GNX Language Tools

Version 4

Series 32000®

GNX — Version 4.4
Commands and Operations
Manual for UNIX® and MS-DOS®
Operating Systems

Customer Order Number 424010515-004

June 1992

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
4.0	May 1990	First Release. *Series 32000/EP* support.
4.1	Sept 1990	Miscellaneous updates.
4.2	Feb 1991	Sync. release. No changes.
4.3	Aug 1991	Sync. release. No changes.
4.4	June 1992	Miscellaneous updates. MS-DOS support added.

PREFACE

The *Series 32000*® GNX (GENIX™ Native and Cross-Support) Language Tools support the development of software for National Semiconductor's *Series 32000* microprocessor family. This manual describes the operation of the GNX Language Tools in a cross environment on a host development system running MS-DOS, UNIX® or a UNIX-derived operating system (*e.g.*, VAX™/UNIX 4.3bsd, SUN™/SunOS, *Series 32000*®/System V).

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

CONTENTS

Chapter 1 OVERVIEW

1.1 INTRODUCTION 1-1
1.2 GNX LANGUAGE TOOLS 1-1
1.3 INSTALLATION 1-3
1.4 MANUAL ORGANIZATION 1-3
1.5 OTHER MANUALS 1-3

Chapter 2 USING THE GNX LANGUAGE TOOLS

2.1 INTRODUCTION 2-1
2.2 GNX UTILITIES 2-2
2.2.1 C Optimizing Compiler 2-2
2.2.2 Pascal Optimizing Compiler 2-2
2.2.3 Assembler 2-3
2.2.4 Linker 2-3
2.2.5 DBUG - Multi-Window Symbolic Debugger 2-3
2.2.6 Source Level Profiler (sprof) 2-4
2.2.7 Nburn 2-4
2.2.8 Nar Librarian 2-4
2.2.9 Ncmp 2-5
2.2.10 Nlorder 2-5
2.2.11 Nnm 2-5
2.2.12 Nsize 2-5
2.2.13 Nstrip 2-5
2.3 RECOMMENDED INSTALLATION 2-5
2.3.1 Directly Executable Programs 2-6
2.3.2 Indirectly Executable Programs 2-6
2.3.3 Include Files 2-6
2.3.4 Libraries 2-7

Chapter 3 CROSS DEVELOPMENT

3.1 INTRODUCTION 3-1
3.2 SOME BASIC CONCEPTS 3-1
3.3 GNX SUPPORT LIBRARIES 3-2
3.4 MONITORS 3-4
3.5 COMPILING A PROGRAM 3-4
3.6 A SAMPLE DEBUG SESSION 3-5

3.7 USING NBURN 3-7

Chapter 4 GNX TARGET SETUP

4.1 INTRODUCTION 4-1
4.2 GNX TARGET SETUP (GTS) 4-1
4.2.1 Interactive Mode 4-1
4.2.2 Non-Interactive Mode 4-1
4.3 TARGET CONFIGURATION FILE CONTENTS 4-2
4.3.1 Description of Parameters 4-2
4.3.2 Choices 4-2
4.3.3 Syntax of Target Configuration File 4-4
4.4 TARGET CONFIGURATION FILE LOCATION 4-6
4.5 GNXENV.H 4-6
4.6 OVERRIDING THE TARGET CONFIGURATION FILE 4-6
4.7 TARGET CONFIGURATION FILES AND GNX TOOLS 4-7
4.8 USING THE GTS MENUS 4-9

Chapter 5 COMMAND SUMMARIES

5.1 INTRODUCTION 5-1
5.2 GTS 5-2
5.3 MINSTALL 5-3
5.4 MONFIX 5-4
5.5 NAR 5-6
5.6 NBURN 5-8
5.7 NCMP 5-13
5.8 NLORDER 5-14
5.9 NNM 5-15
5.10 NSIZE 5-18
5.11 NSTRIP 5-19

Appendix A GLOSSARY

INDEX

Chapter 1

OVERVIEW

1.1 INTRODUCTION

The GNX (GENIX Native and Cross-Support) language tools consist of a C compiler, a Pascal compiler, an assembler, a linker, debuggers, monitors, basic I/O routines, and other tools. The GNX language tools support the development of software for National Semiconductor's *Series 32000* microprocessor family. The GNX tools are cross-development tools. A program is developed and compiled on a host system, such as an IBM PC, and then downloaded to a *Series 32000* microprocessor-based system such as the FX164ED - Evaluation/Development board, an embedded target system, or a *Series 32000* In-System Emulator (ISE™) unit, for execution and debugging.

This manual describes the GNX software for UNIX and MS-DOS host development systems.

1.2 GNX LANGUAGE TOOLS

The GNX language tools include the following:

- Development Tools
  All GNX tools are distributed on magnetic media. See Table 1-1 for a complete list of GNX language tools.
- Language and Runtime Libraries
  The GNX libraries provide a complete runtime environment, including I/O math and full C support.
- Firmware Monitors
  The firmware monitors are distributed in source form. The monitor provides an essential service during the debugging of a program.
- Release Letter
  The release letter is an important document that should be read and retained for reference by all users of the GNX language tools. The release letter contains general information about the product, software installation instructions, known bugs, verification procedures, and supplemental information not found in the reference manuals.

Table 1-1. GNX Language Tools

GNX TOOLS	PURPOSE
nmcc	C Optimizing Compiler
nmpc*	Pascal Optimizing Compiler (Optional)
nasm	Assembler
sprof	Source level profiler
nmeld	Linker
nar	File archiver
nlorder	Sorts object files to make efficient libraries
dbug	Multi-window symbolic debugger**
dbg32*	Symbolic debugger
gts	GNX target setup utility
nburn	PROM programmer formatting utility
ncmp	Compares GNX binary files
nnm	Prints symbol table
nsize	Size of an object file
nstrip	Removes symbol table information
monitors	Source directories for all *Series 32000* monitors
minstall*	Installs GNX binary files
monfix*	A monitor fix utility
lib	Directories for the run-time libraries and indirectly executable programs
include	Header files

* Not provided in MS-DOS

** Multi-window environment is not supported in MS-DOS

1.3 INSTALLATION

Installation instructions for the software are provided in the release letters which accompany the GNX language tools.

1.4 MANUAL ORGANIZATION

Chapter 1 provides an overview of the GNX language tools.

Chapter 3 describes the GNX cross development methodology.

Chapter 4 describes GTS, a menu-driven configuration setup utility.

Chapter 5 describes the GNX command syntax and command line options.

1.5 OTHER MANUALS

Other GNX manuals of interest are listed below:

- C Optimizing Compiler Reference Manual.
  The C optimizing compiler generates high-quality code for the *Series 32000* architecture. This manual provides guidelines for using the optimizer, information regarding the compilation process, extensions to the C programming language and implementation issues.
- Pascal Optimizing Compiler Reference Manual.
  This manual provides guidelines for using the optimizer, information regarding the compilation process, extensions to the Pascal programming language and implementation issues.
- Assembler Reference Manual.
  The GNX Assembler is a support program that assembles *Series 32000* Assembly Language source programs and generates relocatable object modules. This manual describes the GNX Assembler in detail.
- Linker User's Guide.
  The GNX Linker is a tool used to link object files, produced either by the GNX Assembler and Compiler, or by the GNX Linker. The linker combines object files by resolving symbolic references, allocating output sections, and relocating memory addresses to produce an executable object file. This manual describes the GNX Linker in detail.
- Symbolic Debugger (DBUG) Reference Manual.
  DBUG is an interactive debugger that can debug programs developed with National Semiconductor's GNX Software Development Package. This manual describes the function, operation and use of the GNX Symbolic Debugger, DBUG.
- Support Libraries Reference Manual.
  Provides an overview of the GNX support libraries, including system calls and

emulation libraries.

- Language Tools Technical Notes.
  This manual describes the initialization of ROMable code on a *Series 32000*-based development system using the GNX (GENIX Native and Cross-Support) Version 4 development tools. This manual also contains a discussion of the GNX tools' support for modularity.

- COFF Programmer's Guide.
  This manual describes the GNX Language tools' implementation of the Common Object File Format (COFF) for *Series 32000* microprocessor-based systems, and serves as a "how to " guide for implementors of language tools.

Chapter 2

USING THE GNX LANGUAGE TOOLS

2.1 INTRODUCTION

This chapter is intended for new GNX software developers, as well as the infrequent user who needs a brief refresher. After reading it you should have a general knowledge of the available GNX tools, as well as the basic commands. Pointers to the more detailed manuals will be given for those who want more detailed information.

GNX is a *cross* support tool set, i.e., executable code is developed on a host system, such as an IBM PC, and not on a target system, such as a *Series 32000* development board. During the development cycle you convert a source program, written in C, Pascal, or even Assembly, to binary machine code that can be executed by a *Series 32000* Embedded Processor. The machine code can either be downloaded to a target board's RAM (for debugging), or programmed in ROM once the program has been verified.

The GNX software also includes a set of *monitors*, which are small programs that actually run on *Series 32000* development boards. The monitors' main task is to communicate with the GNX debugger that runs on the host computer during a debugging session.

The GNX language tools provide a complete environment in which to develop your application. All tools, except the monitor, are installed and executed on your host computer. Only your application and monitor run on the target system.

During the development process, your program can be in any of the following formats:

- Source file.
  This is usually in C, sometimes in *Series 32000* Assembly.

- Object file.
  This is a machine language representation of the assembly file, as produced by the GNX assembler, or directly by the compiler. It is *relocatable*, i.e., it does not yet contain the final mapping of the code to absolute addresses.

- Executable file.
  This is a file that can be executed directly by a *Series 32000* Embedded Processor. It is usually the result of combining several object files with mapping information, and is the product of the GNX linker. This file can be downloaded to a target board with help from the GNX debugger, or further processed and converted to PROM format accepted by most PROM programmers.

Your code may also be in the form of library files. These files are compiled and assembled as any other program, but instead of being linked together in an executable file, they are arranged into a library by the librarian tool. A library usually contains useful routines that are shared by many applications. When a library is linked together with other object files to create an executable file, only the library files that are actually used are extracted and are included in the final program. Thus the size of the executable file is minimized. Several useful libraries are supplied with the GNX language tools, see Chapter 3.

2.2 GNX UTILITIES

This section describes most of the components of the GNX language tools that are invoked at the command line.

2.2.1 C Optimizing Compiler

The GNX C Optimizing Compiler, `nmcc`, conforms to the full Kernigham and Ritchie C language definition, augmented with ANSI C prototypes, const and volatile type modifiers and other extensions. This compiler is a modular language processor consisting of five separate programs: a driver, a preprocessor (`cpp`), a compiler front end (`cc_fe`), a code generator (`cgen_coff`), and a global optimizer (`opt`). It can create object, executable, or assembly language code according to the compiler options specified by the user.

The optimizations are performed globally, by looking at the code for a whole procedure at a time, and not only in the local context of a line or loop. The Compiler is also aware of the special hardware support available on each CPU, and includes support for embedded programming and application-specific instructions. Thus, you can write even your most time-critical applications in C.

For detailed information on the GNX C Optimizing Compiler, refer to the *Series 32000 GNX–Version 4 C Optimizing Compiler Reference Manual.*

2.2.2 Pascal Optimizing Compiler

The GNX Pascal Optimizing Compiler, `nmpc`, (not supported on MS-DOS) supports standard Pascal, as defined by the International Standards Organization (ISO db7185 level 1), with a number of extensions. The extensions are a superset of the Berkeley UNIX "pc" compiler extensions.

The GNX Pascal Optimizing Compiler, `nmpc`, is a modular language processor consisting of five separate programs: a driver, a preprocessor (`cpp`), a compiler front end (`pas_fe`), a code generator (`cgen_coff`), and a global optimizer (`opt`). The compiler

can create object or assembly language code according to compiler options specified by the user.

For detailed information on the GNX Pascal Optimizing Compiler, refer to the *Series 32000 GNX–Version 4 Pascal Optimizing Compiler Reference Manual.*

2.2.3 Assembler

The GNX Assembler, `nasm`, assembles *Series 32000* assembly language source programs and generates relocatable object modules. Relocatable object modules must be linked to create executable load modules which may be run on a *Series 32000* microprocessor system. `nasm` is a full featured macro assembler that includes procedure definition, conditional statement generation, repetitive statement generation, and much more.

For detailed information on the GNX Assembler, refer to the *Series 32000 GNX–Version 4 Assembler Reference Manual.*

2.2.4 Linker

The GNX Linker, `nmeld`, creates executable files by combining object files, performing relocation and resolving external references. The linker also processes symbolic debugging information.

For detailed information on the GNX Linker, refer to the *Series 32000 GNX—Version 4 Linker User's Guide.*

2.2.5 DBUG - Multi-Window Symbolic Debugger

DBUG is the GNX symbolic debugger. DBUG provides a multi-window oriented user interface* and a rich command set.

DBUG provides the following features:

- Support for the HP64700 family of In-System Emulators for the *Series 32000* CPUs.
- Fast communication with development (or target) boards, and In-System Emulators, via Ethernet.
- Extensive breakpoint, trace and print capabilities.

- Command files and Log files.
- Symbolic disassembly.
- Function keys - any DBUG command can be attached to a function key*.
- Command history mechanism, as part of the human interface*.
- Command aliasing, to enable command invocation using abbreviations.

Features marked with a * are not supported in MS-DOS.

For more details see the *GNX Version 4 Symbolic Debugger (DBUG) Reference Manual.*

2.2.6 Source Level Profiler (sprof)

`sprof` is a unique code coverage profiler, supplying the programmer with statistics about the number of executions of each source line according to actual runs of the program. `sprof` output can be used to:

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

For more details see the *GNX Version 4 Compiler Reference Manuals.*

2.2.7 Nburn

This utility loads the specified bytes of an executable file to an EPROM burner in ASCII-HEX, Intel-hex, Extended Intel-hex or Motorola s-record format. See Section 5.6 for the synopsis and options of `nburn` and Section 3.7 for using `nburn`.

2.2.8 Nar Librarian

This utility maintains groups of files combined into a single archive file. `Nar` is used to create and update library files as used by the GNX linker `nmeld`. See Section 5.5 for the synopsis and options of `nar`.

2.2.9 Ncmp

`Ncmp` compares two *Series 32000* GNX binary files and prints the byte and line number at which a difference occurs (see Section 5.7).

2.2.10 Nlorder

`Nlorder` finds ordering relations for object libraries. The input may be one or more object or library archive (see `nar`) files. `Nlorder` writes to standard output a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second file. The output may be processed by `tsort` (if available on the host system) to find an ordering of a library suitable for one-pass access by `nlorder`. See Section 5.8 for the synopsis and options of `nlorder`.

2.2.11 Nnm

The `nnm` utility displays the symbol table of a *Series 32000* GNX object file. See Section 5.10 for the synopsis and options of `nnm`.

2.2.12 Nsize

`Nsize` displays the size information for each section and optional header information of a *Series 32000* GNX object file. See Section 5.11 for the synopsis and options of `nsize`.

2.2.13 Nstrip

`Nstrip` strips symbol and line number information from a *Series 32000* GNX object file. See Section 5.12 for the synopsis and options of `nstrip`.

2.3 RECOMMENDED INSTALLATION

The following sections describe the default arrangement of the GNX language tools on your host system. The term `GNXDIR` refers to the home directory in which the GNX tools are installed. For example, `GNXDIR` could be `C:\gnx4` in a typical MS-DOS installation.

All directly executable programs (`nmcc`, `nmeld`, `nar`, etc.) reside in the `GNXDIR` directory. Include files reside in the `GNXDIR/include` (`GNXDIR\include` on MS-DOS) directory and its subdirectories. Libraries and indirectly executable programs (the C preprocessor, `cpp`, etc.) reside in the `GNXDIR/lib` (`GNXDIR\lib` on MS-DOS) directory.

The tools normally search GNXDIR, GNXDIR/lib for the appropriate files. This normal search pattern can be altered by setting the environment variables TMPDIR, CMDDIR, LIBPATH, and INCLUDEPATH.

The environment variable TMPDIR redefines the location at which temporary files are created in the compilation process (UNIX default: /tmp; MS-DOS default: current directory). This environment variable should be used with care, especially on small systems with tiny partitions, which overflow when compiling huge files.

The environment variables CMDDIR, LIBPATH, and INCLUDEPATH are described in the following sections.

2.3.1 Directly Executable Programs

For ease of access to the directly executable programs, it is recommended that GNXDIR be added to the PATH variable.

2.3.2 Indirectly Executable Programs

Most of the GNX language tools and optional compilers call other programs to perform distinct phases of the work in hand. For example, the C compiler (nmcc) calls the C preprocessor (cpp), the C compiler front end (cc_fe), the code generator (cgen_coff), the assembler (nasm), and finally the linker (nmeld).

To find the directory of the indirectly executable programs, each tool determines its home directory and appends /lib (\lib on MS-DOS). For example, if one executes a tool such as nmcc residing in directory /NSC, nmcc searches /NSC/lib for cpp. This search directory can be changed by setting an environment variable CMDDIR. For example, if the CMDDIR environment variable is set to /newNSC/lib, nmcc uses /newNSC/lib/cpp as the C preprocessor.

2.3.3 Include Files

By default, the compilers search for the include files in the GNXDIR/include directory (*e.g.*, /usr/NSC/include, if the root is /usr/NSC).

The search path can be changed by defining an environment variable INCLUDEPATH which contains the list of directories that are used to search for the include files. Thus, if INCLUDEPATH is set to /usr/sysV/include:/newNSC/include, these two directories will be used to search for the include files instead of GNXDIR/include.

NOTE: The C compiler provides a `-I` option to specify a directory containing include files. If the `-I` option is used, the specified directory is searched before the directories in `GNXDIR/include` or those specified by the INCLUDEPATH variable.

2.3.4 Libraries

Normally the libraries are located in the directory `GNXDIR/lib`. The linker (`nmeld`) looks for the libraries in `GNXDIR/lib`.

In addition to libraries, `GNXDIR/lib` contains the start-up object files (`crt0.o`, etc.) needed by the C (`nmcc`), and Pascal (`nmpc`) compilers, if the compilers are installed.

The library search path can be changed by setting an environment variable LIBPATH to a list of directories which should be searched for libraries. If the LIBPATH environment variable is set to `/NSC/dblib:/newNSC/newlib`, these directories are searched for libraries and start-up code instead of `GNXDIR/lib`. If the LIBPATH environment variable contains more than one directory name, the start-up file `crt0.o` must reside in the first directory in the list.

NOTE: The linker provides a `-L` option which specifies a directory containing libraries. If this option is used, the specified directory is searched before `GNXDIR/lib` or the LIBPATH variable.

If you want a mixture of standard and nonstandard libraries, set the LIBPATH variable, and make sure that the nonstandard library does not contain files with the same names as files in the standard library. If greater flexibility is required, you may have to copy or link standard libraries to nonstandard locations.

Chapter 3

CROSS DEVELOPMENT

3.1 INTRODUCTION

This chapter will explain the process of generating code suitable for execution on a target board. It will show you how to invoke the compiler, how to select the proper libraries and use the linker, how to use `dbug` as a debugger, and finally, how to use `nburn` to create binary files for the PROM programmer.

3.2 SOME BASIC CONCEPTS

The process of creating an executable file, loading it to a target board, and debugging it, depends on four special files, in addition to library files and your normal application files. These are the target configuration file, the linker definition file, the start-up object file (`crt0.o`), and the DBUG configuration file.

The target configuration file controls several default parameters in the development process, such parameters as the CPU you are using, the FPU (if any), the default bus width of the system, etc. This file also contains data relevant to the linking and debugging phases.

You can create a target configuration file by using the GNX Target Setup program (GTS). For more information about the target configuration file and GTS, see chapter 4 in this manual.

The linker definition file is where you specify the physical target memory configuration; RAM and ROM address ranges, initial stack pointer value, etc. This file determines the actual mapping of your application to memory during the linking process. You specify the name of the linker definiton file either in the command line when invoking the linker, or in the target configuration file. Note that if you do not specify a linker definition file in either place, a default linker definition file, placed in GNXDIR, will be used (`linker.def`).

