloat 4-63
pi 4-64
pow 4-65
randomx 4-67
relation 4-68
rem 4-69
rint 4-71
sin 4-72
sinh 4-74
sqrt 4-75
tan 4-76
tanh 4-77
y0 4-20
y1 4-20
yn 4-20
fwrite routine 3-16

G

gamma function 4-55
gcvt routine 3-9
Generate a fault 3-2
Get a string from a stream 3-21
Get character or word from stream 3-19
Get descriptor table size 2-11
getc routine 3-19
fgetc 3-19
getchar 3-19
getw 3-19
getchar routine 3-19
getdtablesize 2-7
getdtablesize() 2-15
getdtablesize system call 2-11
getpid() call 2-3
gets routine 3-21
fgets 3-21
getw routine 3-19
gmtime routine 3-7
Group ID 2-3

H

Hyperbolic sine function
  fsin 4-74
  sin 4-74
hypot function 4-56

I

IEEE 754 5-1, 5-4
IEEE 754 enhancements to FPU 5-7
IEEE 754 4-6
  compliance 4-6
Implemented IEEE 754 operations, list of 5-8
index routine 3-47
inexact result 4-6
Inexact result exception 5-11, 5-13
inf function 4-57
Infinity 4-8
Infinity function
  finf 4-57
  inf 4-57
INIT__ 5-4
Initialize random number generator 4-67
  initrand 4-67
Initialize the FSR 5-4
initrand 4-67
initstate routine 3-36
Insert/remove element from queue 3-22
insque routine 3-22
  remque 3-22
Instruction codes, list of 5-6
Integer absolute value 3-3
Integer format 4-2
Integer formats
  byte format 4-2
  double-word format 4-3
  word format 4-3
Integral value function
  frint 4-71
  rint 4-71
Introduction, math library 4-1
Invalid argument 2-2
invalid operation 4-6
Invalid operation exception 5-11, 5-13
I/O error 2-1
Is a directory 2-2
isatty routine 3-23

J

j0 function 4-20
j1 function 4-20
jn function 4-20

L

ldexp routine 3-17
libdb_fpe.a 5-2
libfpe.a 5-2, 5-3
libfpe.a, installed in 5-3
Library handling of exceptions 5-12
ln function, see log 4-58
localtime routine 3-7
log function 4-58
log10 function 4-59
log1p function 4-60
log2 function 4-61
Logarithm function
  flog 4-58
  flog10 4-59
  flog1p 4-60
  flog2 4-61
  log 4-58
  log10 4-59
  log1p 4-60
  log2 4-61
longjmp routine 3-46
lseek system call 2-12

M

make.com 5-2
malloc routine 3-24
  calloc 3-24
  free 3-24
  realloc 3-24
Mantissa 4-4
Math argument 2-3
Math environment function
  fp_gmathenv 4-42
  fpgtrpvctr 4-44
  fp_smathenv 4-50
  fpstrpvctr 4-52
Math environment functions, using 4-9
Math environment variables 4-6
Math library 1-1
math library functions 4-12
Math library provides 4-6
math library provides 4-9
Math library provides 5-4, 5-5
Maximum floating-point values 4-4
memccpy routine 3-26
memchr routine 3-26
memcmp routine 3-26
memcpy routine 3-26
Memory allocator 3-24
Memory routines
  memccpy 3-26
  memchr 3-26
  memcmp 3-26
  memcpy 3-26
  memset 3-26
memset routine 3-26

Minimum floating-point values 4-4
Min/max floating-point values, figure of 4-5
Min/max values, table of 4-4
mod function, see fmod 4-37
modf function, see fmodf 4-38
modf routine 3-17
Mount device busy 2-2
Move read/write pointer 2-12