For more information on the linker definition file, see the *GNX Version 4 Linker User's Guide.*

The start-up object file is usually linked together with your application. The main purpose of the GNX default file, `crt0.o`, is to interface between your application and the debugger. For example, it sets the stack pointer to the value defined in the linker definition file and contains the default start-up entry-point (`start:`). The default start-up code, `crt0.o`, is linked if the linker is called indirectly by the compiler. If you

perform the linking by explicitly invoking the linker, the start-up code must be specified. You may use your own start-up code if you prefer.

The DBUG configuration file can be used to set basic debugger parameters, such as the name of the I/O port used for debugging, the baud rate, and other parameters which can override the setting in the `.gnxrc` file. This file can also contain definitions of aliased commands, and can contain a list of debugger commands that you want to execute whenever the debugger is invoked.

The DBUG configuration file is local to each directory, and if it does not exist, the debugger will use information from the target configuration file; the name of the CPU, the name of the monitor, etc. Other parameters, such as the name of the I/O port, can be specified on the command line, or from inside the debugger.

3.3 GNX SUPPORT LIBRARIES

The GNX libraries support application development in C and PASCAL. In addition, these libraries facilitate debugging of user programs by providing input and output capability with the user terminal or host filesystem. Programs linked with these libraries can run on a development board or an in-system emulator.

The libraries are located in the `GNXDIR/lib` directory in which the GNX tools are installed. Table 3-1 lists each library and its description. Note that there is a set of libraries for generating non-modular code, a set for producing modular code and a set for producing high speed fp emulation (Hfp) code. Refer to the *Series 32000 GNX–Version 4 Language Tools Technical Notes* for additional information on modularity, and to *Series 32000 GNX–Version 4 Support Libraries Reference Manual Chapter 6* for more information on using the high speed fp emulation library.

Table 3-1. Libraries

NON-MODULAR CODE LIBRARIES	MODULAR CODE LIBRARIES	HFP CODE LIBRARIES	DESCRIPTION
libc.a	libXc.a	libHc.a	C library
libctp.a	libXctp.a	libHctp.a	Library containing compiler specific routines
libg.a	libXg.a	libHg.a	Library for C and FORTRAN when compiled with the –g option
libpas.a	libXpas.a	libHpas.a	Pascal interface library
libm.a	libXm.a	libHm.a	Math library
lib381m.a	libX381m.a		Support library for FPU
		libHfp.a	High speed fp emulation library
libfpe.a	libXfpe.a		Floating-point emulation library

C programs must always link with `libc.a` and `libctp.a`, and they need `libm.a` if using any math functions. Pascal programs need `libpas.a`, `libm.a`, and `libc.a` in that order. In addition to the libraries, C and PASCAL programs must link with start-up and run-time initialization code for debugging, such as the start-up object file `crt0.o` (`Xcrt0.o` for modular code) located in the `GNXDIR/lib` directory, or an equivalent startup code.

The math library, `libm.a`, contains standard math functions, such as sin(), log(), etc. The `lib381m.a` supports the NS32081 and NS32181 floating-point unit, for full IEEE compatibility.

The `libHfp.a` is a high speed floating point emulation package that can replace the floating point unit.

The `libfpe.a` is an alternate floating point emulation package, that uses the *Series 32000* trap mechainsm. Code compiled for an FPU, will also run without an

FPU, if this library is linked.

For more information on the GNX Libraries, refer to the *Series 32000 GNX—Version 4 Support Libraries Reference Manual.*

3.4 MONITORS

This section gives a brief description of the different monitors available with the GNX language tools. For GNX software tools to work with any *Series 32000* development board, one of the monitors must be installed on the board.

The GNX monitors are PROM-based firmware monitors for use on a *Series 32000* development board. These monitors allow you to load, execute, and debug development board programs with the DBUG debugger running on a host computer system. Program execution and debugging are controlled by breakpoints. At breakpoints, the user can display and change the contents of memory, internal CPU registers, and Special Purpose and Slave Processor registers.

The monitors also provide run-time services such as physical I/O, interrupt handling, error handling, and virtual I/O. Services are available in the form of supervisor calls.

The GNX release includes the monitors' sources to enable you to tailor a monitor to a specific target system.

For detailed information on the monitors, refer to the *Series 32000 Development Board Monitor Reference Manual.*

3.5 COMPILING A PROGRAM

This section describes the basic steps required to develop an executable file suitable for debugging with `dbug`. To compile the C files `file.c` and `file1.c`, enter the following:

```
nmcc -c file.c file1.c -g
```

Two relocatable files, `file.o` and `file1.o` will be created. The target configuration file determines the CPU and FPU (if any) for which to generate the code. The `-g` flag indicates that symbolic information for debugging is to be produced.

Linking, then, may be performed by entering:

```
nmcc file.o file1.o -o binprog
```

The compiler, in this example, calls the linker with the required libraries, specifies the start-up file, and uses the linker definition file as set by the target configuration file. The `-o` flag indicates the name of the generated executable file.
The same result can be achieved by invoking the linker explicitly:

```
nmeld CRT0 file.o file1.o -lctp -lc -d link.def -m -o binprog
```

where *CRT0* is the full pathname of the `crt0.o` start-up file, and link.def is the name of the linker definition file. The `-m` flag indicates that a memory map is to be generated on the standard output.

If math functions are used, the math library must be specified during linking, as follows:

```
nmcc -c file.o file1.o -lm -o binprog
```

The executable file just generated, `binprog`, will be ready to run on a *Series 32000* development board.

3.6 A SAMPLE DEBUG SESSION

This section assumes that a *Series 32000* development board is installed to operate in stand-aside mode, and is connected to serial port `ttya` of your system. The target configuration file should indicate that the communication will be via the serial port. The sample session shows the interaction between you and the debugger when downloading a program (`binprog`) to a development board, and using `dbug` for debugging. For a detailed description of `dbug`, refer to the *Series 32000 GNX – Version 4 Symbolic Debugger (dbug) Reference Manual.*

1. Connect the board to `ttya`
2. Press the reset switch on the board.
3. Invoke the GNX Debugger (`dbug`)

   ```
   % dbug binprog
   ```

   The following is a log of a debug session:

```
Dbug - Version 4.4
Type 'help' for help
connecting...
connection with ttya established
setup in remote mode
reading symbolic information ...
load with sp 0x100000
loading.............
loaded 8964 bytes of code, 3708 bytes of data
total of 12672 bytes_loaded
(dbug)  list 1
    1   main()
    2   {
    3           printf("hello");
    4   }
(dbug)  stop at 3
[1] stop at "myprog.c":3
run
[1] stopped in main at line 3 in file "myprog.c"
    3           printf("hello");
(dbug)  pcpu
        CPU REGISTERS
$r0 = 0xffffffff   $r1 = 0x00000000   $r2 = 0x00000006
$r3 = 0x00011e64   $r4 = 0x00011e64   $r5 = 0x00000000
$r6 = 0x00000000   $r7 = 0x00000000   $fp = 0x000fffcc
$sp = 0x000fffc8   $pc = 0x0000e179   $psr = 0x0340
$mod = 0xe000      $sb = 0x00011e7c   $is = 0x00000400
$us = 0x000fffc8   $in = 0x00000480   $cfg = 0x00000000
(dbug)  cont
hello
execution completed
(dbug)  quit with save
saved session on file dbug.save
```

NOTE: **The size of the object file and the address of variables and program statements may differ depending on the version of tools.**

3.7 USING NBURN

The GNX linker produces output files that are loadable by the GNX debuggers and kernels of *Series 32000*-based development systems. A program can also be executed by using the `nburn` utility to transfer the program into PROMs.

`Nburn` is a program that converts the text and data sections of a GNX executable file into ASCII-HEX, Intel-hex or Motorola s-record format, which can be down-loaded to a PROM programming device. Output from `nburn` can either be sent directly to a PROM programmer connected to the auxiliary port of the user terminal or stored in a file to be down-loaded later. Each session of the `nburn` program results in a data stream or file containing data for one or more PROMs.

For `nburn` to determine how a GNX program should be converted, several parameters may be specified, among them are buswidth (–w), byte number (–b), and PROM size (–l). PROMs are generally byte wide. For a system with a 16-bit bus (or 2-byte wide bus), assuming no dynamic bus sizing, data from an executable file has to be interleaved into 2 blank PROMs (*i.e.*, byte 0 goes into PROM 0, byte 1 goes into PROM 1, byte 2 goes into PROM 0, etc.) In this case, `nburn` reads text or data from the file and puts out every other byte to the port or output file. For a set of 2764 PROMs (which is 8 Kbytes in length), the byte number 0 PROM contains the first 8192 of the even bytes of the file (*i.e.*, address 0,2,4 ... up to address 16382), assuming that the program is linked to start from location 0 and that the first byte of the program goes to the first location in the PROM. The byte number 1 PROM contains the first 8192 of the odd bytes of the file (*i.e.*, addresses 1,3,5 ... up to address 16383). If the file is too big to fit into the two PROMs, the next set of PROMs will contain data starting from address 16384. `nburn` can also automatically produce output for the next set of PROMs. For example, if a file contains the following data starting from address 0:

```
ea c0 e0 00 05 7d a1 00 09 74 a5 01 c0 f0 00 80
9a 87 5b 5f 38 27 d8 c0 00 00 09 ea 89 0a 67 d8
```

For a buswidth of 4 bytes, the PROM for byte number 0 contains the following bytes from the file:

```
ea 05 09 c0 9a 38 00 89
```

and the PROM for byte number 3 contains:

```
00 00 01 80 5f c0 ea d8
```

Other `nburn` options may be used to specify the start address of PROMs, the number of banks to be selected and other parameters. See Section 5.6 for a detailed description of the `nburn` options.

Chapter 4

GNX TARGET SETUP

4.1 INTRODUCTION

The GNX language tools support several target setup combinations, which can be specified as invocation options. The GNX Target Setup (GTS) utility allows for a one-time target setup specification. This specification can be used afterwards by all GNX tools.

4.2 GNX TARGET SETUP (GTS)

GNX Target Setup (GTS) is an editor that generates a user's local target configuration file which is read by the GNX tools when invoked.

Some GNX applications may have configuration parameter-dependent source code that needs to be conditionally compiled. To accommodate these types of applications, GTS offers the option to generate a C-style include file named `gnxenv.h`. This feature allows the GNX user access to the target configuration parameters at the same point that the parameters are set.

GTS operates in two modes, Interactive or Non-interactive.

4.2.1 Interactive Mode

In Interactive mode, GTS requires user participation. The target configuration file is generated from the user's input. Interactive mode consists of a main menu and sub menus, one for each target parameter. Section 4.8 describes these menus and their use.

4.2.2 Non-Interactive Mode

In Non-interactive mode, GTS generates the local target configuration file from a source file instead of from the user's input. The current target configuration file is first copied to the file specified by the GTSBACKUP parameter or to `gnxrc.bak`, if GTSBACKUP is not set. Then the source file is copied to the local target configuration file in the user's home directory. Non-interactive mode requires that GTS is invoked with the –f option.

4.3 TARGET CONFIGURATION FILE CONTENTS

The target configuration file contains the target parameters and the user-chosen value for each parameter.

4.3.1 Description of Parameters

The parameters are individual factors which affect the behavior of the GNX tools. Table 4-1 lists these parameters and their meanings.

Table 4-1. Target Parameters

PARAMETER	DESCRIPTION
OS:	Specifies which monitor or operating system is the executor of the code that the GNX tool is compiling or debugging.
CPU:	Specifies which *Series 32000* CPU's are on the target.
MMU:	Specifies which, if any, of National's memory management units is on the target.
FPU:	Specifies which, if any, of National's floating-point units is on the target.
COMMTYPE:	Specifies the protocol to be used in communicating with the target.
BUSWIDTH:	Specifies the data buswidth of the CPU.
COMMPORT:	Specifies to which port the target is connected.
PREDEFS:	Specifies which symbols to predefine in the C preprocessor.
LINKERFILE:	Specifies a linker directives file to use instead of the default.
GTS_BACKUP:	Specifies the backup filename of the local target configuration file.

4.3.2 Choices

Each legal selection of a parameter is called a "choice." Table 4-2 lists the parameters and their legal choices.

The parameters, whose choice is a *string*, are supplied by the user. The GNX utilities/tools check for syntactic and semantic correctness of the strings, GTS does not.

Table 4-2. Choices to the Parameters
Sheet 1 of 2

PARAMETER	CHOICES	MEANING
OS	SYS5_2	Target is System 5, Release 2, OS
	SYS5_3	Target is System 5, Release 3, OS
	DBMON	Target is a db/edb with its monitor
	MON16, MON32	"
	MON332, MON332B	"
	MON532, MONGX32	"
	MONCG16, MONGX32E	"
	MONCG160, MONGX320	"
	MONAM160, MONFX16	"
	MONFX164, MONCG160LX	"
	ISEGX320, ISECG16	ISE with an HP64772/8/9 emulator
	ISEFX16, ISEFX164	"
	ISECG160, ISE532	"
	SPMON	Target is Splice with a splice monitor
	OTHER_OS	Target is a non-National OS
CPU	NS32008	CPU is an NS32008.
	NS32016	CPU is an NS32016.
	NS32032	CPU is an NS32032.
	NS32332	CPU is an NS32332.
	NS32532	CPU is an NS32532.
	NS32CG16	CPU is an NS32CG16.
	NS32GX32	CPU is an NS32GX32.
	NS32CG160	CPU is an NS32CG160.
	NS32GX320	CPU is an NS32GX320.
	NS32FX16	CPU is an NS32FX16.
	NS32AM160	CPU is an NS32AM160.
	NS32FX164	CPU is an NS32FX164.
MMU	NOMMU	No MMU is on-board.
	NS32082	MMU is an NS32082.
	NS32382	MMU is an NS32382.
	ONCHIP	MMU is on the CPU chip.
FPU	NOFPU	No FPU is on-board.
	NS32081	FPU is an NS32081.
	NS32181	FPU is an NS32181.
	NS32381	FPU is an NS32381.
	NS32580	FPU is an NS32580.
	EMULATION	Use libHfp emulation.

Table 4-2. Choices to the Parameters
Sheet 2 of 2

PARAMETER	CHOICES	MEANING
COMMTYPE	NOCOMMTYPE	No communication type.
	SERIALA	Debuggers use pre-GNX R2 Rev. C protocol and RS232 line to down-load.
	SERIALB	Debuggers use post-GNX R2 Rev. C protocol and RS232 line to down-load.
	ETHERNET	Debuggers use ethernet to down-load.
COMMPORT	*string*	Target is connected to this port, or target Ethernet ID.
PREDEFS	*string*	Predefine these macros to the C preprocessor.
BUSWIDTH	1	Data bus width is 1 byte.
	2	Data bus width is 2 bytes.
	4	Data bus width is 4 bytes.
LINKERFILE	*string*	Linker directives file to use instead of the default.
GTS_BACKUP	*string*	Filename of target configuration file backups.

4.3.3 Syntax of Target Configuration File

Each line of the target configuration file consists of a parameter and a choice. Each parameter and choice is separated by an "=" equal sign. White space is ignored. Parameters and choice identifiers are case insensitive, except where the value is a string.

Depending on the parameter, the string may be a filename, a pathname, etc. Table 4-3 summarizes the legal strings.

Table 4-3. Legal Strings

STRING	DESCRIPTION
COMMPORT	The *string* must be a valid tty port, or target Ethernet ID. The actual syntax is operating system dependent.
PREDEFS	The *string* is an options string to the C preprocessor. The options string resembles a `cpp` command, except that only the -D, -U, and -I flags are accepted.
LINKERFILE	The *string* is a linker directives file. The actual syntax is operating system dependent.
GTS_BACKUP	The *string* is a valid filename. The actual syntax is operating system dependent.

The following is an example of the contents of a target configuration file:

Example:

```
OS           =  monGX320
CPU          =  NS32GX320
MMU          =  nommu
FPU          =  NS32381
COMMTYPE     =  ETHERNET
BUSWIDTH     =  4
COMMPORT     =  node14
LINKERFILE   =  ldfile
PREDEFS      =  -DHELLO -DHOST_IS_V
GTS_BACKUP   =  mygnxrc.bak
```

`node14` is the Ethernet address to which the development board is connected.

4.4 TARGET CONFIGURATION FILE LOCATION

The GNX tools look for target configuration files in three standard locations, in the following order:

1. The global target configuration file.
   This file is installed by the installation procedure in `GNXDIR`. On UNIX, `GNXDIR/.gnxrc`, on MS-DOS `GNXDIR\gnx.ini`. The global target configuration file should not be modified, except by the system administrator.

2. The user's target configuration file.
   If it exists, the user's target configuration file overwrites the configuration values of the global target configuration file. The user's target configuration file is located in `$home/.gnxrc` on UNIX. On MS-DOS, you must define an environment variable, `HOME` that contains the path of your home directory, e.g.,

   ```
   SET HOME = D:\myhome
   ```

   The user's target configuration file will then be expected in :

   ```
   D:\myhome\gnx.ini
   ```

3. The local target configuration file.
   This file is located in the current directory. On UNIX, this is `./.gnxrc`, on MS-DOS `.\gnx.ini`. If it exists, the local target configuration file overwrites the configuration values of the global and user's target configuration files.

4.5 GNXENV.H

The C-style include file `gnxenv.h` contains only target configuration parameters whose choice is not *string*. This file is optional and is intended to provide source-code-level control of the target specification. The target configuration macros set up in this file are accessed via standard `cpp` directives.

The `gnxenv.h` file is created by answering `y` to the questions asked when exiting GTS. Table 4-4 shows a version of the `gnxenv.h` file for the example in Section 4.3.3.

4.6 OVERRIDING THE TARGET CONFIGURATION FILE

The target specifications in the configuration file can be overridden by using the existing flags that each GNX utility supports. For example, if DBUG is invoked with the `-mmu=382 flag`, the existing target MMU specification parameter in the configuration file is overridden.

```
/*
 * gnxenv.h
 * National Semiconductor Corporation
 * This file was generated by Get GNX Target (gts)
 * on 5/1/90 at 11:42:25
 */
#define SYS5_2                  10
#define SYS5_3                  11
#define DBMON                   12
#define MON16                   13
#define MONCG16                 14
#define MONCG160                15
#define MON32                   16
#define MON332                  17
#define MON332B                 18
#define MON532                  19
#define ISE532                  20
#define MONGX32                 21
#define MONGX320                22
#define MONGX32E                23
#define SPMON                   24
#define OTHER_OS                25
#define OS                      MONGX320
#define NS32008                 30
#define NS32016                 31
#define NS32CG16                32
#define NS32FX16                33
#define NS32CG160               34
#define NS32032                 35
#define NS32332                 36
#define NS32532                 37
#define NS32GX32                38
#define NS32GX320               39
#define CPU                     NS32GX320
#define NOMMU                   50
#define NS32082                 51
#define NS32382                 52
#define ONCHIP                  53
#define MMU                     NOMMU
#define NOFPU                   60
#define NS32081                 61
#define NS32181                 62
#define NS32381                 63
#define NS32580                 64
#define EMULATION               65
#define FPU                     NS32381
#define NOCOMMTYPE              70
#define SERIALA                 71
#define SERIALB                 72
#define ETHERNET                73
#define COMMTYPE                ETHERNET
#define BUSWIDTH                4
```

4.7 TARGET CONFIGURATION FILES AND GNX TOOLS

Table 4-4 demonstrates how each target specification parameter affects the behavior of the GNX tools.

Table 4-4. Parameters and GNX Tools

GNX TOOL	PARAMETER	RESULT
C preprocessor	PREDEFS	Predefines PREDEFS in the C preprocessor.
compiler	CPU	Performs optimizations for the specified CPU.
	BUSWIDTH	Performs alignment and packing optimizations for the specified BUS.
	FPU	Performs floating-point optimizations for the specified FPU.
assembler	CPU	Accepts instructions for the specified CPU only.
	MMU	Accepts instructions for the specified MMU only.
	FPU	Accepts instructions for the specified FPU only.
linker	LINKERFILE	Gets linker directives from `linkerfile`.
debugger	OS	The specified operating system is the executor of the program.
	CPU	The specified CPU is on-board the target.
	FPU	The specified FPU is on-board the target.
	MMU	The specified MMU is on-board the target.
	COMMTYPE	Down-loads to the target using COMMTYPE protocol.
	COMMPORT	The target is located at the tty port COMMPORT.

4.8 USING THE GTS MENUS

The GNX Target Setup (GTS) menu-driven screens are easy to use. The up/down arrows move to the appropriate selection and depressing the <CR> makes that selection. The characters enclosed in "( )" parentheses are called shortcut keys and eliminate excessive keystrokes. Depressing a shortcut key makes a selection automatically (*e.g.*, depressing a **g** on the MAIN MENU moves to the SELECT FPU menu). When each menu is displayed, the instructions and the first selection are highlighted.

There are different types of menus: a command menu, SELECT menus, SET menus, HELP menus, and REVIEW menus.

The MAIN MENU is the only command menu. From the MAIN MENU, each selection is a command that results in a sub-menu.

Each SELECT menus present a target parameter with all of its choices. The user is allowed to "select" one of the legal choices for the value of the target parameter. For example, the OS MENU sets the operating system target. Each SELECT menu indicates its current setup with two asterisks (**).

SET menus correspond to target parameters that accept strings. The commands of the SET menus are: OK, REVIEW SETUP, HELP, CANCEL, DELETE, and ENTER MODE. OK means that the string entered is acceptable and the target parameter corresponding to this menu should be saved. REVIEW SETUP displays the current target configuration for review. HELP provides a brief explanation and examples of the string. CANCEL exits the SET menu, without saving the changes. DELETE deletes the current input line. ENTER MODE allows the user to begin entering or modifying the input string. When a SET menu is selected, ENTER MODE is the current state; a <CR> exits ENTER MODE.

The REVIEW menu is accessed from the MAIN MENU, each SELECT menu, and each SET menu. Depressing any key returns the user to the previous menu.

When target specifications are set and the EXIT command of the main menu is issued, GTS checks for illegal target configuration combinations. If it finds any, it issues a warning and allows the user the opportunity to remedy the illegal combination. Following the combination check, GTS inquires whether or not to generate the file `gnxenv.h`. Finally, it inquires whether or not the changes should be saved, and then exits.

Chapter 5

COMMAND SUMMARIES

5.1 INTRODUCTION

This chapter summarizes the commands and the use of command line options for the GNX software tools running on a UNIX or MS-DOS operating system.

The following is a list of the commands discussed in this chapter:

`gts`	GNX Target Setup
`minstall`	Installs GNX binary files (UNIX only)
`monfix`	A monitor maker (UNIX only)
`nar`	Archive and library maintainer
`nburn`	Generates formatted data streams
`ncmp`	Compares two binary files
`nlorder`	Finds ordering relocation for an object library
`nnm`	Prints name list of *Series 32000* GNX object files
`nsize`	Prints section sizes of object files
`nstrip`	Strips symbol and line number information

For informatiom on `dbug`, `nasm`, `nmcc`, `nmpc`, `nmeld` and `sprof`, see the appropriate manuals. (`sprof` is described in the *C Compiler* manual.)

5.2 GTS

`gts` – constructs a target configuration file

USAGE

`gts` [**-f**]

DESCRIPTION

`Gts` is a menu-driven program which takes the user's target specification and constructs either a user's target configuration file, or a local target configuration file, which is read by each of the GNX tools when invoked. The user's target specification is written to the file `.gnxrc` (`gnx.ini` on MS-DOS) in the user's home directory. The local target specification is written to the file `./.gnxrc` (`.\gnx.ini` on MS-DOS). For futher details of the target configuration files and their locations, see Section 4.4.

Some GNX applications may have configuration-parameter-dependent source code that may need to be conditionally compiled. To accommodate these types of applications, `gts` offers the option to generate a C-style include file named `gnxenv.h`. This feature offers the GNX user access to the target configuration parameters at the same point that the parameters are set.

The target configuration file consists of the GNX target parameters and their user-chosen values.

A detailed description of the target parameters and their choices is found in Chapter 4.

`-f` Generates the local target configuration file from a source file (Non-interactive mode). Default is from user input (Interactive mode).

FILES

`.gnxrc`	(`gnx.ini`)	local target configuration file
`$HOME/.gnxrc`	(`%HOME%\gnx.ini`)	private target configuration file
`GNXDIR/.gnxrc`	(`%GNXDIR%\gnx.ini`)	default target configuration file

Files in brackets are for MS-DOS.

5.3 MINSTALL

`minstall` – installs GNX binary files

USAGE

`minstall` [*option*] *filename* [*filename*] *destination*

DESCRIPTION

The *binary* GNX file is copied (or moved if the –m option is specified) to *destination*. If *destination* already exists, it is removed before *binary* is copied (or moved). If the destination is a directory, then the *binary* GNX file is copied (or moved) into the destination directory with its original filename.

The mode for the *destination* is set to 0664.

The available options are

- **`-m`** move file instead of copying.
- **`-s`** strip symbols from the destination file.
- **`-x`** remove source file's extension.
- **`-v`** prints out the version number of `minstall`.

Note: This command is not available under MS-DOS.

5.4 MONFIX

`monfix` – a monitor maker

USAGE

`monfix` [*option*] *filename*

DESCRIPTION

`Monfix` modifies the first 16 bytes of the text segment of *filename* for making a *Series 32000*-based board monitor. *Filename* must be an executable file in GNX format.

`Monfix` makes the first text byte the entry point because any *Series 32000* board bootstrap program requires that upon reset or power-up the first text byte be the entry point.

`Monfix` adds three instructions to the file: Load Module register, Load Static Base register, and Branch to the actual entry point. The values for these three addresses are computed from the `a32.out` header and module table entries but can be overridden by user-supplied values. Instructions generated and starting addresses are as follows:

address	instruction	
0	lprw	mod, $modvalue
4	lprd	sb, $sbvalue
10	br	entrypoint

`-b` *number*
: Uses *number* as the address where the program loads. Defaults to zero.

`-e` *number*
: Real entry point is at address *number*. `Monfix` uses *number* as the parameter for the `br` (third) instruction.

`-m` *number*
: Initial MOD register value is *number*. `Monfix` uses *number* as the parameter for the `lprw` (first) instruction.

-s *number*
Initial static base register value is *number*. `Monfix` uses *number* as the parameter for the `lprd` (second) instruction.

-v Prints out the version number of `monfix`.

All *numbers* are hexadecimal.

The default *filename* is `a32.out`.

Note: This command is not available under MS-DOS.

5.5 NAR

nar – archive and library maintainer for portable archives

USAGE

nar *key* [*modifier*] [*posname*] *afile filename* ...

DESCRIPTION

The nar command maintains groups of files combined into a single archive file. Its main use is to create and update library files for use by the linker. It can be used, though, for any similar purpose. The magic string and the file headers used by nar consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

When nar creates an archive, it creates headers in a format that is portable across all machines running the GNX language tools. The archive symbol directory is used by the linker nmeld to issue multiple passes over libraries of object files in an efficient manner. An archive symbol directory is created and maintained by nar only when there is at least one object file in the archive. This file is neither mentioned to nor accessible to the user. Whenever the nar command is used to create or update the contents of such an archive, the symbol directory is rebuilt. The s option described below forces the symbol directory to be rebuilt.

The minus "–" is optional to *key*, followed by one character from the set d, r, q, t, p, V, m, or x, optionally concatenated with one or more modifiers from the set v, u, a, i, b, c, l and/or s. *Posname* is the reference file used by the positioning characters a, b, or i; *afile* is the archive file. The *filenames* are constituent files in the archive file. The meanings of the *key* characters are as follows:

- **d** Deletes the named files from the archive file.
- **r** Replaces the named files in the archive file. If the optional character u is used with r, only those files with dates of modification later than the archive files are replaced. If the archive does not already exist, it is created. If the file is not already present in the archive, it is placed at the end. An optional positioning character a, b, or i may be specified along with a *posname*, which allows the user to determine the placement of new files within the archive.
- **q** Quickly appends the named files to the end of the archive file. Optional positioning characters are invalid. This command does not check whether the added members are already in the archive.

t Prints a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Prints the named files in the archive.

m Moves the named files to the end of the archive. If a positioning character is present, the *posname* argument must be specified.

x Extracts the named files. If no names are given, all files in the archive are extracted. In neither case does `x` alter the archive file.

v Prints the version number of the `nar` program being used.

The meanings of the modifier characters are as follows:

v Gives a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with `t`, gives a long listing of all information about the files. When used with `x`, precedes each file with a name.

c Suppresses the message that is produced by default when *afile* is created.

l Places temporary files in the local current working directory, rather than in the directory specified by the environment variable TMPDIR or in the default directory `/tmp`.

n Suppresses generation of symbol directory, thus shortening nar processing time. This command is useful when incrementally creating large libraries by several calls to `nar`. When used, all calls should be made with the `-n` option, except for the last call.

s Forces the regeneration of the archive symbol directory, even if `nar` is not invoked with an option which modifies the archive contents. This command is useful to restore the archive symbol directory of an archive created using the `-n` option, or an archive whose symbol table has been removed using the `nstrip` utility.

a,**b**, or **i**
Optional character positioning characters. When used, the *posname* argument is required and specifies that new files are to be placed after `a` or before `b` or `i` *posname*.

MS-DOS Additional Options:
@filename reads `nar` options from file. The **@** option directs the GNX Version 4 archiver to read additional options from the named file. This option avoids the MS-DOS limitation on the length of invocation lines, and enables passing options of unlimited length.

5.6 NBURN

nburn – generates formatted data for an EPROM programmer

USAGE

nburn [*options*] [*filename*]

DESCRIPTION

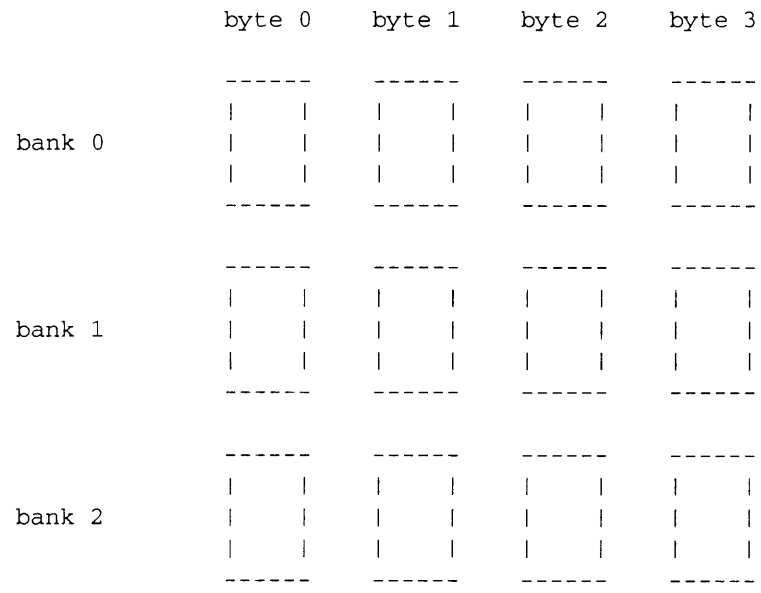
filename is an **nburn** input file. The input file for **nburn** should be a GNX executable object file produced by the GNX linker. The default input file name is `a32.out`.

The **nburn** utility is used to burn EPROMs by converting data from GNX executable object files into an EPROM programmer format. Three formats are supported: ASCII-hex (default), Intel-hex and Motorola. The output is directed into separate files, one for each EPROM. If no output file name is specified, the output is directed to the auxiliary (printer) port of a VT100 compatible terminal; the EPROM programmer must then be attached to the auxiliary port.

Each EPROM holds part of the data bytes of a memory block, depending on the system bus-width (or word size). If the system bus_width is 2, the EPROMs come in pairs. In each pair one EPROM holds the even address data bytes and the other EPROM holds the odd address data bytes. Together they form a bank. If the system bus-width is 4, the EPROMs come in quartets: one EPROM holds the first data bytes of each word, another EPROM holds the second data byte of each word and so on. These 4 EPROMs again form a bank.

The **nburn** utility allows you to create output for one or more EPROMs in various ways. The ROM area is regarded by the **nburn** utility as a two-dimensional matrix of EPROMs. One dimension is the EPROM byte number (i.e. which byte in the word the EPROM holds). The other dimension is the bank number, since the **nburn** utility can produce output for several consecutive banks.

The figure below illustrates a two-dimensional matrix of EPROMs:



Using the **nburn** utility you can both specify each EPROM in the matrix, and specify each row or column of EPROMs. It is also possible to specify the complete matrix in order to produce output for the entire ROM area in a single invocation of **nburn**. If the output is directed to a file(s), **nburn** will create one output file for each EPROM.

The **nburn** invocation options listed below. All integer constants can be specified in C syntax (i.e. decimal, hexadecimal, or octal).

-w*size*
: Used to specify the buswidth (word size). *size* can be have the values: 1, 2, 4, 8, 16 or 32. If this option is not specified, the default buswidth is the value of the BUSWIDTH parameter of the GNX target setup (see Section 5.2). If the BUSWIDTH parameter is missing, a default buswidth of 2 bytes (16 bits) is used.

-b*bytenumber*
: Used to select an EPROM in a bank. The EPROM is denoted by the byte number of the word. *bytenumber* can have any value in the range of 0 to *buswidth*-1 (*buswidth* is the system buswidth). By default all EPROMs in the bank are selected.

-x*address*

Used to specify the start address of the first bank. *address* is an integer constant.

-l*size*

Used to specify the size of the EPROMs in use. *size* is specified in Kbytes (i.e. the EPROM size is 1024**size*). Default is 64 Kbytes.

-k[*bank#*:]*num*

Used to specify which banks should be selected. If two values are specified in the form *bank#:num*, *num* banks are selected starting from bank *bank#*. If only one value is specified, *num* banks are selected starting from the first bank (bank 0). By default **nburn** will select only the first bank.

-o *filename*

Used to specify the output filename. Since **nburn** can produce several output files, *filename is generally used as a generic name. For each output file,* **nburn** will add the extension *_banknum_bytenum* to the file name (unless the -n option is specified). *banknum* is the EPROM bank number, *bytenum* is the EPROM byte number. If this option is not specified, **nburn** will direct the output(s) to an auxiliary port of a VT100 compatible terminal. The EPROM programmer should be attached to this port. Before writing the output for each EPROM, **nburn** prints a message to the terminal and waits for you to plug in the EPROM.

-n Specifies that a filename extension is not required. This option is useful when only one output file is to be produced. The output file name in this case will be exactly as specified in the -o option.

-p*offset*

Used to specify an EPROM offset to which the data will be directed. By default the offset is 0. This option applies only to EPROMs of the first bank.

-i Specifies **nburn** output in Intel-hex format. This format supports EPROMs of up to 1 Mbytes. The default format is ASCII-hex, which format supports EPROMs of up to 64 Kbytes.

-m{1|2|3}

Specifies **nburn** output in one of the three Motorola formats: format 1 supports EPROMs of up to 64 Kbytes; format 2 supports EPROMs of up to 16 Mbytes; format 3 supports EPROMs of up to 4 Gbytes. The default format is ASCII-hex, which format supports EPROMs of up to 64 Kbytes.

-c

Specifies that a checksum byte is added for each EPROM. This option is used if the contents of the EPROM are to be verified at run-time (e.g. for diagnostics purposes). The checksum byte will be located in the first unused byte of the EPROM. The checksum byte value is calculated such that the xor of the one's complement of all bytes (including the checksum byte) is 0. Unused bytes are set by EPROM programmers to 0xFF, and therefore do not affect the checksum. If all bytes of the EPROM are occupied, **nburn** will issue a warning and will not add a checksum byte.

EXAMPLES

1. `nburn -x0x10000 -o out execfile`

   This command is used to burn one bank of EPROMs located at address 0x10000. The word size will be taken from the BUSWIDTH parameter of the GNX target setup. Assuming that the word size is 4, **nburn** will produce four output files, one file for each EPROM:

   ```
   out_0_0 out_0_1 out_0_2 out_0_3
   ```

   The EPROM size will be 64K (default size). The output format will be ASCII-hex (default format).

2. `nburn -w2 -b1 -x0x20000 -132 -i -o OUT execfile`

   This command is used to burn the second EPROM in a bank located at address 0x20000. The word size is 2 (i.e. there are two EPROMs in each bank, 0 and 1). The byte number is 1, signifying that the second EPROM is required. The EPROM size is 32K and the output format is intel-hex. Only one output file will be produced:

   ```
   out_0_1
   ```

3. `nburn -w2 -x0x10000 -k2 -m2 -o out execfile`

This command is used to burn two consecutive banks of EPROMs. The start address of the first bank is 0x10000. Each bank has two EPROMs because the word size is 2. Since there are four EPROMs, the following four output files will be produced:

```
out_0_0          (first bank, first EPROM)
out_0_1          (first bank, second EPROM)
out_1_0          (second bank, first EPROM)
out_1_1          (second bank, second EPROM)
```

The EPROM size is 64K (default). The output format is motorola format number 2.

4. `nburn -w4 -k1:1 -i execfile`

This command is used to burn the second bank (bank number 1) of a set of consecutive banks of EPROMs. The address of the first bank is 0 (default). However, since the second bank is required, data will be taken from the start address of the second bank. This address is equal to the size of one bank (i.e. the EPROM size 64K, multiplied by four EPROMs in each bank, gives 256K or 0x40000). Because no output file is specified, the output will be directed to the terminal's auxiliary port where the EPROM programmer is attached. Before writing the output for the first EPROM, **nburn** will print the following message:

```
Creating output data for bank 1 byte number 0
Press RETURN when ready...
```

A similar message is printed for each other EPROM. The appropriate EPROM should be plugged into the programmer before pressing the RETURN key.

SEE ALSO

Section 3.7
Series 32000 Development Board Monitor Reference Manual

5.7 NCMP

ncmp – compares two GNX binary files

USAGE

ncmp [*option*] *filename1 filename2* [*skip1* [*skip2*]]

DESCRIPTION

The two files are compared. If a file is "–," the standard input is used. Only one file may be standard input. ncmp read the first eight bytes from each file to determine the type. Type may be COFF, Modular COFF, archive, or any other legal type. Files are compared on the basis of their types. If the input file types differ, ncmp terminates and an error message is printed. Exit code 0 is returned for identical files, 1 for differing files, and 2 for an inaccessible or missing argument.

For COFF files, ncmp ignores the file time stamp. For archive files, ncmp ignores both the file time stamp and the file modification types in archive headers of each COFF member.

The *skip1* and *skip2* parameters are offsets from the beginning of *filename1* and *filename2* to be compared, respectively. The skip offsets may be specified as either octal (0), decimal or hexadecimal (OX) numbers.

Options:

- **-a** Compares two archives, ignoring the order of members in the archive, and prints a message for each differing archive member. (Note that archives with a differing member order will usually have a differing symbol directory.)
- **-l** Prints the byte number (decimal) and the differing bytes (octal) for each difference.
- **-s** Prints nothing for differing files; return codes only.
- **-V** Prints out the version number of ncmp.
- **-v** Prints the byte number (decimal) and the differing bytes (octal) for each difference, while adding the character interpretation of differing bytes for each difference.

5.8 NLORDER

`nlorder` – finds ordering relation for an object library

USAGE

`nlorder` *filename ...*

DESCRIPTION

The input is one or more object or library archive (see *nar*) *filenames*. `Nlorder` writes to standard output a list of pairs of object filenames, meaning that the first file of the pair refers to external identifiers defined in the second file. The output may be processed by `tsort`, if available on a *Series 32000*/UNIX host system, to find an ordering of a library suitable for one-pass access by `nmeld`.

This command line builds a new library from existing `.o` files.

```
nar cr library `nlorder *.o | tsort`
```

`Nlorder` invokes `nnm`. The user's PATH environment variable must be set to specify the directory containing `nnm`.

5.9 NNM

nnm – prints name list of *Series 32000* GNX object file

USAGE

nnm [*option*] *filename*

DESCRIPTION

The nnm command displays the symbol directory of each *Series 32000* GNX object file. The object file may be a relocatable or absolute GNX object file; or it may be an archive of relocatable or absolute GNX object files. For each symbol, the following information will be printed:

Name The name of the symbol (only the first 74 characters).

Value The symbol's value, depending on its storage class, is expressed as an offset or an address.

Class The symbol's storage class.