N

NAN 4-12
Native application FPEE integration 5-3
neg function 4-62
Negation function
  fneg 4-62
  neg 4-62
Neighbor function
  nextdouble 4-63
nextdouble function 4-63
nextfloat function 4-63
No space left on device 2-2
No such device 2-2
No such device or address 2-1
No such file or directory 2-1
Non-local goto 3-46
Normalized floating-point number 4-3
Not a directory 2-2
Not a Number (NAN) 4-7
Not a typewriter 2-2
Not enough core 2-1
Not owner 2-1
NS32081 and NS32381 differences 5-8
NS32081/NS32381 and FPEE 5-8
NS32381 has 5-8
NS32381 implements 5-8
NS32381 version FPEE library, creating
  UNIX 5-2
  VMS 5-2
Number formats 4-2
  double-precision numbers 4-3
  floating-point format 4-3
  integer format 4-2
  single-precision numbers 4-3

O

Open a file for reading 2-14
Open a file for writing 2-14
Open a file to create a new file 2-14
Open a stream 3-14
open call 2-6
open system call 2-14
Operating system call simulation 1-1
Output conversion 3-9
overflow 4-6
Overflow exception 5-13
Overflow exceptions returned values, list of 5-15
Overflow on conversion from float to integer 5-11
Overflow signaled 5-11

P

Permission denied 2-1
perror routine 3-28
pi function 4-64
pow function 4-65
power function, see pow 4-65
Power function
  fpow 4-65
  pow 4-65
Predicate function
  ffinite 4-35
  finite 4-35
printf routine 3-29
  fprintf 3-29
  sprintf 3-29
Process entry function
  fp_procentry 4-45
Process exit function
  fp_procexit 4-46
Process ID 2-3
Push character back into input stream 3-50
Put a string on a stream 3-34
Put character/word on a stream 3-32
putc routine 3-32
  fputc 3-32
  putchar 3-32
  putw 3-32
putchar routine 3-32
puts routine 3-34
  fputs 3-34
putw routine 3-32

Q

QNAN 4-12
qsort routine 3-35
Quicker sort 3-35
Quiet NAN 4-7

R

Random number generator 3-36
  randomx 4-67
random routine 3-36
  initstate 3-36
  setstate 3-36
  srandom 3-36
randomx 4-67
Read input 2-16
Read mode 2-3
read system call 2-16
Read-only file system 2-2
realloc routine 3-24

re_comp routine 3-38
re_exec routine 3-38
Regex routines
  re_comp 3-38
  re_exec 3-38
Regular expression handler 3-38
relation function 4-68
rem function 4-69
remainder function, see rem 4-69
Remainder function
  drem 4-29
  fdrem 4-29
  frem 4-69
  rem 4-69
Remove directory entry of a file 2-18
remque routine 3-22
Reposition a stream 3-18
Reserved operand values 4-2
Reserved operand values and operations 4-6
Reserved operand values
  denormalized numbers 4-8
  infinity 4-8
  Not a Number (NAN) 4-7
Return codes 2-1
Return Value A-2
rewind routine 3-18
rindex routine 3-47
rint function 4-71
Rounding mode 5-10
Rounding mode function
  fp_getround 4-40
  fp_setround 4-48
Routines for changing generators 3-36
Routines that use simulated system calls
  list of 2-4
Routines
  abort 3-2
  abs 3-3
  asctime 3-7
  atof 3-4
  atoi 3-4
  atol 3-4
  bcmp 3-5
  bcopy 3-5
  bzero 3-5
  calloc 3-24
  ceil 3-13
  clearerr 3-12
  ctime 3-7
  ecvt 3-9
  exit 3-10
  fabs 3-13
  fclose 3-11
  fcvt 3-9
  fdopen 3-14
  feof 3-12
  ferror 3-12
  fflush 3-11
  ffs 3-5
  fgetc 3-19
  fgets 3-21
  fileno 3-12
  floor 3-13
  fopen 3-14
  fprintf 3-29
  fputc 3-32
  fputs 3-34
  fread 3-16
  free 3-24
  freopen 3-14
  frexp 3-17
  fscanf 3-40
  fseek 3-18
  ftell 3-18
  fwrite 3-16
  gcvt 3-9
  getc 3-19
  getchar 3-19
  gets 3-21
  getw 3-19
  gmtime 3-7
  index 3-47
  initstate 3-36
  insque 3-22
  isatty 3-23
  ldexp 3-17
  localtime 3-7
  longjmp 3-46
  malloc 3-24
  memccpy 3-26
  memchr 3-26
  memcmp 3-26
  memcpy 3-26
  memset 3-26
  modf 3-17
  perror 3-28
  printf 3-29
  putc 3-32
  putchar 3-32
  puts 3-34
  putw 3-32
  qsort 3-35
  random 3-36
  realloc 3-24
  re_comp 3-38
  re_exec 3-38
  remque 3-22
  rewind 3-18
  rindex 3-47
  scanf 3-40
  setbuf 3-44
  setbuffer 3-44
  setjmp 3-46
  setlinebuf 3-44
  setstate 3-36
  sprintf 3-29
  srandom 3-36
  sscanf 3-40
  strcat 3-47
  strchr 3-47