Type The symbol's type and derived type. If the symbol is an instance of a structure or of a union, the structure or union tag is given following the type, *e.g.*, struct-tag. If the symbol is an array, the array dimensions are given following the type, *e.g.*, `char[n][m]`. Note that the object file must have been compiled with the –g option of the nmcc command for this information to appear.

Size The symbol's size in bytes, if available. Note that the object file must have been compiled with the –g option of the nmcc command for this information to appear.

Line The source line number at which the symbol is defined, if available. Note that the object file must have been compiled with the –g option of the nmcc command for this information to appear.

Section For storage classes static and external, the object file section containing the symbol, *e.g.*, text, data or bss.

The output of nnm may be controlled using the following options:

-o Prints the value and size of a symbol in octal instead of in decimal.

-x Prints the value and size of a symbol in hexadecimal instead of in decimal.

-h Does not display the output header data.

-v Sorts external symbols by value before they are printed. Sorts the first 74 characters by default.

-r Prepends the name of the object file or archive to each output line.

-p Produces easily parsable, terse output. Each symbol name is preceded by its value (blanks if undefined) and one of the letters `U` (undefined), `A` (absolute), `T` (text segment symbol), `D` (data segment symbol), `S` (user-defined segment symbol), `R` (register symbol), `F` (file symbol), `B` (bss section), or `C` (common symbol). If the symbol is local (non-external), the type letter is in lowercase. Name can be up to 1024 characters.

-n Sorts external symbols by name before they are printed.

-e Prints only external and static symbols.

-f Produces full output. Prints redundant symbols (.text, .lib, .comment, .data, and .bss), normally suppressed.

-u Prints undefined symbols only.

-V Prints the version of the `nnm` command.

-T By default, `nnm` prints the entire name of each symbol listed. Because object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned. The –T option causes `nnm` to truncate every name which would otherwise overflow its column and then places an asterisk as the last character in the displayed name to mark it as truncated.

Options may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both **`nnm name -e -v`** and **`nnm -ve name`** print the static and external symbols in *name*, with external symbols sorted by value.

MS-DOS Additional Options:

`@filename` reads `nnm` options from file. The **`@`** option directs the GNX `nnm` utility to read additional options from the named file. This option avoids the MS-DOS limitation on the length of invocation lines, and enables passing options of unlimited length.

DIAGNOSTICS

"nnm: *filename*: cannot open"
if *filename* cannot be opened.

"nnm: *filename*: bad magic"
if *filename* is not an appropriate GNX object file.

"nnm: *filename*: no symbols"
if the symbols have been stripped from *filename*.

5.10 NSIZE

`nsize` – prints section sizes of GNX object files

USAGE

`nsize` [*option*] *filename*

DESCRIPTION

The `nsize` command produces section size information for each section in an object file. The size of the text, data, bss (uninitialized data), link, and static sections are printed along with the total size of the object file. If an archive file is input to the `nsize` command, the information for all archive members is displayed.

`-n` Includes NOLOAD sections in the size.

`-o` Forces the output in octal. Default is decimal.

`-f` Produces full output, that is, it prints the size of every loadable section, followed by the section name in parentheses.

`-v` Prints out the version number of the `nsize` utility.

`-x` Forces the output in hexadecimal. Default is decimal.

MS-DOS Additional Options:

`@filename` reads `nsize` options from file. The **`@`** option directs the GNX Version `nsize` utility to read additional options from the named file. This option avoids the MS-DOS limitation on the length of invocation lines, and enables passing options of unlimited length.

DIAGNOSTICS

"`nsize:` *filename*`:  cannot open`"
if *filename* cannot be opened.

"`nsize:` *filename*`:  bad magic`"
if *filename* is not an appropriate GNX object file.

5.11 NSTRIP

`nstrip` – strips symbol and line number information from a GNX object file

USAGE

`nstrip` [*option*] *filename*

DESCRIPTION

The `nstrip` command strips the symbol directory and line number information from GNX object files, including archives. Once this has been performed, no symbolic debugging access will be available for that file; therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol directory can be controlled by using any of the following options:

- **`-l`** Strips line number information only; does not strip any symbol directory information.
- **`-x`** Strips local symbols and line number information only.
- **`-b`** Strips local symbols except scoping information (*e.g.*, beginning and end-of-block delimiters) only.
- **`-r`** Resets the relocation indexes into the symbol directory. Strips the local symbols and line number information only.
- **`-v`** Prints the version of the `nstrip` utility on the standard error output.

The –l, –x, –b, and –r options are mutually exclusive.

If there are any relocation entries in the object file and any symbol table information is to be stripped, `nstrip` will complain and terminate without stripping *filename*, unless the –r flag is used.

If the `nstrip` command is executed on a GNX archive file, the archive symbol directory will be removed. The archive symbol directory must be restored by executing the `nar` command with the `s` option before the archive can be link-edited by the `nmeld` command. `Nstrip` will instruct the user with appropriate warning messages when this situation arises.

The purpose of this command is to reduce the file storage overhead taken by the object file.

MS-DOS Additional Options:
`@filename` reads `nstrip` options from file. The `@` option directs the GNX Version `nstrip` utility to read additional options from the named file. This option avoids the MS-DOS limitation on the length of invocation lines, and enables passing options of unlimited length.

DIAGNOSTICS

"`nstrip: `*filename*`: cannot open`"
if *filename* cannot be opened.

"`nstrip: `*filename*`: bad magic`"
if *filename* is not an appropriate GNX object file.

"`nstrip: `*filename*`: relocation entries present; cannot strip`"
if *filename* contains relocation entries and the –r flag is not used, the symbol directory information cannot be stripped.

Appendix A
GLOSSARY

.gnxrc (gnx.ini on MS-DOS) A GNX target specification file that is used by GNX tools to obtain the CPU, FPU, MMU, system bus-width, and OS target specifications.

Assembler Assembles *Series 32000* assembly language source programs and generates relocatable object modules. Relocatable object modules must be linked to create executable load modules.

COFF Acronym for the Common Object File Format. This is the standard object file format for the Unix System V operating system, and for the GNX software tools. A COFF file contains machine code and data and additional information for relocation and debugging purposes.

Cross configuration When the compilation and execution of the compiled program are done on different machines (the host and target machines are different).

DBUG GNX symbolic debugger. It is used for the symbolic debugging of high level and assembly language programs.

Development board The 32000 based system used for developing/running programs and user applications.

Executable object file An executable object file is the final product of a linking process. In an executable object file all external symbolic references have been resolved. The executable object file is therefore in a form that can be executed on the *Series 32000*-based target system.

GTS A menu-driven program which takes the user's target specification and constructs a target configuration file, which is read by each of the GNX tools when invoked.

Host machine The machine on which the compiler runs.

Linker A GNX utility that creates executable files by combining object files, performing relocation and resolving external references. The linker also processes symbolic debugging information.

Minstall A GNX utility that copies or moves a binary file to a specified destination.

Monfix A GNX utility that creates *Series 32000* bootstrap programs by modifying the first 16 bytes of a GNX executable file.

Monitor A GNX utility that provides the interface between the hardware execution

environment and the GNX language tools running on a host development system. The monitor is provided in PROMs on National Semiconductor's development board. Sources of the monitor are supplied with each GNX binary release.

Nar Utility that maintains groups of files combined into a single archive file. The utility is used to create and update library files used by the GNX Linker.

Nburn A GNX utility that loads the specified bytes of a file to an EPROM burner in ASCII-HEX, Intel-hex, extended Intel-hex, or Motorola s-record format.

Ncmp A GNX utility that compares two files.

Nlorder A GNX utility that displays the ordering relation for object files. The input may be one or more object or library archive files.

Nnm A GNX utility that displays the symbol table of a *Series 32000* GNX object file. This tool is used to obtain information on a symbol within an executable object file. For each symbol, Nm displays the symbol name, storage class, type, size, and source line number at which the symbol is defined.

Nsize A GNX utility that displays the size information for each section and optional header information of a *Series 32000* GNX object file.

Nstrip A GNX utility that strips symbol and line number information from a *Series 32000* GNX object file, thereby reducing the size of the executable file.

Object file A file that is the output of either the assembler or the linker. An object file contains compiled code and data and additional information for relocation information and debugging purposes.

Option The UNIX term for a parameter, specified on the command line, that is used to control the utility.

Target machine The machine on which the program being compiled will run.

INDEX

A

Assembler
  (nasm) 2-3
AVAIL_SWAP 2-6

B

Board
  development A-1

C

C optimizing compiler
  (nmcc) 2-2
CMDDIR 2-6
Command summaries 5-1
  gts 5-2
  minstall 5-3
  monfix 5-4
  nar 5-6
  nburn 5-8
  ncmp 5-13
  nlorder 5-14
  nnm 5-15
  nsize 5-18
  nstrip 5-19
Compare binary files (ncmp) 2-5
Configuration
  cross A-1

D

DB support library 3-2
Debugger
  dbug 2-3

E

Environment variables
  AVAIL_SWAP 2-6
  CMDDIR 2-6
  INCLUDEPATH 2-6
  LIBPATH 2-6
  PITFILE 2-6
  TMPDIR 2-6

G

GNX language tools 1-1
  table 1-2
GNX utilities 2-2
  dbug 2-3
  nar 2-4
  nasm 2-3
  nburn 2-4
  ncmp 2-5
  nlorder 2-5
  nmcc 2-2
  nmeld 2-3
  nmpc 2-2
  nnm 2-5
  nsize 2-5
  nstrip 2-5
gts command 5-2

I

Include files 2-6
INCLUDEPATH 2-6

L

LIBPATH 2-6
Librarian (nar) 2-4
Libraries 2-7
Linker
  nmeld) 2-3

M

Machine
  host A-1
  target A-2
minstall command 5-3
mon16 3-4
mon32 3-4
mon332 3-4
mon532 3-4
monCG16 3-4
monCG160 3-4
monfix 3-7
  command 5-4
monGX32 3-4
monGX320 3-4
monGX32E 3-4
Monitors
  mon16 3-4

mon32 3-4
mon332 3-4
mon532 3-4
monCG16 3-4
monCG160 3-4
monGX32 3-4
monGX320 3-4
monGX32E 3-4

N

nar command 5-6
nburn
command 5-8
utility 2-4, 3-7
ncmp command 5-13
nlorder
command 5-14
utility 2-5
nnm
command 5-15
utility 2-5
nsize
command 5-18
utility 2-5
nstrip
command 5-19
utility 2-5

O

Option A-2

P

Pascal
compiler 2-2
PITFILE 2-6

S

Sample sessions
link with development board 3-5
SPROF 2-4

T

TMPDIR 2-6

U

Using
nburn and monfix 3-7

Series 32000®

GNX — Version 4.4
Linker User's Guide

Customer Order Number 424010506-004

June 1992

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
4.0	May 1990	First Release. Addition of directive file allocation options. Automatic generation of special symbols is now possible. A section can now have both a ROM and RAM address.
4.1	Sep 1990	Updated error list in appendix B.
4.2	Feb 1991	Synchronization revision. No changes.
4.3	Aug 1991	Synchronization revision. No changes.
4.4	Jun 1992	MS-DOS support added.

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

CONTENTS

Chapter 1 OVERVIEW

1.1 INTRODUCTION 1-1
1.2 REASONS FOR A LINKER 1-1
1.3 THE COMMON OBJECT FILE FORMAT 1-1
1.4 LINKER INPUT 1-2
1.4.1 The Simple Object File 1-2
1.4.2 The Partially Linked Object File 1-2
1.4.3 The Library File 1-3
1.4.4 The Directive File 1-3
1.5 LINKER OUTPUT 1-3
1.5.1 The Executable Object File 1-3
1.5.2 Partially Linked Object File 1-3
1.5.3 The Memory Map 1-4
1.6 LINKER FUNCTIONS 1-4
1.6.1 Resolution of Symbolic References 1-4
1.6.2 Allocation of Output Sections 1-5
1.6.3 Relocation of Memory Addresses 1-5

Chapter 2 THE INVOCATION LINE

2.1 INTRODUCTION 2-1
2.2 UNIX ENVIRONMENT INVOCATION 2-1
2.2.1 Libraries 2-2
2.3 VMS ENVIRONMENT INVOCATION 2-2
2.3.1 Libraries 2-3
2.4 INVOCATION OPTIONS 2-3
2.4.1 Specify Output Filename 2-3
2.4.2 Specify Directive File 2-4
2.4.3 Specify Library Filename 2-4
2.4.4 Specify Library Directory 2-5
2.4.5 Request Memory Map 2-5
2.4.6 Specify Program Entry Point 2-6
2.4.7 Retain Relocation Information 2-6
2.4.8 Keep Relocation Information 2-7
2.4.9 Strip Symbolic Information 2-7
2.4.10 Strip Local Symbolic Information 2-7
2.4.11 Specify Undefined Symbol 2-8
2.4.12 Request Initialization Table 2-8

2.4.13 Specify Fill Value for Section Gaps 2-8
2.4.14 Suppress Size Warning Message for Common Data . . . 2-9
2.4.15 Suppress Error Message 2-9
2.4.16 Issue Warning for Defined Common Data 2-9
2.4.17 Output Linker Version Information 2-10
2.4.18 Specify Version Stamp 2-10

Chapter 3 THE LINKER DIRECTIVE FILE

3.1 INTRODUCTION 3-1
3.2 STRUCTURE OF THE DIRECTIVE FILE 3-1
3.2.1 Example of a Directive File 3-2
3.3 DIRECTIVE FILE EXPRESSIONS 3-2
3.3.1 Examples 3-3
3.4 COMMENT 3-3
3.5 INPUT FILE SPECIFICATION 3-3
3.6 **MEMORY** STATEMENT 3-4
3.7 **SECTIONS** STATEMENT 3-5
3.7.1 Output Section Specification 3-5
3.7.2 Input Section Specification 3-6
3.7.3 Allocating a Section to Memory 3-8
3.7.4 Aligning a Section 3-11
3.7.5 Setting the Section Type 3-12
3.7.6 Grouping Output Sections 3-13
3.8 ASSIGNMENT STATEMENT 3-14
3.8.1 Symbol Assignment Within **SECTIONS** Statement . . . 3-15
3.8.2 Creating Gaps Within An Output Section 3-15
3.9 OUTPUT FILE OPTIONS 3-16
3.9.1 Change Default Filename and Permission 3-16
3.9.2 Optional Header Magic Number 3-16

Chapter 4 RESOLUTION OF SYMBOLIC REFERENCES

4.1 INTRODUCTION 4-1
4.1.1 Examples of Symbol Definition and Reference 4-1
4.1.2 Symbol Resolution Using the Symbol Table 4-2
4.1.3 Example 4-2
4.2 LIBRARY PROCESSING 4-3
4.2.1 Example 4-3
4.3 COMMON DATA PROCESSING 4-4
4.3.1 Examples 4-5
4.4 SYMBOL DEFINITION IN THE DIRECTIVE FILE 4-5
4.5 LINKER DEFINED SYMBOLS 4-6

4.5.1 Example 4-6

Chapter 5 ALLOCATION OF OUTPUT SECTIONS

5.1 INTRODUCTION 5-1
5.2 CREATING OUTPUT SECTIONS FROM INPUT SECTIONS . . 5-1
5.2.1 Example 5-1
5.3 ASSIGNING AN ADDRESS TO AN OUTPUT SECTION 5-2
5.3.1 Example 5-3
5.4 DATA INITIALIZATION SUPPORT 5-3
5.4.1 Example 5-5
5.5 LINKER CREATED INPUT SECTIONS 5-7
5.5.1 Example 5-7
5.6 MEMORY MAP 5-8
5.6.1 Example 5-9

Chapter 6 RELOCATION OF MEMORY ADDRESS

6.1 INTRODUCTION 6-1
6.2 RELOCATION INFORMATION 6-1
6.2.1 Example 6-1
6.3 THE RELOCATION PROCESS 6-2
6.3.1 Example 6-3

Appendix A DIRECTIVE FILE EXPRESSIONS

A.1 INTRODUCTION A-1
A.2 INTEGER SYNTAX A-1
A.2.1 Decimal Value Syntax A-1
A.2.2 Octal Value Syntax A-1
A.2.3 Hexadecimal Value Syntax A-1
A.3 UNARY OPERATORS A-2
A.4 BINARY OPERATORS A-2
A.5 ASSIGNMENT OPERATORS A-2
A.6 SPECIAL FUNCTIONS A-3
A.6.1 Size of Output Function A-3
A.6.2 Memory Address Function A-4
A.6.3 File Address Function A-4
A.6.4 Next Address Function A-4
A.6.5 Highest Memory Address Function A-4

Appendix B LINKER ERROR MESSAGES

B.1 INTRODUCTION B-1

B.2 ERROR MESSAGES B-1

Appendix C GLOSSARY

INDEX

HOW TO USE THIS MANUAL

PURPOSE OF THIS GUIDE

This guide describes the GNX Native and Cross Support Linker. The GNX Linker is an essential component of any *Series 32000* microprocessor software development tool set.

The GNX Linker is used to create an executable file for any *Series 32000*-based native or cross application. For embedded applications, a powerful and flexible user-written directive file can be used to control linker actions.

INTENDED AUDIENCE

This guide is for the programmer who uses GNX software development tools to generate application software for *Series 32000*-based systems.

HOW THIS GUIDE IS ORGANIZED

Chapter 1 is an overview of the linker. It provides information on linker input, output and functions. Chapter 2 summarizes the linker invocation line and lists all invocation options and arguments. The linker directive file is explained in Chapter 3. This chapter contains an explanation of the syntax used to control linker actions. Examples are provided to illustrate the use of the directive file.

Chapters 4, 5, and 6 describe in detail the linker operations. Chapter 4 is devoted to the resolution of symbolic references. Chapter 5 explains the allocation of output sections. And the relocation of memory addresses is described in Chapter 6.

Appendix A defines directive file expressions. Error messages are explained in Appendix B. And Appendix C is a glossary of GNX terms used in this guide.

CONVENTIONS USED IN THIS GUIDE

Convention	Meaning
`ld { `*option* ` | `*filename*` } ...`	Syntax descriptions show all actual user-input as constant width. Italics indicate generic operands that are user-supplied. Spaces are significant and must be entered as shown. Multiple spaces and tabs may be used in place of a single space.
{ }	Large braces indicate two or more items, of which only one must be used. The arguments are separated by a logical OR sign (|).
[ ]	Large brackets indicate that the enclosed item(s) is optional.
|	Logical OR sign is used to separate items of which only may be used.
...	Ellipsis indicate optional repetition of the preceding item(s).

Note: All other characters, including small braces and brackets ({ }, [ ]), are actual user-input.

RELATED DOCUMENTS

The following National Semiconductor documents are recommended for additional information:

- *Support Libraries Reference Manual* provides details on the GNX library routines.
- *COFF Programmer's Guide* describes the object file format.
- *Assembler Reference Manual* describes the syntax, format and function of the GNX assembly language.

NEW AND CHANGED FEATURES

The following technical changes have been made to the GNX Linker since Version 3:

- The directive file allocation options `INTO`, `ROMBIND` and `ROMINTO` have been added.
- A section can now be duplicated (i.e. have both a ROM and RAM address). The linker will allocate memory for this section twice.
- An invocation option to request an initialization table has been added.
- Automatic generation of special symbols is now possible.
- The directive file is now processed after all other input files, regardless of invocation position.

Chapter 1

OVERVIEW

1.1 INTRODUCTION

The GNX Linker is a tool used for the linking of object files. Object files are produced either by the GNX Assembler and Compiler when translating source-language programs, or by the GNX Linker when creating partially linked object files to be used in a subsequent linking operation. The linker combines object files by resolving symbolic references, allocating output sections, and relocating memory addresses to produce an executable object file. This file can then be executed on a *Series 32000*-based target system.

The linker is controlled by the invocation line and a directive file. A default directive file is supplied with the GNX package. A user-written directive file can be used to override the default directive file. This ability is especially useful for embedded applications.

1.2 REASONS FOR A LINKER

The GNX Linker is an essential component of the GNX software development package. It provides for:

- Quick and easy modular programming since separately compiled object files can be linked together into a single executable object file.
- Control over memory allocation.

1.3 THE COMMON OBJECT FILE FORMAT

The Common Object File Format (COFF) was developed by AT&T as the standard UNIX/SO System V format for object files. The GNX tools use this definition as the standard object file format.