strcmp 3-47
strcpy 3-47
strlen 3-47
strncat 3-47
strncmp 3-47
strncpy 3-47
strrchr 3-47
swab 3-49
sys_errlist 3-28
timezone 3-7
ungetc 3-50

S

sbrk system call 2-17
scanf routine 3-40
fscanf 3-40
sscanf 3-40
setbuf routine 3-44
setbuffer 3-44
setlinebuf 3-44
setbuffer routine 3-44
setjmp routine 3-46
longjmp 3-46
setlinebuf routine 3-44
setstate routine 3-36
sign function, see copysign 4-26
Signals 4-6
Simulated system calls 2-6
sin function 4-72
Sine function
fsin 4-72
sin 4-72
Single-precision function
facos 4-13
fasin 4-15
fasinh 4-16
fatan 4-17
fatan2 4-18
fatanh 4-19
fcabs 4-21
fcbrt 4-23
fceil 4-24
fcompound 4-25
fcopysign 4-26
fcos 4-27
fcosh 4-14, 4-28
fdrem 4-29
fexp 4-31
fexp2 4-32
fexpm1 4-33
ffabs 4-34
ffloor 4-36
fhypot 4-56
finf 4-57
flog 4-58
flog10 4-59
flog1p 4-60
flog2 4-61
fneg 4-62
fpi 4-64
fpow 4-65
frelation 4-68
frem 4-69
frint 4-71
fsin 4-72
fsinh 4-74
fsqrt 4-75
ftan 4-76
ftanh 4-77
nextfloat 4-63
Single-precision numbers 4-3
sinh function 4-74
SNAN 4-12
Split into mantissa and exponent 3-17
sprintf routine 3-29
sqrt function 4-75
Square root function
fsqrt 4-75
sqrt 4-75
srandom routine 3-36
sscanf routine 3-40
Standard calling convention A-1
Status flag function 4-39
fp_getexptn 4-39
strcat routine 3-47
strchr routine 3-47
strcmp routine 3-47
strcpy routine 3-47
Stream status inquiries 3-12
String operations 3-47
String routines 3-47
index 3-47
rindex 3-47
strcat 3-47
strchr 3-47
strcmp 3-47
strcpy 3-47
strlen 3-47
strncat 3-47
strncmp 3-47
strncpy 3-47
strrchr 3-47
strlen routine 3-47
strncat routine 3-47
strncmp routine 3-47
strncpy routine 3-47
strrchr routine 3-47
Support libraries 1-1
swab routine 3-49
Swap bytes 3-49
sys_errlist routine 3-28
System call dependencies 1-1
System calls 1-1, 2-1
System calls, summary of 2-3
System calls
close 2-7
creat 2-6, 2-8
description of 2-3

dummy implementations 1-2
exit() 2-6
_exit 2-10
getdtablesize 2-11
implemented 1-1
lseek 2-12
open 2-6, 2-14
read 2-16
sbrk 2-17
simulated 2-6
unlink 2-18
write 2-20

T

tan function 4-76
Tangent function
ftan 4-76
ftanh 4-77
tan 4-76
tanh 4-77
tanh function 4-77
Terminate a process 2-10
Terminate a process after flushing output 3-10
Text file busy 2-2
timezone 3-7
Too many levels of symbolic links 2-3
Too many open files 2-2
Trap 4-6
Trap enable flag function
fp_gettrap 4-41
fp_settrap 4-49
fp_testtrap 4-53
Trap handler 4-7
Trap type 5-10

U

ULP 4-12
underflow 4-6
Underflow exception 5-10, 5-13
ungetc routine 3-50
unlink system call 2-18
User ID 2-3

V

Values from functions 2-1

W

Without FPEE trap handler 4-9
Word format 4-3
Write mode 2-3
Write on a file 2-20
write system call 2-20

Y

y0 function 4-20
y1 function 4-20
yn function 4-20

National Semiconductor

MICROCOMPUTER SYSTEMS DIVISION

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) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada 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:

____

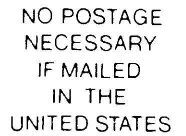
____

____

____

____

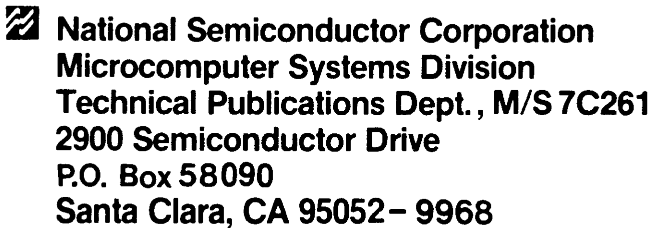
GNX — Version 3 Support Libraries Reference Manual



BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE



National Semiconductor Corporation
Microcomputer Systems Division
Technical Publications Dept., M/S 7C261
2900 Semiconductor Drive
P.O. Box 58090
Santa Clara, CA 95052-9968

Software Problem Report

Name: ______________________________

Street: ______________________________

City: ______________________ State: ______ Zip: ____________

Phone: ______________________ Date: ____________

Instructions

Use this form to report bugs, or suggested enhancements. Mail the form to National Semiconductor. Technical Support may be contacted at:

(800) 538-1866 - U.S. non CA
(800) 672-1811 - CA only
(800) 223-3248 - Canada only
((0)8141) 103-330 - West Germany

Category

☐ Software Problem ☐ Request For Software Enhancement

☐ Other ☐ Documentation Problem, Publication # ____________

Software Description

National Semiconductor Product ______________________

Version ____________ Registration # ____________

Host Computer Information ______________________

Operating System ______________________

Rev. ____________ Supplier ____________

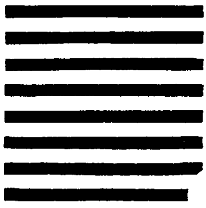
Problem Description

Describe the problem. (Also describe how to reproduce it, and your diagnosis and suggested correction.) Attach a listing if available.

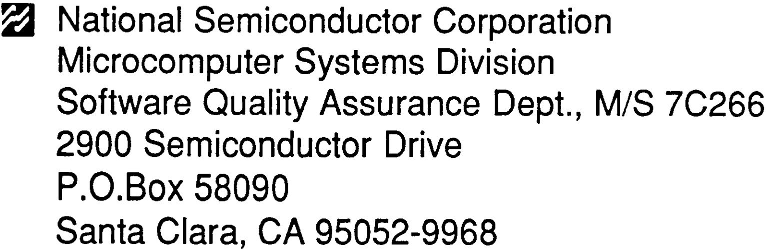
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 409 SANTA CLARA, CA



POSTAGE WILL BE PAID BY ADDRESSEE



National Semiconductor Corporation
Microcomputer Systems Division
Software Quality Assurance Dept., M/S 7C266
2900 Semiconductor Drive
P.O.Box 58090
Santa Clara, CA 95052-9968

National Semiconductor Use Only

Tech Support ____________________ Date Received ______________
Software Q.A. ____________________ Date Received ______________
Report Number ____________________
Action Taken :

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 Piedias
(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