Object files are basically made up of sections. A section is a contiguous block of code or data having common attributes, and is the smallest unit of relocation. A section can have any name, however the standard section names that are created by default are:

- .text. The .text section contains executable code.
- .data. The .data section contains initialized data. This data is available at runtime without any explicit assignment statement from the program.

- .bss. The .bss section contains uninitialized data. Since this data is uninitialized, the .bss section does not occupy space in the object file. When program execution starts, the .bss values found in memory are environmentally dependent. Most environments initialize the .bss section to zeroes.

A more thorough review of the structure of the GNX COFF file may be found in the *COFF Programmer's Guide.*

1.4 LINKER INPUT

Input to the linker consists of simple object files produced by the assembler or compiler, partially linked object files produced by a previous linking operation, and library files. These input files are combined to produce an output file. The linker directive file can also be considered as linker input.

The term "input section" is defined as a section of a linker input object file.

1.4.1 The Simple Object File

The simple object file is the most common type of linker input. It is created when the assembler or compiler translates source-language programs into COFF. Simple object files may contain unresolved external references. A simple object file is specified either on the invocation line (see Sections 2.2 and 2.3) or in a directive file (see Section 3.5).

1.4.2 The Partially Linked Object File

The partially linked object file is produced by a previous linking operation. It contains unresolved external references, and is therefore not executable. Partially linked object files must be included as input in a subsequent linking operation that produces an executable file.

The partially linked object file is used for many programming needs:

- A complex linking task is made easier by creating small groups of simple object files. Each group is then linked together to create a partially linked object file. Finally the partially linked object files can be linked to produce an executable object file.
- A large program can be modified without having to relink the entire program.
- A set of routines can be produced for use in different application programs.

Refer to Section 2.4.7 for an explanation of the Retain Option used to create partially linked files.

1.4.3 The Library File

The library file is a collection of simple object files, each representing a useful function. Several library files are supplied as part of the GNX software package. You can also create your own library file using the GNX archiver (see the *Commands and Operations Manual* for details).

Library members that are referenced (by an external symbolic reference) are selected by linker and included in the linking process. Library members that are not referenced are not included in the linking process.

Refer to Section 2.4.3 for the invocation line option to specify a library file.

1.4.4 The Directive File

The directive file controls certain actions of the linker (specifically memory allocation). You can exercise considerable control over the linking operation by creating your own directive file. This feature is especially useful for embedded applications.

Chapter 3 offers a detailed description of the directive file.

1.5 LINKER OUTPUT

Output of the linker consists of executable object files and partially linked object files. A memory map can also be considered as linker output.

The term "output section" is defined as a section of a linker output object file.

1.5.1 The Executable Object File

The executable object file is the final linker output. In an executable object file all external symbolic references have been resolved. The executable object file is therefore in a form that can be executed on the *Series 32000*-based target system.

1.5.2 Partially Linked Object File

The partially linked object file is produced by a previous linking operation. It contains unresolved external references, and is therefore not executable. Partially linked object files must be included as input in a subsequent linking operation that produces an executable file.

The partially linked object file is used for many programming needs:

- A complex linking task is made easier by creating small groups of simple object files. Each group is then linked together to create a partially linked object file. Finally the partially linked object files can be linked to produce an executable object file.
- A large program can be modified without having to relink the entire program.
- A set of routines can be produced for use in different application programs.

Refer to Section 2.4.7 for an explanation of the Retain Option used to create partially linked files.

1.5.3 The Memory Map

The memory map illustrates the allocation of memory after the linking process. It also illustrates the composition of the output sections from input sections. Section 5.6 provides a complete explanation of the memory map.

1.6 LINKER FUNCTIONS

The linker performs three basic functions: resolution of symbolic references, allocation of output sections, and relocation of memory addresses.

1.6.1 Resolution of Symbolic References

A symbol is used either to mark a program location or to represent a data element. Object files contain a symbol table. The symbol table is comprised of information about symbols defined or referenced in the source program. An external symbol is a symbol that can be referenced from any object file.

The linker resolves references to external symbols. The resolution of symbolic references is the process by which the linker matches an external symbolic reference with its definition. This process is described in Chapter 4.

A symbol can also be defined in a directive file. Section 3.8 explains the use of the assignment statement for this purpose.

1.6.2 Allocation of Output Sections

The linker determines which part of memory is available for the allocation of output sections. The linker then assembles output sections from input sections and binds each output section to a starting memory address.

Through use of the directive file, you can specify memory configuration. The directive file instructs the linker which parts of memory are available for allocation (see Section 3.6), how to construct output sections from input sections, and how to allocate memory for these output sections (see Section 3.7). Control over the allocation of memory allows you to create a memory layout for various hardware requirements.

Refer to Chapter 5 for a complete description of the allocation of output sections.

1.6.3 Relocation of Memory Addresses

Once external symbolic references have been resolved and output sections allocated to memory, the linker assigns each symbolic reference its actual memory address.

Chapter 6 explains the relocation of memory address.

Chapter 2

THE INVOCATION LINE

2.1 INTRODUCTION

This chapter explains the GNX Linker invocation line. Linker invocation is host-specific and is therefore different for UNIX and VMS operating system environments.

The linker is directly invoked by specifying the appropriate invocation name followed by invocation line arguments. These arguments specify a list of object and library files to link and linker options.

In the UNIX environment, the linker is often invoked by a a compiler driver. A compiler driver invokes the linker with a predetermined set of linker options. If these options are not suitable for your needs, you can either force the compiler driver to pass specific options to the linker, or terminate the compilation process after object file creation and invoke the linker directly.

2.2 UNIX ENVIRONMENT INVOCATION

SYNTAX FOR NATIVE LINKING:

`ld` { *option* | *filename* } ...

SYNTAX FOR CROSS LINKING:

`nmeld` { *option* | *filename* } ...

`ld` or `nmeld` is the invocation name.
option is any valid linker invocation line option. Each option is preceded by a dash (—), and options must be separated by a space.
filename is any valid object or library file. Object and library filenames must be separated by a space. *filename* must include a complete or relative pathname if the specified object or library file is not in your current directory.

filename and *option* can be placed in any order within the invocation syntax. However, object files which contain symbolic references to a library must precede that library filename on the invocation line. Libraries specified explicitly or through the `-l` invocation line option are processed as they are encountered.

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

2.2.1 Libraries

Libraries can be specified through the `-l` invocation line option (Section 2.4.3). The linker searches for these libraries by directory, according to a default directory list. The default library location for a UNIX cross environment is *gnxdir*`/lib` (where *gnxdir* is the top-level directory of the installed GNX tools). The default locations for a UNIX native environment (i.e., SYS32/30) are *gnxdir*`/usr/lib`, *gnxdir*`/lib`, `/usr/lib`, and `/lib`.

The `LIBPATH` environment variable can be set to override the default directory search.

SYNTAX:

`setenv LIBPATH` *directory-search-list* (c-shell)
`LIBPATH=`*directory-search-list* `; export LIBPATH` (shell)

directory-search-list is a list of library directories separated by colons.

At link-time, the linker will search the listed directories for the libraries specified with the `-l` invocation line option.

2.3 VMS ENVIRONMENT INVOCATION

SYNTAX:

`nmeld` [*option* ...] *filename* [*,filename* ...]

`nmeld` is the invocation name for the VMS environment.
option is any valid linker invocation line option. Each option is preceded by a slash (/).
filename is any valid object or library file. Object and library filenames must appear as a comma-separated list. *filename* must include a complete file specification if the file is not in your current directory (see Section 2.3.1).

A default extension, .OBJ, is assumed if a specified object filename has no extension.

option can be placed either before or after the filename list. Within the filename list, filenames may appear in any order. However, object files which contain symbolic reference to a library must precede that library filename on the invocation line.

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 Libraries

Libraries can be specified on the linker invocation line in the invocation syntax. Alternatively, the GNX$LIBRARY and GNX$LIBRARY_*n* logical names may be set at the VMS invocation line to define libraries for linking.

SYNTAX:

```
define GNX$LIBRARY library-filename
define GNX$LIBRARY_1 library-filename
define GNX$LIBRARY_2 library-filename
...
```

library-filename is any library filename with a complete pathname.

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. When a logical name is defined for a library, it does not have to be explicitly mentioned on the invocation line. The linker automatically processes these libraries after all other input files.

2.4 INVOCATION OPTIONS

This section describes the linker invocation options for both the UNIX and VMS environments. Invocation options are used for controlling linking operations such as specifying a directive file, requesting an output memory map, naming the output file, and suppressing error and warning messages.

UNIX invocation options begin with a dash (—). VMS invocation options begin with a slash (/). Tables 2-1 and 2-2 at the end of this chapter provide an abbreviated syntax guide for the invocation options of each host system, respectively.

2.4.1 Specify Output Filename

This invocation option is used to specify a name for the output file that is produced by the linker. Use of this option overrides the default output filename.

SYNTAX:

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

filename is any valid filename.

In the UNIX environment, the default output filename is `a32.out` when cross linking and `a.out` when native linking. In the VMS environment, the default output filename is *output_filename*.EXE. *output_filename* is the name of the first filename on the invocation line.

2.4.2 Specify Directive File

With the Specify Directive File invocation option you can designate a directive file to be used by the linker in creating the executable object file. Use of this invocation option will override the default directive file used by the linker.

SYNTAX:

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

filename is any valid directive file. *filename* must include a complete pathname if the directive file is not in your current directory.

When a directive file is not specified in the invocation line, the linker will use the directive file specified by the `LINKERFILE` parameter of the GNX Target Setup (gts) utility program. Refer to the *GNX Commands and Operations Manual* (Version 4) for a detailed discussion of the `gts` utility program.

If the `LINKERFILE` parameter is missing the linker uses the default directive file `linker.def`, located in the GNX root directory on a cross environment, or the *gnxdir*`/bin` directory in a native environment (where *gnxdir* is the top-level directory of the installed GNX tools).

If `linker.def` does not exist, the linker issues a warning message and then follows a predefined trivial link process. You should not depend on the trivial link process to produce meaningful results.

See Chapter 3 for a full explanation of the linker directive file.

2.4.3 Specify Library Filename

The Specify Library Filename invocation option can be used to specify libraries to be used by the linker. This option is useful for specifying system libraries and is available only in the UNIX environment.

SYNTAX:

`-l`*x*	(UNIX)
Not applicable	(VMS)

x is the sequence of up to nine characters that define a system library name in the filename `lib`*x*`.a`.

On the invocation line the `-l`*x* option must follow the list of object files with external references that will be resolved in the specified library. The linker will first search for this library in the directories specified through the `-L` invocation line option (Section 2.4.4) and then in the default library locations (Section 2.2.1.)

2.4.4 Specify Library Directory

With this option you define the directory in which the linker will first search for a library specified with the `-l` invocation option (Section 2.4.3). This option is available only in the UNIX environment.

SYNTAX:

`-L`*dir*	(UNIX)
Not applicable	(VMS)

dir is any valid directory pathname containing user libraries.

The `-L` option must precede any `-l` option. The linker searches for libraries specified through the `-l` invocation line option first in *dir* and then in the default library locations (Section 2.2.1).

2.4.5 Request Memory Map

The Request Memory Map invocation option generates a memory map of the executable object file. The format and contents of the memory map are detailed in Section 5.6.

SYNTAX:

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

map_filename is any valid map filename.

If you use this option in the UNIX environment, the memory map is sent to standard output. You must redirect standard output to save the memory map to a file.

If you use this option in the VMS environment, by default the memory map is created as a file and named *output_filename*.MAP, with *output_filename* corresponding to the name of the linker executable file that is mapped. If *map_filename* is specified, it will have the default extension .MAP.

2.4.6 Specify Program Entry Point

This option is used to indicate to the linker the entry point of your program. The entry point is used by the operating system or the debugger as the starting point for program execution. Entry point information is a special part of the COFF file and does not necessarily represent the actual beginning of the .text section.

By default, the linker will look for the external symbol `start` as indicating the entry point. If `start` is not found, the linker will look for the external symbol `_main`. If `_main` is not found, the linker issues a warning message and sets the entry point to 0.

SYNTAX:

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

symbol is an external symbol that marks the program entry point. To preserve case-sensitivity on VMS, *symbol* should be enclosed in double-quotes (""). If *symbol* is not found, the linker issues a warning message and sets the entry point by following the above mentioned default procedure.

2.4.7 Retain Relocation Information

The Retain Relocation Information option must be used if the product of the linking operation is to be a partially linked file (i.e. not all symbolic references have been resolved) that may be used as input in a subsequent link.

SYNTAX:

`-r`	(UNIX)
`/RETAIN`	(VMS)

Use of the Retain Relocation Information option ensures that the linker will retain relocation information and not issue a linking error for unresolved external references.

2.4.8 Keep Relocation Information

This option allows you to instruct the linker to keep relocation information in your executable object file.

SYNTAX:

`-k`	(UNIX)
`/KEEP`	(VMS)

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

2.4.9 Strip Symbolic Information

With this option you can instruct the linker to remove the symbol table and line number information from the executable file the linker produces, thereby reducing the size of the executable file.

SYNTAX:

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

Since symbolic information is used for debugging and for relocation of memory addresses, the strip symbolic information option should not be specified if the output file is either to be used in debugging or is a partially linked object file.

2.4.10 Strip Local Symbolic Information

The Strip Local Symbolic Information invocation option is used to remove only local symbolic information from the linker output file. This option is useful for reducing the size of linker output files.

SYNTAX:

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

Since local symbolic information is used for debugging, the strip local symbolic information option should not be specified if the output file is to be used in debugging. However, the output file can be a partially linked object file.

2.4.11 Specify Undefined Symbol

The Specify Undefined Symbol invocation option allows you to create a "pseudo" external reference to a symbol. This may be used to force the linker to select a library member for the linking process. Unless this option is specified, the linker will link library member only if they resolve an external symbolic reference from a previously specified object file.

SYNTAX:

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

symbolname is any valid symbol name. To preserve case sensitivity on VMS, may be enclosed in double-quotes (" ").

2.4.12 Request Initialization Table

The Request Initialization Table invocation option instructs the linker to create an initialization table to be used at run-time. The initialization table is used to copy code or data from ROM to RAM, and to initialize to zero all uninitialized data (typically found in the .bss section). The table is found in the linker created section .init.

SYNTAX:

`-i`	(UNIX)
`/INITTABLE`	(VMS)

For a detailed description of the initialization table refer to Section 5.4.

2.4.13 Specify Fill Value for Section Gaps

This invocation option is used to specify a fill value other than zero for section gaps. By default, the linker fills any gaps created in output sections with a zero value.

SYNTAX:

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

int is an 2-byte unsigned integer constant.

2.4.14 Suppress Size Warning Message for Common Data

This invocation option is used to suppress the issuance of warning messages by the linker when various references to common data are of different memory address sizes.

SYNTAX:

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

Common data are consolidated by the linker and allocated to a linker created .bss section. Normally, every reference to common data is of the same size. However, if the references are of different sizes, the linker will issue a warning message for each reference that is contrary to the first found common data size. In this case the linker will use the largest size.

2.4.15 Suppress Error Message

With this option you can instruct the linker to suppress all nonfatal error messages that describe problems encountered during the linking operation.

SYNTAX:

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

Only fatal errors cause the linking operation to abort immediately. Error messages describing fatal errors will be issued by the linker.

2.4.16 Issue Warning for Defined Common Data

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

SYNTAX:

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

2.4.17 Output Linker Version Information

The Output Linker Version Information option produces information regarding the version and revision numbers of the linker in use.

SYNTAX:

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

Version and revision number information is useful in determining whether changes and updates in the linker package apply to the linker you are using.

2.4.18 Specify Version Stamp

This option allows you to specify a version stamp for identifying linker output files.

SYNTAX:

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

A version stamp is a 16-bit integer constant. It is stored in a special field in the optional header of linker output files.

Table 2-1. Unix/MS-DOS Environment Invocation Options

OPTION	EXPLANATION	SECTION
`-k`	Keep relocation information in executable file	Section 2.4.8
`-V`	Output linker version information	Section 2.4.17
`-i`	Request an initialization table	Section 2.4.12
`-r`	Retain relocation information	Section 2.4.7
`-m`	Request an output memory map	Section 2.4.5
`-d` *filename*	Specify a directive file	Section 2.4.2
`-L`*dir*	Specify a directory to search for libraries	Section 2.4.4
`-f` *int*	Specify a fill value	Section 2.4.13
`-l`*x*	Specify a library file for linking	Section 2.4.3
`-e` *symbol*	Specify a program entry point	Section 2.4.6
`-u` *symbol*	Specify an undefined symbol	Section 2.4.11
`-o` *filename*	Specify an output filename	Section 2.4.1
`-VS` *int*	Specify a version stamp	Section 2.4.18
`-x`	Strip local symbolic information	Section 2.4.9
`-s`	Strip symbolic information	Section 2.4.10
`-S`	Suppress error messages	Section 2.4.15
`-M`	Issue warning for defined common data	Section 2.4.16
`-t`	Suppress size warning for common data	Section 2.4.14
`@` *filename*	Read options from file *filename* (MS-DOS only)	

Table 2-2. VMS Environment Invocation Options

OPTION	EXPLANATION	SECTION
/KEEP	Keep relocation information in executable file	Section 2.4.8
/VERSION	Output linker version information	Section 2.4.17
/INITTABLE	Request an initialization table	Section 2.4.12
/RETAIN	Retain relocation information	Section 2.4.7
/MAP[=*filename*]	Request an output memory map	Section 2.4.5
/DIRECTIVES=*filename*	Specify a directive file	Section 2.4.2
/FILL=*int*	Specify a fill value	Section 2.4.13
/OUTPUT=*filename*	Specify an output filename	Section 2.4.1
/ENTRY=*symbol*	Specify a program entry point	Section 2.4.6
/USYM=*symbol*	Specify an undefined symbol	Section 2.4.11
/STAMP=*int*	Specify a version stamp	Section 2.4.18
/NOLOCAL	Strip local symbolic information	Section 2.4.10
/STRIP	Strip symbolic information	Section 2.4.9
/SILENT	Suppress error messages	Section 2.4.15
/MULDEFS	Issue warning for defined common data	Section 2.4.16
/NOWARNING	Suppress size warning for common data	Section 2.4.14

Chapter 3

THE LINKER DIRECTIVE FILE

3.1 INTRODUCTION

The linker directive file controls certain linker functions. It mainly dictates memory configuration, output section content, and allocation of output sections.

A default linker directive file named `linker.def` is supplied with the GNX software package. It contains predefined directive definitions. In a cross environment, this file resides in the GNX top-level directory and includes the configuration to produce an executable object file for a *Series 32000* development board. In a native environment, this file resides in the *gnxdir*/lib directory (where *gnxdir* is the GNX top-level directory) and includes the configuration to produce an executable object file for a *Series 32000*-based native system.

You can exercise considerable control over the linking operation by creating a directive file that contains directive definitions tailored to your unique needs. This directive file is especially useful for embedded applications. The linker is instructed to use a user-written directive file through the Specify Directive File invocation option (see Section 2.4.2).

This chapter provides an overview of the directive file. Section 3.2 explains the structure. Section 3.3 briefly describes the expressions used in the directive file. The remainder of the chapter defines the various parts of the directive file, providing a detailed description of use, syntax and examples.

3.2 STRUCTURE OF THE DIRECTIVE FILE

A directive file is made up of the following parts:

- Comments. A comment may be used for documentation purposes.
- Input file specifications. An alternative to specifying input files on the invocation line.
- `MEMORY` statements. A `MEMORY` statement is used to define which parts of the memory are available for allocation of output sections.
- `SECTIONS` statements. A `SECTIONS` statement is used to control the construction of output sections from input sections and the allocation of output sections to memory addresses.

- Assignment statements. An assignment statement both defines a symbol and assigns the symbol an absolute address.
- Output file options. Controls certain output file characteristics.

The syntax of the linker directive file is case-insensitive, except when reference is made to filenames or symbols. References to filenames or symbols must obey the rules of the host environment. A UNIX environment is case-sensitive. A VMS environment is case-insensitive.

3.2.1 Example of a Directive File

```
/* Input file specification */
a.o
b.o
c.o


/* Memory configuration */

MEMORY {
     mem1 : origin=0x10000, length=0x8000
     mem2 : origin=0x20000, length=0x8000
}

/* Output section construction and allocation */

SECTIONS {
     .text BIND(0x10000)    : { *(.text) }
     .data INTO(mem2)       : { *(.data) }
     .bss  INTO(mem2) ALIGN(64) : { *(.bss)  }
}

/* Special symbol assignment */

end_bss = ADDR(.bss) + SIZEOF(.bss) ; /* End of .bss section */
```

3.3 DIRECTIVE FILE EXPRESSIONS

Directive file expressions are used as arguments for certain options and as right-hand-side of assignment statements. Expressions consist of integer constants, operators, special functions, and parentheses. The value of a directive file expression is always a 4-byte unsigned integer. The value generally represents a memory address.

Appendix A provides a complete description of directive file expressions.

3.3.1 Examples

```
0x1000
```

`0x1000` is an integer constant having the value of 1000 in hexadecimal.

```
ADDR(.text)+SIZEOF(.text)
```

This is the sum of the start address of the `.text` output section and its size. The result is the address following the end of the .text section.

3.4 COMMENT

A comment can be placed anywhere in the directive file. Comments begin with a slash and asterisk (/*) followed by one or more lines of text. The comment is terminated with an asterisk and slash (*/). Comments cannot be nested.

3.5 INPUT FILE SPECIFICATION

Alternatively to specifying input files on the linker invocation line, you can specify input files in a linker directive file. An input file is any object or library file to be linked.

Input filenames may appear anywhere in the directive file, except within a `MEMORY` or `SECTIONS` statement. The placement is significant because the linker processes input files as they are encountered in the directive file. It is recommended that you place input filenames before the `SECTIONS` directive so that the `SECTIONS` directive will be applicable to all input files.

SYNTAX:

filename

filename is any valid input filename. An input filename may include a full or partial pathname. A filename containing special characters should be enclosed in double-quotes (" ") to avoid conflict with the definition of special characters in the directive file.

3.6 MEMORY STATEMENT

The MEMORY statement is used to specify the configured and unconfigured (i.e. nonavailable) areas of memory. If a MEMORY statement is not specified, the linker assumes the maximum amount of configured memory address space (0x0 through 0xFFFFFFFF).

If one or more MEMORY statements are specified, the linker treats all memory areas not within these statements as unconfigured. Unconfigured memory is not used in the linker allocation process. Therefore, output sections can not be allocated within unconfigured memory.

SYNTAX:

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

mem_name is any symbolic name to be associated with the specified configured memory area.
int is a valid integer constant (in decimal, hexadecimal, or octal format).
attributes are a sequence of one or more of the following attribute letters:

- I – The named memory area is initializable.
- R – The named memory area is readable.
- W – The named memory area is writeable.
- X – The named memory area is executable.

Attribute letters can only be used to direct a section to a memory area with specified attributes (see Section 3.7.3). Attribute letters have no other use.

A configured memory area is a contiguous block of memory. It starts at the address specified by the value given to ORIGIN and contains the number of bytes specified as the value of LENGTH. ORIGIN may be abbreviated to ORG, and LENGTH may be abbreviated to LEN.

Any number of configured memory areas may be declared within one MEMORY statement. However, if more than one memory area is declared, no overlap should exist among the specified areas. A memory area overlap causes the linker to issue an error message and terminate the linking process.

Each memory area can be referenced from the SECTIONS statement either by name or by attribute.

EXAMPLE:

```
MEMORY {
     ROM (R)   : ORIGIN = 0x10000 LENGTH = 0x40000
     RAM (RW)  : ORIGIN = 0x80000 LENGTH = 0x100000
}
```

3.7 SECTIONS STATEMENT

The SECTIONS statement is used to specify how output sections are constructed from input sections, and to allocate memory for output sections.

SYNTAX:

```
SECTIONS {
     output_section_spec [options] : {input_section_spec ... }
     ...
}
```

output_section_spec is a specification of the output section to be created. This is generally the name of the output section.
options are a list of allocation options (Section 3.7.3) or section type options (Section 3.7.5).
input_section_spec is a specification of an input section. This input section is combined with the other specified input sections to produce an output section.

EXAMPLE:

```
SECTIONS {
     .text BIND(0x8000) : { file1.o(.text) file2.o(.text) }
     .data ALIGN(16)    : { file1.o(.data) file2.o(.data) }
}
```

Details of the use of the SECTIONS statement follow below.

3.7.1 Output Section Specification

SYNTAX:

output_section_name [MODULE (*module_name*)]

output_section_name is the name of the output section. You can give an output section any name (up to 8 characters).
module_name is any valid module name.

You can use the MODULE option to associate a module name with an output section. This option only applies to systems that use the *Series 32000* modularity support feature (for further details see the *GNX Language Tools Technical Notes*). If an output section has been associated with a module, only input sections of the same module or input sections which are not associated with any module can be directed into the output section.

3.7.2 Input Section Specification

Input sections are specified in the SECTIONS statement in various ways:

1. Specifying all sections of the input file.

   SYNTAX:

   filename

   filename is any valid input filename. The filename can include a full or partial pathname. A filename containing special characters may be enclosed in double quotes (" ") to avoid conflict with the directive syntax. If *filename* is a library, this specification applies to all library members which have been selected for the linking process.

   EXAMPLE:

   ```
   .xxx : { abc.o }
   ```

   Output section .xxx will consist of all input file abc.o sections.

2. Specifying only certain sections of the input file.

   SYNTAX:

   filename (*section_name* ...)

   EXAMPLE:

   ```
   .text : { file.o(.text .data) }
   ```

   Output section .text will consist of sections .text and .data from file.o.

3. Specifying only certain sections from **all** input files indicated on the invocation line or in the directive file before the SECTIONS statement.

   SYNTAX:

   *(*section_name* ...)

EXAMPLE:

```
.text : { *(.mod) *(.text) }
```

Output section `.text` will consist of sections `.text` and `.mod` from all input files.

4. Specifying the common data from the input file. The common data reside in a linker-created .bss section.

   SYNTAX:

   filename `[COMMON]`

   EXAMPLE:

   ```
   .bss1 : { a.o[COMMON] }
   ```

   Output section `.bss1` will consist of the common data from input file `a.o` (see Section 4.3 for an explanation of common data).

5. Specifying the common data from **all** input files indicated on the invocation line or in the directive file before the `SECTIONS` statement.

   SYNTAX:

   `*[COMMON]`

   EXAMPLE:

   ```
   .bss1 : { *[COMMON] }
   ```

   Output section `.bss1` will consist the common data from **all** input files. files.

6. Specifying the linker-created module table entry of the input file (refer to the *Series 32000 Language Tools Technical Notes*).

   SYNTAX:

   filename `[MOD]`

   EXAMPLE:

   ```
   .mod1 : { a.o[MOD] }
   ```

   Output section `.mod1` will consist of the linker-created module table entry of input file a.o.

7. Specifying the linker-created module table entry of **all** input files indicated on the invocation line or in the directive file before the `SECTIONS` statement.

SYNTAX:

```
*[MOD]
```

EXAMPLE:

```
.mod : { *(.mod) *[MOD] }
```

Output section `.mod` will consist of .mod sections from all input files (user-defined module table entries) and linker-created .mod sections (linker-created module table entries).

8. Specifying the initialization table created by the linker. The initialization table resides in a linker-created .init section (see Section 5.4 for an explanation of data initialization).

   SYNTAX:

   ```
   *[INIT]
   ```

   EXAMPLE:

   ```
   .text : { *(.text) *[INIT] }
   ```

   Output section `.text` will consist of both the .text sections from all input files and the linker-created .init section (which contains an initialization table).

3.7.3 Allocating a Section to Memory

Output sections can be allocated to memory by using allocation options in the following ways:

1. Binding a section to a particular memory address by using the `BIND` option. This instructs the linker to assign a configured memory address to the specified output section.

   SYNTAX:

   `BIND(`*expression*`)`

   expression is any valid linker expression (refer to Appendix A for a complete description of linker expressions). The expression value is a memory address to which the output will be bound.

EXAMPLE:

```
.text BIND(0x10000) : { *(.text) }
```

The .text output section will be allocated at address 0x10000.

2. Directing a section to a memory area by name using the INTO option. This instructs the linker to assign a memory address within the memory area to the specified output section. The output section must fit in the memory area.

SYNTAX:

INTO(*mem_name*)

mem_name is a name that has been associated with a configured memory area through use of the MEMORY directive (see Section 3.6).

EXAMPLE:

```
MEMORY {
     ROM : origin=0x1000 length=0x2000
     RAM : origin=0x10000 length=0x80000
}

SECTIONS {
     .text INTO(ROM) : { *(.text) }
     .data INTO(RAM) : { *(.data) }
     ...
}
```

The .text output section will be allocated within the ROM memory area as defined with the MEMORY statement. The .data output section will be allocated within the RAM memory area.

NOTE: For compatibility with older linker versions, you can also direct a section to memory by using the > option. It must be placed after the braces enclosing the input section list. This is equivalent to using the INTO option.

SYNTAX:

> *mem_name*

EXAMPLE:

```
.text : { *(.text) } > RAM
```

The .text output section will be allocated within the RAM memory area.

3. Directing a section to memory by attributes by using the INTO option. The INTO option instructs the linker to assign a memory address, within any memory area having the listed attributes, to the specified output section.

SYNTAX:

```
INTO( (attributes) )
```

attributes is a sequence of attribute letters (I, R, W, X, meaning respectively init, read, write and execute).

EXAMPLE:

```
MEMORY {
     ROM1 (R) : origin=0x1000 length=0x2000
     ROM2 (R) : origin=0x8000 length=0x2000
     RAM1 (RW) : origin=0x10000 length=0x20000
     RAM2 (RW) : origin=0x80000 length=0x20000
}

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

The .text output section will be allocated within a memory area that has only read attributes (ROM1 or ROM2). The .data output section will be allocated within a memory area that has read and write attributes (RAM1 or RAM2).

4. Binding or directing a ROM copy of a section to memory by using the ROMBIND and ROMINTO options. These options are equivalent to the BIND and INTO options, respectively. Use of the ROMBIND or ROMINTO option instructs the linker to allocate memory for a ROM copy of the specified output section. The output section will therefore have two addresses: a ROM address and a RAM address. This can be used for data initialization (see Section 5.4 for details).

SYNTAX:

```
ROMBIND (expression)
ROMINTO (mem_name)
ROMINTO( (attributes) )
```

expression is any valid linker expression.
mem_name is a name that has been associated with a configured memory area through use of the MEMORY directive.
attributes is a sequence of attribute letters (I, R, W, X, meaning respectively init, read, write and execute).

EXAMPLES:

```
.data BIND(0x10000) ROMBIND(0x800000) : { *(.data) }
```

The .data output section will be allocated at address 0x10000. A copy of the .data section will be allocated at address 0x800000. This copy is used only for initialization purposes. At run-time the .data section will be copied by an initialization routine from address 0x800000 (ROM address) to its actual (RAM) address 0x10000.

```
MEMORY {
     ROM : origin=0x1000 length=0x2000
     RAM : origin=0x10000 length=0x80000
}

SECTIONS {
     .text INTO(ROM) : { *(.text) }
     .data INTO(RAM) ROMINTO(ROM) : { *(.data) }
     ...
}
```

The .text output section will be allocated within the ROM memory area as defined with the MEMORY statement. The .data output section will be allocated within the RAM memory area, and a copy of the .data section will be allocated within the ROM memory area. This copy is used only for initialization purposes.

3.7.4 Aligning a Section

Aligning an output section to an alignment value ensures that the output section will be assigned a memory address that is a multiple of the value. The ALIGN option is used for this purpose.

SYNTAX:

ALIGN(*expression*)

expression is any valid linker expression (Appendix A).

NOTE: The ALIGN option is ignored when it appears in conjunction with the BIND or ROMBIND allocation options, because the allocation options are specified with a particular address.

EXAMPLES:

```
.text ALIGN(16) : { *(.text) }
```

The .text output section will be allocated anywhere within available configured memory but its address must be a multiple of 16.

```
.text INTO(RAM) ALIGN(16) : { *(.text) }
```

The .text output section will be allocated within the memory area RAM and its address will be a multiple of 16.

3.7.5 Setting the Section Type

Section type information is stored in the section header of the COFF file. This information indicates how the section is to be handled by the linker and the debugger/operating system, 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 groups of section type options are available to control how the output section is processed and to specify the contents of an output section.

SYNTAX:

(*type_option*)

type_option is any valid section type option from the following groups:

1. Options which control the processing of output sections
   - DSECT
   - NOLOAD
   - COPY
   - INFO
   - OVERLAY
   - LIB
2. Options that specify the contents of an output section

- TYP_TEXT
- TYP_DATA
- TYP_LINK
- TYP_BSS
- TYP_MOD

For a detailed definition of these options see the GNX COFF Programmer's Guide.

EXAMPLE:

```
.mod (NOLOAD) BIND(0x10000) : { *(.mod) }
```

The .mod output section will be a NOLOAD section.

3.7.6 Grouping Output Sections

Several output sections may be grouped to create a contiguous block of memory by using the GROUP option. This allows you to allocate consecutive sections without having to specify an allocation option for each section. Although the output sections are grouped together in memory, they remain separate.

SYNTAX:

```
GROUP [ group_options ] : {
      output_section_spec [ type_option ] : {input_section_spec ... }
}
```

group_options are the allocation options BIND, ROMBIND, INTO and ROMINTO, and the option ALIGN.
output_section_spec is a specification of the output section to be created.
type_option is any section type option (see Section 3.7.5 for the list of these options).
input_section_spec is a specification of an input section.

EXAMPLE:

```
.text BIND(0x8000) : { *(.text) }
GROUP BIND(0x10000) : {
      .data : { *(.data) }
      .bss  : { *(.bss)  }
}
```

The .data and .bss output sections will be grouped to a contiguous block of memory starting at address 0x10000.

NOTE: Output sections that are specified within a `GROUP` option may be qualified only by section type options, since all other options are specified in `GROUP` level.

3.8 ASSIGNMENT STATEMENT

Symbols can be defined and assigned a memory address at link-time through use of the assignment statement. This is useful for two reasons:

- To use link-time computed information at run-time. You can assign a symbol an expression that is calculated by the linker, and then use it in your program.
- To bind a symbol to an address in a flexible way. If you define a symbol in a directive file and later want to change its address, you simply change the assignment in the directive file and re-link. Therefore there is no need to recompile your program.

SYNTAX:

symbol = *expression* ;

symbol is any valid symbol name.
expression is any valid linker expression.

The syntax supports other assignment operators in addition to "=" (a complete list of assignment operators can be found in Appendix A).

Since the assignment of symbols to a memory address is made at the end of the linker allocation phase, the linker does not recognize addresses assigned previously. Therefore, the placement of the assignment statement within a directive file is not important, and may be placed anywhere within the directive file.

Note that the linker does not check that the memory address assigned to a symbol is within configured memory.

EXAMPLES:

```
abc = 0x1000 ;
```

A symbol named `abc` will be defined and its address will be 0x1000.

```
sdata = ADDR(.data)
```

A symbol named `sdata` will be defined and its address will be the start address of the .data output section.

3.8.1 Symbol Assignment Within SECTIONS Statement

A symbol assignment may also be used within a SECTIONS statement. Such an assignment can use the symbol "." to denote the current location in memory. This assignment should appear as part of the input section specification list.

EXAMPLE:

```
.text : { *(.text) xxx = . ; }
```

A symbol named xxx will be defined and assigned the end address of the .text output section (the end address is the value of the current location of the assignment).

Note that this symbol assignment can also be specified outside the SECTIONS statement:

```
xxx = ADDR(.text) + SIZEOF(.text) ;
```

3.8.2 Creating Gaps Within An Output Section

The current location symbol itself may be assigned a value. This can be used to create a gap within an output section. The linker normally combines input sections in a contiguous fashion when creating an output section. However, by incrementing the value of the current location you can create a gap of unallocated space.

SYNTAX:

. += *expression*

expression is any valid linker expression.

EXAMPLE:

```
.text : { a.o(.text)   . += 0x1000 ;  b.o(.text) }
```

The .text output section will consist of the .text section of file a.o, a gap of of 0x1000 (created by the current location assignment), and the .text section of file b.o.

By default, the linker fills the gaps created within an output section with zeros or with a value specified with the Specify Fill Value invocation option (see Section 2.4.13). However, you can specify a fill value for a specific output section. This overrides any other specified fill value.

SYNTAX:

output_section_name [*options*] : {*input_section_spec ...* } = *fill_value*

fill_value is a two-byte integer constant.

EXAMPLE:

```
.text : { a.o(.text)  . += 0x1000 ;  b.o(.text) } = 0xffff
```

Each word in the gap created inside the .text output section will contain the value 0xffff (i.e. the gap will be filled with 1's).

3.9 OUTPUT FILE OPTIONS

Three characteristics of the output file can be changed through use of output file options: the default filename, the default execution permission set, and the header magic number.

3.9.1 Change Default Filename and Permission

By default, the linker produces an executable file for the cross environment named `a32.out` that does not have executable permission. To override this default for the native environment, the following statement can be included anywhere within the directive file:

```
OPTION NATIVE
```

This results in a default output file named `a.out`. This option also gives executable permission to the output file. Note that in native environment, the user-written directive file must be set up to create an output file that compiles with this environment's conventions.

3.9.2 Optional Header Magic Number

The optional header magic number in a COFF file provides memory loading information to the operating system in a native environment. By default, the linker sets the optional header magic number to 0417 (octal). To override this default, the following statement should be included anywhere within the directive file:

```
OPTION OMAGIC int
```

int is any 16-bit integer constant.

Chapter 4

RESOLUTION OF SYMBOLIC REFERENCES

4.1 INTRODUCTION

A symbol is used either to mark a program location or to represent a data element. In high-level languages, symbols represent variables, functions or labels.

An external symbol is a symbol that can be referenced from any object file. The definition (defining point) of a symbol is a source program statement which associates the symbol with an explicit location in a section of the object file. A symbol can be defined only in one object file.

A symbolic reference is the use of a symbol in a statement that is not its definition. An external symbolic reference is a reference to a symbol which is defined in another object file.

4.1.1 Examples of Symbol Definition and Reference

In the assembly language:

```
.globl abc
```

The .globl assembler directive is used to declare the symbol `abc` as external.

```
xxx:
```

This assembly language label defines the symbol `xxx` since it is associated with an explicit location within one of the sections.

```
yyy::
```

A label, followed by a double colon, defines an external symbol. This example is equivalent to:

```
yyy:
.globl yyy
```

The first statement defines the symbol `yyy`. The second statement declares it as external.

```
movd sym,r0
```

This is a reference to the symbol `sym`. If `sym` is not defined in the same program, the assembler considers this an external symbolic reference.

In the C language:

```
int i;
int j = 5;
main {
extern k;
     ...
}
```

The variable `i` is an external symbol since it is declared outside any function. `k` is also an external symbol because it is declared as such (by the term "extern"). And `j` is a defined external symbol since it is initialized at its point of declaration. This initialization associates `j` with a location within the .data section.

4.1.2 Symbol Resolution Using the Symbol Table

The COFF object file contains a symbol table. The symbol table is comprised of information about symbols defined or referenced in the source program. If a symbol was defined in the source program, it will have a "defined" status in the object file's symbol table entry. If a symbol is only referenced in the source program (and not defined), it will have an "undefined" status in the object file's symbol table entry.

The resolution of symbolic references is the process by which the linker matches an external symbolic reference with its definition. It does so by using the information in the symbol tables of the object files. If a symbol has the status "undefined" in one object file, the linker searches for the symbol's definition in the other object files. The linker checks that no symbol is defined more than once, and that there is no symbol left undefined. If such a symbol is found, the linker issues an error message and terminates the linking process.

4.1.3 Example

```
object_1

a : defined
b : defined
c : undefined
```

```
object_2

a : undefined
b : defined

d : defined
```

When linking the two object files, the symbols `a` and `d` are resolved correctly by the linker. `a` is defined in `object_1` and appears in `object_2` as undefined. `d` appears only in the object file in which it is defined. The linker will issue an error message about symbols `b` and `c`, since `b` is defined twice and `c` is never defined.

4.2 LIBRARY PROCESSING

The resolution of external symbolic references to a library member involves a slightly different process. A library member becomes part of the linker's input only if it contains a definition of an external symbol that has been referenced in a previous input file or library member. This is unlike regular object files, which are completely included in the linker's input.

A library file is a collection of object files typically containing useful routines. When the GNX archiver (see the *GNX Commands and Operations Manual*) builds a library from object files, it creates a symbol directory as the first member of the library. The library symbol directory is a list of all defined external symbols found in the library members. For each such symbol, there is a pointer to the library member where the symbol is defined. When the linker processes a library it scans the symbol directory, selecting the definitions that resolve currently undefined external symbols.

When a library member is selected for the linking process, it may create new external symbolic references (for example, one library member can refer to a symbol which is defined in another library member). For this reason, the linker will scan the symbol directory of a library repeatedly until the the definitions in the symbol directory no longer resolve external symbols (i.e. all references to library members have been resolved in previous passes). Therefore, for efficiency of the linking process, the ordering of library members should be such that a library member containing a reference to another library member should be place first in the library. The GNX `lorder` utility can be used to calculate the best ordering for library members (see the *GNX Commands and Operations Manual*).

4.2.1 Example

Consider a main program that has only one external symbolic reference: a call to the `malloc` routine found in the GNX library libc.a. The `malloc` routine itself calls three additional routines from libc.a: `bcopy`, `getpagesize` and `sbrk`. The program is linked by

```
nmeld main.o -lc                        (UNIX)
nmeld main.obj,gnxdir:libc.a            (VMS)
```

The library will be processed depending on the ordering of the symbol directory. Assume that the ordering of the symbols is `...bcopy...`

malloc...getpagesize...sbrk.... The linker processes the symbol directory as shown below:

Pass	Symbol Processed	Resolves a Reference	Current Unresolved References
0			malloc
1	bcopy	no	malloc
1	malloc	yes	bcopy, getpagesize, sbrk
1	getpagesize	yes	bcopy, sbrk
1	sbrk	yes	bcopy
2	bcopy	no	
2	malloc	no	
2	getpagesize	no	
2	sbrk	no	

Now assume that the ordering of the symbols is ...malloc...bcopy... getpagesize...sbrk... . The linker processes the symbol directory as shown below:

Pass	Symbol Processed	Resolves a Reference	Current Unresolved References
0			malloc
1	malloc	yes	bcopy, getpagesize, sbrk
1	bcopy	yes	getpagesize, sbrk
1	getpagesize	yes	sbrk
1	sbrk	yes	

As can be seen, in the second case one pass is sufficient to resolve all external references to the library.

4.3 COMMON DATA PROCESSING

Common data refers to external symbols with a special attribute. This attribute instructs the linker to not terminate the linking process even if the symbol definition is not in any object file.

All common data is associated with a special linker created .bss section. The size of this section is based on the combined size of all the common data. When the linker

consolidates common data, it checks that different references to the same symbol indicate the same size. If the references indicate data of different sizes, the linker issues a warning message and allocates space in memory according to the largest reference.

You can specify an input section that contains common data by use of the `SECTIONS` statement's [COMMON] notation in the directive file.

4.3.1 Examples

In the assembly language:

```
.comm abc,4
```

The .comm assembler directive is used to declare the symbol `abc` as a common external symbol with a size of 4 bytes.

In the C language:

```
int i;
main() {
...
}
```

The variable `i` is a common external symbol. In C, any variable that is declared outside a function, is uninitialized, and is not preceded by the `extern` modifier, is by default considered as a common external variable.

In the Fortran language:

```
COMMON /CCC/ ABC,DEF
```

The symbol `CCC` (the name of the common area) is a common external symbol. Its size is the sum of the sizes of variables `ABC` and `DEF`.

4.4 SYMBOL DEFINITION IN THE DIRECTIVE FILE

Normally, the definition of an external symbol is found in one of the input files. However, you can also define a symbol at link time through use of the assignment statement in the directive file (Section 3.8). This creates an external symbol and associates it with an absolute address.

4.5 LINKER DEFINED SYMBOLS

Certain special symbols are referenced in useful routines and have a universal use. These symbols have a default definition and value that is automatically assigned by the linker. You can override the default definition and value of special symbols by supplying your own definition (either in the source program or in the linker directive file). These symbols are:

Symbol Name	Meaning	Default Value
_etext	end address of .text section	end address of .text section
_edata	end address of .data section	end address of .data section
_end	start of heap	next address after highest allocated memory address
_HEAP$_START	start of heap	next address after highest allocated memory address
_HEAP$_MAX	heap limit address	highest address in configured memory
_ _STACK_START	initial top of stack	0xffffffff
_ _INIT_TABLE	address of initialization table	address of linker-created .init section

4.5.1 Example

In the assembly language:

```
lprd sp,$__STACK_START
```

The symbol `__STACK_START` is used to initialize the stack pointer (sp).

Chapter 5

ALLOCATION OF OUTPUT SECTIONS

5.1 INTRODUCTION

The allocation process of the linker takes place after all input files have been read and all external symbolic references have been resolved. This process includes constructing output sections from input sections and assigning memory addresses to each output section. The linker directive file can be used to exercise considerable control over the allocation of memory.

5.2 CREATING OUTPUT SECTIONS FROM INPUT SECTIONS

Each output section is constructed from one or more input sections. The list of input sections to be combined to produce an output section is determined in two ways:

1. Through the user-specified `SECTIONS` statement in the directive file (Section 3.7).
2. By the default linker rule.

The default linker rule applies to input sections that have not been associated with an output section through use of the `SECTIONS` statement. Such input sections are associated with an output section that has the same name. If there is no output section with the same name, the linker creates a new output section with the input section name and associates the input section with the newly created output section.

5.2.1 Example

Consider two input files, a.o and b.o, each containing the input sections .text, .data, .bss, .mod, and .link. For the following `SECTIONS` statement:

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

Output sections will be constructed as follows (Reason 1 indicates a user specification; Reason 2 indicates the linker default rule):

Output section	Contents	Reason
.text	.text of a.o	1
	.text of b.o	1
.data	.data of a.o	1
	.link of a.o	1
	.data of b.o	2
.bss	.bss of a.o	2
	.bss of b.o	2
.mod	.mod of a.o	2
	.mod of b.o	2
.link	.link of b.o	2

NOTE: When using the *Series 32000* modularity support feature some input sections are associated with a module name (for further details see the *GNX Language Tools Technical Notes*). Such input sections cannot be combined with input sections that are associated with a different module name. They can be combined with input sections that are associated with the same module name or with input sections that are not associated with any module name.

5.3 ASSIGNING AN ADDRESS TO AN OUTPUT SECTION

Before assigning an address to an output section, the linker determines which parts of memory are available for allocation. By default, the linker assumes that the maximum amount of configured memory space, 0x0 through 0xFFFFFFFF, is available for allocation. However, you can (and should) specify the areas of memory to be configured, and therefore available for allocation, through use of the `MEMORY` statement in the directive file (Section 3.6).

There are four phases in the allocation process (see Section 3.7.3):

1. The linker processes all the `BIND` options used in the `SECTIONS` statement. The `BIND` option has the highest priority in determining output section addresses.
2. The linker processes all the `INTO` options of the `SECTIONS` statement to direct output sections to memory areas by name.
3. The linker processes all the `INTO` options of the `SECTIONS` statement to direct output sections to memory areas by attributes.
4. The linker assigns memory addresses to all unallocated output sections using a find-first-fit algorithm.

If the linker cannot process any one of the above phases, it issues an error and terminates the linking process.

5.3.1 Example

Consider the following directive file:

```
MEMORY {
     MEM1 (R)   : ORIGIN = 1000 LENGTH = 1000
     MEM2 (RW)  : ORIGIN = 3000 LENGTH = 1000
}

SECTIONS {
     .text INTO(MEM1) : { *(.text) }
     .data BIND(3500) : { *(.data) }
     .bss             : { *(.bss)  }
     .mod INTO((R))   : { *(.mod)  }
}
```

Assume that the size of the output section of .text is 500, of .data is 400, of .bss is 500, and of .mod is 32.

The steps in the allocation process are as follows:

1. Output section .data is allocated at address 3500, as specified in the `BIND` option.
2. Output section .text is allocated within memory area MEM1. Since this area is empty, the .text section will be allocated at its starting address 1000.
3. Output section .mod should be allocated within a memory area with an R attribute. The only memory area which has an R attribute is MEM1. This section is allocated right after the .text section, at address 1500.
4. Since there is no allocation option specified for the .bss output section, it will be allocated in the first memory address where it fits. Though the .bss section does not fit in memory area MEM1, it does fit in the memory area MEM2. Therefore the .bss section will be allocated at the starting address of MEM2, 3000.

5.4 DATA INITIALIZATION SUPPORT

The linker can be used to support data initialization for an embedded environment. The two kinds of initializations are:

- Variables that are initialized at their point of declaration rather than by assignment at run-time.
- Variables that are uninitialized are automatically initialized to zero.

To implement the first kind of initialization, you must duplicate the section that contains the initialized data (typically the .data section). This is done through use of the `ROMBIND` and `ROMINTO` output section options (Section 3.7.3). These options instruct the linker to assign a second address to the output section (a ROM address). The output section should be burned on ROM and copied to RAM at run-time, in order for it to

be writable. The output section is thus allocated twice (on ROM and RAM). References to symbols that are defined in duplicated sections are modified by the linker according to the section RAM address, where it resides at run-time.

To implement the second kind of initialization, the sections that contain uninitialized data (typically the .bss section) are initialized to zero at run-time.

When invoked with the Request Initialization Table invocation option (Section 2.4.12) the linker generates an initialization table. This table can be used by an initialization routine. The initialization table provides two types of entries:

- One kind of entry for each duplicated section. The information in this kind of entry may be used to copy sections from ROM to RAM.
- One entry for each section of uninitialized data (typically .bss). The information in this kind of entry may be used to initialize sections.

The structure of an initialization table is:

Type	Size	Name	Description
string	8	i_secname	Section name
unsigned integer	4	i_srcaddr	Section source (ROM) address. N/A for sections of uninitialized data and contains 0xffffffff in that case.
unsigned integer	4	i_trgaddr	Section target (RAM) address
unsigned integer	4	i_size	Section size
unsigned integer	4	i_type	Section type (identical to COFF section header flags).

The C structure declaration for this table may be found in the `inittab.h` header file supplied with the GNX package. The initialization table generated by the linker resides in a linker-created .init input section (see Section 5.5).

5.4.1 Example

Consider a simple C program that uses initialized and uninitialized variables:

```
int i = 5 ;
int j ;
main()
{
    if ( i == 5 && j == 0)
     printf("PASSED \n") ;
    else
     printf("FAILED \n") ;
}
```

Initialized variables are allocated in a .data section. Uninitialized variables are allocated in a .bss section. In order to run this program correctly in an embedded environment, data initialization is necessary. The .data section should be burned on ROM and copied to RAM before program execution. The .bss section should be filled with zeroes. By using the linker data initialization support, this process is simpler. A sample directive file for linking the program is:

```
MEMORY {
 ROM : ORG = 0x80000 LEN = 0x20000
 RAM : ORG = 0x10000 LEN = 0x40000
}

SECTIONS {
 .text INTO(ROM) : { }
 .data ROMINTO(ROM) INTO(RAM) : { }
 .bss INTO(RAM) : { }
 .init INTO(ROM) : { }
}
```

The .text section that contains the program code is directed to the ROM memory area.

The .data section is duplicated, and directed to the RAM and ROM memory areas. Note that all references to the .data section will be to the the RAM copy. The ROM copy is used only for initialization purposes and is not used after initialization is completed.

The .bss section is directed to the RAM memory area.

The .init section, which contains the initialization table, is directed to the ROM memory area.

The program is now ready for linking. In order to take advantage of the data initialization support the Request Initialization Table invocation option must be used. This option is `-i` on UNIX and `/INITTABLE` on VMS. The initialization table can then be

used by to perform the required initializations. The crt0 initialization routine, provided as part of the GNX package, can be used to perform the initialization. The invocation line for the above program is:

nmeld *gnxdir* /lib/crt0.o main.o -lc -i -o prog (UNIX)

NMELD GNXDIR:CRT0.OBJ,MAIN.OBJ,GNXDIR:LIBC.A /INIT /OUT=PROG.EXE (VMS)

where *gnxdir* is the top-level directory of the GNX package.

The .data section should now be burned to ROM. This is done by using the nburn utility (see *The Commands and Operations Manual*). A sample invocation of nburn (assuming 64K EPROMs) is:

nburn -x0x80000 -l64 prog -o prog.hex (UNIX)

NBURN /VADDRESS=0x80000 /PROMSIZE=64 PROG.EXE /OUTPUT=PROG.HEX (VMS)

You can write your own initialization routine. A sample C initialization routine is:

```
#include <inittab.h>
#define UNDEF 0xffffffff
init()
{
    extern INITTAB _INIT_TABLE[] ;  /* _INIT_TABLE is defined by the Linker */
    INITTAB *inittab_entry ;        /* a pointer to the current init table
                                       entry */

    for (inittab_entry = _INIT_TABLE;       /* start of init table */
    inittab_entry->i_secname[0] != 0;      /* last entry is a null one */
         inittab_entry++) {

         if (inittab_entry->i_srcaddr == UNDEF)
            /* No source address - its an uninitialized data (bss) area.
               Should be set to zero.
            */
            memset(inittab_entry->i_trgaddr, ' ', inittab_entry->i_size) ;
          else
            /* It's an initialized data area - copy from ROM to RAM */
            memcpy(inittab_entry->i_trgaddr,
                   inittab_entry->i_srcaddr,
                   inittab_entry->i_size) ;
    }
}
```

5.5 LINKER CREATED INPUT SECTIONS

Normally, input sections are a part of input files. However, the linker sometimes creates "dummy" input sections. These dummy input sections are added to the input section list and participate in the allocation process like any other input section. Dummy input sections are created when:

- the linker allocates common data. The linker creates a .bss input section (Section 4.3).
- the linker creates module table entries for the *Series 32000* modularity support feature. The linker creates a .mod input section (Section 3.7.1).
- the linker creates an initialization table for data initialization purposes. The linker creates a .init input section.

During the allocation process, the linker treats these input sections like all other input sections. The dummy input sections can be identified in the linker memory map by the term "linker_defined" that replaces the filename for these sections (see Section 5.6 below).

5.5.1 Example

Consider two C programs `a.c` and `b.c`. For the program `a.c`:

```
int i ;
static int j ;
main()
{
    ...
}
```

For the program `b.c`:

```
int i ;
static int j ;
foo()
{
    ...
}
```

Each of the `j` variables is local to its module and allocated by the compiler/assembler to a local .bss section. The `i` variables are common external symbols which are not allocated at compile-time. Instead they are consolidated by the linker and allocated to a linker-created .bss section. This .bss section is added to the input section list. The linker will thus have three .bss input sections: two from each of the object files and one that the linker has created. All three are input for the linker allocation process.

5.6 MEMORY MAP

The following information appears on the memory map (all address and size values are in hexadecimal).

- Output section, lists each output section in the order it appears in memory. For those output sections that have been duplicated, the ROM copy will be denoted by (R) next to the output section name.
- Input section, lists each input section that was linked to produce the specified output section.
- Memory address, denotes the starting address in memory of a particular input or output section.
- Size of section, lists the total size of the output section and the individual size of each input sections.
- Section contents, specifies the input file from which the input section originated. This is either an object file or a library file. The term "linker_defined" is used to indicate that the section was created by the linker.

The term "fill space" may appear in the section contents column, and indicates a gap created in the output section through use of the current location symbol assignment of the directive file (Section 3.8.2). The term "UNUSED" refers to unallocated or unconfigured memory.

5.6.1 Example

```
output          input           memory          size     section
section         section         address                  contents

.text                           e000             55c
                .text           e000              30     main.o
                .text           e030             404     libc.a:malloc.o
                .text           e434              24     libc.a:bcopy.o
                .text           e458              18     libc.a:getpagesize.c
                .text           e470              ec     libc.a:sbrk.o

UNUSED          e55c     to     f000

.data                           f000              d4
                .data           f000              10     main.o
                .data           f010              60     libc.a:malloc.o
                .data           f070               0     libc.a:bcopy.o
                .data           f070              60     libc.a:getpagesize.c
                .data           f0d0               4     libc.a:sbrk.o

.bss                            f0d4              8c
                .bss            f0d4               0     main.o
                .bss            f0d4              80     libc.a:malloc.o
                .bss            f154               0     libc.a:bcopy.o
                .bss            f154               0     libc.a:getpagesize.c
                .bss            f154               0     libc.a:sbrk.o
                .bss            f154               c     linker_defined

UNUSED           f160    to     ffffffff
```

Chapter 6

RELOCATION OF MEMORY ADDRESS

6.1 INTRODUCTION

Once external symbolic references have been resolved and output sections allocated to memory, the linker calculates the final addresses of symbolic references. The linker also modifies the contents of the holes within the code or data. Holes are small pieces of code or data that are based symbolic references. A hole must be modified to reflect the new address of the symbolic reference on which it is based. This process includes not only external symbolic references, but also other symbolic references that need to be updated (for example, a reference that uses absolute addressing).

6.2 RELOCATION INFORMATION

Relocation information is part of the COFF file. The assembler creates a relocation table for each section of the file. Each relocation table entry provides information about a hole that should be updated as a result of link-time section allocation. This hole may be a machine instruction operand or data (typically address constants, module table entries, etc.).

Each relocation entry consists of

- An address of the hole that should be updated.
- A pointer to the symbol that is associated with the reference.
- The hole type (format, size, and semantic).

Refer to the *COFF Programmer's Guide* for further information.

6.2.1 Example

Consider the following assembly program:

```
bsr foo
movd r0,abc
.data
abc: .double 5
```

The first two instructions must be relocated at link-time.

The first instruction, `bsr foo`, refers to an external symbol (`foo`) whose address is unknown at assembly-time. The assembler encoding for this instruction is

```
02 c0 00 00 00 (bsr *+0)
```

The relocation entry created by the assembler contains the following information:

- A hole address that points to the instruction's second byte (the location of the instruction operand).
- The index of the symbol `foo` in the symbol table.
- The hole type (the hole size is 4 bytes, and the hole is pc-relative).

The second instruction, `movd r0,abc`, has a reference to the local symbol `abc`. However this reference is in absolute addressing mode, and absolute addresses are unknown at assembly time. The assembler encoding for this instruction is

```
57 05 c0 00 00 0c (movd r0,@12)
```

The relocation entry created by the assembler contains the following information:

- A hole address that points to the instruction's third byte (the location of the instruction's second operand).
- The index of the symbol `.data` in the symbol table. The symbol `abc` is already allocated in the .data section and its final address will be updated according to the final address of the .data input section. The address of the symbol `.data` is also the address of the .data input section.
- The hole type (the hole size is 4 bytes, and the hole is an absolute address).

6.3 THE RELOCATION PROCESS

During the relocation process, the linker scans the relocation table of each input section. For each relocation entry, the linker calculates the new value of the hole. The calculation is based on the referenced symbol's new address, on the new address of the hole (only when the hole is a pc-relative operand), and on the type of the hole found in the relocation entry. The new value of the hole should fit in the size allocated to it, otherwise the linker issues an error message. After the new value of the hole has been calculated, the linker updates the hole with this new value.

6.3.1 Example

Using the example in Section 6.2.1, two holes are modified. Assume that the final address of the symbol `foo` is 0x10000, and that the final address of the `bsr` instruction is 0xe000. The displacement for the `bsr` instruction should thus be modified to 0x10000-0xe000=0x2000. The operand is modified by the linker and the new instruction encoding is:

```
02 c0 00 20 00
```

Assume now that the final address of the symbol `abc` is 0xf000. The second operand of the `movd` instruction is modified to this absolute address. The new instruction encoding is:

```
57 05 c0 00 f0 00
```

Appendix A

DIRECTIVE FILE EXPRESSIONS

A.1 INTRODUCTION

Directive file expressions are used as arguments for certain options and as right-hand-side of assignment statements. Expressions consist of integer constants, operators, special functions, and parentheses. The value of a directive file expression is always a 4-byte unsigned integer. The value generally represents a memory address.

Expressions are used in two places in the directive file:

- As arguments to the `BIND`, `ROMBIND` and `ALIGN` options and to the `NEXT` function.
- As the right-hand side of an assignment statement.

A.2 INTEGER SYNTAX

The linker accepts three radices for unsigned integer input: decimal (the default), hexadecimal, and octal. Integer input in the directive file syntax is denoted by the word *int*. Unless otherwise noted, *int* represents a 4-byte unsigned integer.

A.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.

A.2.2 Octal Value Syntax

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

A.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).

A.3 UNARY OPERATORS

The linker supports the following unary operators:

1. Logical negation
2. One's complement
3. Two's complement

A unary operator has a higher precedence than a binary operator in expression evaluation.

A.4 BINARY OPERATORS

The linker supports the following binary operators (listed in order of precedence):

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

A.5 ASSIGNMENT OPERATORS

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.

A.6 SPECIAL FUNCTIONS

Five special functions provide useful information about output sections and memory addresses. These functions and the information they return are listed in Table A-1 below.

Table A-1. Special Functions

FUNCTION	RETURNED VALUE
SIZEOF	Size of a specified output section
ADDR	Memory address of a specified output section
FILEADDR	File address of a specified output section
NEXT	Next memory address aligned to a specified value
HIGHMEMADDR	Next address after highest allocated memory address

A.6.1 Size of Output Function

The size of the output 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, otherwise it returns a zero.

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

A.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 for the first section recognized.

A.6.3 File Address Function

The file address function returns the file address of a section's raw data in the 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 output file, otherwise it returns zero.

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

A.6.4 Next Address Function

The next address function returns the next available memory address (i.e. after the most recently allocated output section), which is a multiple of a specified value. The syntax for the next function is

```
NEXT ( int )
```

int must be greater than zero.

A.6.5 Highest Memory Address Function

The highest memory address function returns the next available memory address after the highest address that has been allocated in memory. The syntax for the highest memory address is

```
HIGHMEMADDR
```

Appendix B

LINKER ERROR MESSAGES

B.1 INTRODUCTION

This appendix contains a list of all linker error messages. There are five types of error messages:

- System Error - the result of an incorrect call to the operating system. A system error will cause immediate termination of the linking process. An explanation of the error follows the error message.
- Warning - A warning error has no impact on the linking process.
- Severe Error - Severe errors accumulate, and eventually result in the termination of the linking process.
- Fatal Error - A fatal error will cause an immediate termination of the linking process.
- Internal Error - An internal error is caused by an internal problem in the linker. If you should encounter an internal error please contact National Semiconductor immediately. Internal error messages are not listed in this appendix.

The error messages are listed in alphabetical order. The error message type and an explanation is also given.

B.2 ERROR MESSAGES

`Cannot close file` *filename*

Type: System

Explanation: An error has been detected when closing the file.

`Cannot open default directive file` *filename*. `Proceeding with default linker processing`

Type: Warning

Explanation: The default directive file cannot be opened. The linker therefore uses a default allocation process, which assumes that the maximum amount of

configured memory space is available and allocates output sections contiguously from address 0.

`Cannot open specified directive file` *filename*

Type: System

Explanation: The user-specified directive file cannot be opened.

`Cannot open input file` *filename*

Type: System

Explanation: The input object or library file cannot be opened.

`Cannot open output file` *filename*

Type: System

Explanation: The output file cannot be opened with write permission.

`Common symbol` *symbolname* `has an explicit definition`

Type: Warning

Explanation: The common symbol which appears in one or more input files is defined in another input file. This is not an error. All references to the symbol are resolved according to its definition. Common symbols without a definition are consolidated and allocated memory space by the linker.

`Common symbol` *symbolname* `multiply declared with differing sizes. Larger size used`

Type: Warning

Explanation: The references to the symbol in various input files have differing size specifications. The linker uses the largest size specified for allocation.

`Directive file MEMORY statement error: overlapping memory areas` *mem1* `and` *mem2*

Type: Fatal

Explanation: An error has been detected in the memory configuration as specified in the directive file MEMORY statement. Two memory areas overlap.

Directive file BIND/ROMBIND option error: address *addr* (specified for output section *secname*) is already allocated to another output section

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The specified argument address of the BIND/ROMBIND option is not available since it is already allocated to another output section.

Directive file BIND/ROMBIND option error: address *addr* (specified for output section *secname*) is not in configured memory

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The specified argument address of the BIND/ROMBIND option is not available since it is not within configured memory.

Directive file BIND/ROMBIND option error: output section *secname* does not fit at specified address *addr*

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The output section does not fit at the address specified by the BIND/ROMBIND option since there is not enough unallocated memory space at this point.

Directive file BIND/ROMBIND option is used for output section *secname* This overrides any other allocation option

Type: Warning

Explanation: Both a directive file BIND/ROMBIND option and another allocation option are specified for the output section. The linker ignores the other allocation option, since BIND/ROMBIND options have the highest priority.

Directive file INTO/ROMINTO option error: cannot direct output section *secname* to a memory area - no memory area with the specified attributes was defined

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The output section is directed by attributes to a memory area but no memory area with the requested attributes was defined in the MEMORY statement.

Directive file INTO/ROMINTO option error: memory area *mem*, specified for output section *secname*, is undefined

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The output section is directed to a named memory area which was not defined in the MEMORY statement.

Directive file INTO/ROMINTO option error: output section *secname* (size *size*) cannot fit in any memory area having the specified attributes

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The output section is directed by attributes to a memory area but there is not enough space left for this output section in any memory area that has the specified attributes.

Directive file INTO/ROMINTO option error: output section *secname* (size *size*) does not fit in memory area *mem*

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The output section is directed to a named memory but there is not enough space left for this output section in the memory area.

Directive file error: input file *filename* specified inside SECTIONS statement not found in input file list

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The file name specified as part of an input section specification is not in the input file list. The input file list includes all input files specified in the invocation line and in the directive file before the SECTIONS statement.

Directive file error: integer constant *int* too big near line *num*

Type: Fatal

Explanation: The directive file contains an invalid integer constant. Integer constants must be in the range 0 - (2**32)-1.

`Directive file error: invalid memory attribute` *att* `specified. Will be ignored`

Type: Warning

Explanation: The directive file contains an invalid memory attribute letter. This is ignored by the linker.

`Directive file error: current location symbol (.) decremented inside output section` *secname* `specification`

Type: Fatal

Explanation: The directive file SECTIONS statement contains an error. The current location symbol (.) is used incorrectly in the output section specification, such that its new value is less than its old value. The current location symbol may never be decremented.

`Directive file error: current location symbol (.) used not inside SECTIONS statement, near line` *linenum*

Type: Fatal

Explanation: The current location symbol (.) is used outside the SECTIONS statement of the directive file. The current location symbol may be used only in an assignment statement that appears as part of an input section list in a SECTIONS statement.

`Directive file error: optional header magic number, specified for OPTION OMAGIC, exceeds 16 bits`

Type: Fatal

Explanation: An invalid integer constant is specified as an argument for the directive file OPTION OMAGIC statement. The argument should not exceed 16 bits (greater than 65535).

`Directive file error: specified MEMORY area` *mem* `exceeds 32 bit address range`

Type: Fatal

Explanation: The directive file MEMORY statement contains an error. The memory area is specified with an address range that exceeds 32 bits (i.e. the starting address plus length exceeds the value 2**32).

`Directive file expression error: output section` *secname*`, used as an argument for SIZEOF, FILEADDR or ADDR function, is not found`

Type: Fatal

Explanation: The directive file expression contains an error. The output section that was specified as an argument to the SIZEOF, FILEADDR or ADDR function is not found.

`Directive file expression error: symbol` *symbolname* `not found`

Type: Fatal

Explanation: The directive file assignment statement contains an error. The symbol that was specified as part of the right-hand-side of an assignment statement is not found.

`Directive file parse error:` *err* `near line number` *linenum*

Type: Fatal

Explanation: The directive file contains a parsing error. This is typically a syntax error. The parsing error type is denoted by *err*.

`Directive file syntax error: ending quote expected near line` *num*

Type: Fatal

Explanation: The directive file contains a syntax error. A string is missing the end quote.

`Directive file syntax error: unrecognized keyword after OPTION near line` *linenum*

Type: Fatal

Explanation: The directive file contains a syntax error. An invalid keyword is specified after the OPTION keyword.

`Dynamic memory allocation failed (`*num* `bytes required)`

Type: Fatal

Explanation: A system request for the dynamic allocation of *num* bytes has failed.

Entry point is not specified and default entry point symbols (start or _main) cannot be found. Entry point will be set to 0.

Type: Warning

Explanation: The linker failed to determine the program entry point either because it was not specified explicitly through the Specify Entry Point invocation option, or the symbols that mark the entry point by default (start or _main) were not defined in any input file. Therefore the linker will set the entry point to the address 0.

External procedure descriptor (xpd) refers to a symbol which is defined in module that does not have a module table entry. Reference to symbol *symbolname* (index *num*) from section *secname* in *filename*

Type: Severe

Explanation: Applicable only for 32000 modularity. During the relocation process, an external procedure descriptor (see the Assembler .xpd directive) referring to a symbol without a module table entry is detected (the Assembler .module directive was not used for this module). Therefore the linker is unable to modify the contents of xpd.

Inconsistent type declarations of common external symbol *symbolname*

Type: Warning

Explanation: The common symbol has differing type specifications in various modules. Since the symbol is common, no module has precedence. The linker uses the first type specification.

Initialization table requested for a non-executable output file. Request ignored.

Type: Warning

Explanation: The Request Initialization Table and Retain Relocation Information invocation options were specified at the same time. Since the output object file is not executable (being only partially linked), an initialization table cannot be created.

Input file *filename* is not in proper COFF format (bad magic number)

Type: Fatal

Explanation: The magic number is incorrect. The magic number resides in the first two bytes of any object files. If these two bytes contain an invalid value, the file is not recognized as a COFF file and is not processed by the linker.

Instruction operand or address constant cannot fit in space after relocation. Reference to symbol *symbolname* (index *num*) from section *secname* , in *filename*

Type: Severe

Explanation: A piece of code or data (hole), based on a symbolic reference, cannot be modified because its value after relocation does not fit its space. This may be the result of using byte/word displacements or absolute addresses.

Insufficient invocation line arguments

Type: Fatal

Explanation: This message is applicable for UNIX systems only. The linker was invoked without any invocation arguments or options.

Integer constant *int*, specified as an argument to an invocation option, is too big

Type: Warning

Explanation: An argument to the Specify Fill Value or Specify Version Stamp invocation option is out of range. The argument must be in the range 0-2**16-1.

Invalid integer constant *int*, specified as an argument to an invocation option

Type: Fatal

Explanation: An invalid integer constant value is used for the invocation options Specify Fill Value or Specify Version Stamp. This is either an invalid integer constant specification or the integer constant is not within the legal range (greater than 65535).

Invocation option *opt* requires an argument

Type: Fatal

Explanation: This message is applicable for UNIX systems only. The invocation option is specified without the required argument. Note that some invocation options may require a space before the argument.

Library file *filename*, specified with -l invocation option, not found

Type: Fatal

Explanation: This message is applicable for UNIX systems only. The library file specified with the -l invocation option is not found in any directory in the library search path.

Library file *filename* has no symbol directory. The GNX archiver utility may be used to restore it

Type: Fatal

Explanation: The input library file does not contain a symbol directory. This file therefore cannot be processed. The symbol directory may not exist because the library file has been stripped. The GNX archiver (n)ar may be used to rebuild the symbol directory.

Link table entry offset not divisable by 4 after relocation. Reference to symbol *symbolname* (index *num*) from section *secname*, in *filename*

Type: Severe

Explanation: This error message is applicable only for 32000 modularity. The link table entry offset, used as an operand of the Series 32000 cxp, must be a multiple of 4.

Module address in external procedure descriptor is out of range (greater than 65535) after relocation. Reference to symbol *symbolname* (index *num*) from section *secname*, in *filename*

Type: Severe

Explanation: This error message is applicable only for 32000 modularity. The module table entry address field of an external procedure descriptor (generated by the Assembler .xpd directive) is not within the legal range and therefore cannot be modified. This error may be the result of a module table entry located outside the first 64 Kbytes of the memory.

Module table entry located outside the first 64K (address *addr*), in output section *secname*

Type: Warning

Explanation: A module table entry was bound to an address which is not within

the first 64K of memory. This may result an error since a module table entry address is limited to 16 bits. Additional linker errors may be issued as a result of this error.

More than one directive file allocation option is used for output section *secname*. Using allocation option priority rules.

Type: Warning

Explanation: The SECTIONS statement of the directive file has more than one allocation option specified. The linker uses the allocation option with the highest priority.

More than one directive file specified

Type: Fatal

Explanation: More than one directive file is specified. Only one directive file can be specified as linker input.

Multiply defined symbol *symname*, defined in *filename1* already defined in *filename2*

Type: Fatal

Explanation: The symbol has more than one definition. A symbol may have only one definition.

No input object files specified

Type: Fatal

Explanation: No input object files have been specified for the linking process.

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

Type: Warning

Explanation: The input object files have different magic numbers. Object files that use the modularity feature have a different magic number. It is not recommended to mix relocatable and modular object files in one link process.

Output section *secname* (size *size*) cannot fit in remaining unallocated configured memory

Type: Fatal

Explanation: Refers to an output section without an allocation option specified in the directive file. The linker is unable to fit the output section into the remaining unallocated configured memory since there is not enough space left for it.

`Program base relative offset in an external procedure descriptor is out of range (greater than 65535). Reference to symbol` *symbolname* `(index` *num*`) from section` *secname*`, in` *filename*

Type: Severe

Explanation: This error message is applicable only the modularity feature. The program base relative offset field of an external procedure descriptor (generated by the Assembler .xpd directive) is out of range and therefore cannot be modified. This error may be caused when calling a procedure which is located more than 64K after its module program base.

`Read error on file` *filename*

Type: Fatal

Explanation: An error was detected when trying to read from the file.

`Seek error on file` *filename*

Type: System

Explanation: An error was detected when trying to perform a seek operation on the file.

`Specified entrypoint symbol` *symbolname* `does not exist. Using default entrypoint`

Type: Warning

Explanation: The symbol specified as the program entry point on the invocation line cannot be found. Therefore a default entry point is determined according to the linker's default rules.

Specified undefined symbol *symbolname* never resolved

Type: Severe

Explanation: The definition of the symbol is not found in any of the input object files or in the directive file.

Unable to recover from previous errors

Type: Fatal

Explanation: The linking process is aborted because of previously reported severe errors.

Undefined symbol *symbolname*, first referenced in file *filename*

Type: Severe

Explanation: The definition of the symbol that is referenced in the input object file is not found any input object file or in the directive file.

Unknown invocation option *opt*

Type: Fatal

Explanation: This message is applicable for UNIX systems only. An unrecognized invocation option is specified.

Write error on file *filename*

Type: System

Explanation: An error was detected when tryping to write on the file.

Appendix C

GLOSSARY

.gnxrc (gnx.ini on VMS) A GNX target specification file that is used by GNX tools to obtain the CPU, FPU, MMU, system bus-width, and OS target specifications.

.bss section A COFF file section. It normally contains uninitialized data.

.data section A COFF file section. The .data section contains initialized data.

.text section A COFF file section. The .text section contains executable code.

Allocation The process by which the linker constructs output sections from input sections and allocates memory for the output sections.

COFF Acronym for the Common Object File Format. This is the standard object file format for the Unix System V operating system, and for the GNX software tools. A COFF file contains machine code and data and additional information for relocation and debugging purposes.

Common data Refers to external symbols that are not defined in any input object file, but are instead consolidated and allocated by the linker. Examples of common data include symbols that are declared with the .comm assembler directive, uninitialized variables declared in C outside any function, and Fortran COMMONs.

Cross configuration When the compilation and execution of the compiled program are done on different machines (the host and target machines are different).

Directive file The directive file controls certain actions of the linker (especially memory configuration and allocation). A directive file to be used as input for the linking process may be specified on the linker invocation line.

Entry point The starting point of program execution. The entry point address is part of the information saved in an executable object file. A symbol to mark the entry point may be specified on the linker invocation line.

Executable object file An executable object file is the final product of a linking process. In an executable object file all external symbolic references have been resolved. The executable object file is therefore in a form that can be executed on the *Series 32000*-based target system.

External symbol A symbol that is recognized by all modules. Such a symbol can be defined in one module but referenced from any module.

Initialization table A table created by the linker to support data initialization. This table may be used by programs requiring initialized data in an embedded environment. The initialized data is copied from ROM to RAM at run-time. The initialization table provides information about memory segments to be copied from ROM to RAM or to be filled with zeros at run-time. This information is used by both the standard C initialization routine (crt0) or by a user-initialization routine.

Input section A COFF section of a linker input object file. The linker combines input sections to create output sections. By default input sections of the same name are combined to create one output section having this name in the output file.

Library file A collection of object files that typically contains useful routines. The linker selects from a specified library file those object files which resolve external references.

Memory map A description of the memory layout after the linking process. A memory map is an optional output of the linker.

Object file A file that is the output of either the assembler or the linker. An object file contains compiled code and data and additional information for relocation and debugging purposes.

Option The UNIX term for a parameter, specified on the command line, that is used to control the utility.

Output section A COFF section of a linker output object file. The linker combines input sections to create output sections. By default input sections of the same name are combined to create one output section having this name in the output file.

Partially linked object file An object file created by the linker, which is unexecutable since it contains unresolved external references. A partially linked object file may be used as input for a subsequent linking process.

Qualifier The VMS term for a parameter, specified on the command line, that is used to control the utility.

Relocation information A part of the object file that is used by the linker in the relocation process. Relocation information contains information on symbolic references that require modification of pieces of code or data (holes) at link time. The linker uses this information to calculate the final value of the holes.

Relocation process The process by which the linker modifies pieces of code or data (holes) that cannot be calculated before link-time. These holes are typically addresses or displacements that are created as a result of symbolic references. After the allocation process is completed the linker assigns final values to these holes, using relocation information (part of the COFF file) from the input object files.

Resolution of symbolic references The process by which the linker matches

external symbolic references with their definition.

Section A contiguous block of code or data having common attributes. In the COFF file code and data are separated to sections. Typically there are three types of sections: the .text section, containing machine code; the .data section, containing initialized data; and the .bss section, containing uninitialized data.

Symbol A symbol is used either to mark a program location or to represent a data element. Each symbol is associated with a memory address after the linking process.

Symbol directory Part of a library file. The symbol directory contains information on the external symbols which are defined in library members. The linker uses the symbol directory to select the correct library member for the linking process.

Symbol table Part of the object file. The symbol table contains information about symbols defined or referenced in the source program(s), and is used for various purposes such as resolution of external references (by the linker) and symbolic debugging.

Symbolic reference The use of a symbol in a statement other than its definition. An external symbolic reference is a reference to a symbol which is defined outside the module in which it resides.

INDEX

A

Aligning, output section 3-11
Allocation of output sectionOPs 3-1
Allocation of output sections 1-5, 3-1, 3-8, 5-1
  assigning an address 5-2
  creating output sections 5-1
  linker created input sections 5-7
  memory map 5-8
  options 3-8
Allocation options 3-8
  BIND 3-8
  INTO 3-8
  ROMBIND 3-8
  ROMINTO 3-8
Assigning a memory address 5-2
Assignment operators A-2
Assignment statement 3-2, 3-14
  operators A-2
  within SECTIONS statement 3-15
Attribute letter 3-4

B

Binary operators A-2
BIND, allocation option 3-8, 5-2
.bss section 1-1, 2-9, 3-8, 4-4, 5-7

C

COFF file 1-1, 2-6, 3-12, 3-16, 4-2, 6-1
  .bss section 1-1
  .data section 1-1
  entry point 2-6
  header magic number 3-16
  relocation information 6-1
  section header 3-12
  symbol table 4-2
  .text section 1-1
Comment 3-1, 3-3
Common data 2-9, 3-7, 4-4
  .bss section 1-1, 2-9, 3-8, 4-4, 5-7
Configuration
  cross C-1
Creating gaps 3-15
Creating output sections 5-1
Current location symbol 3-15

D

.data section 1-1
Directive file 1-3, 1-5, 2-4, 3-1, 4-5, 5-1
  assignment statement 3-14
  comment 3-3
  SECTIONS statement 3-5
  MEMORY statement 3-4
  output file options 3-16
  specification 3-3
  structure 3-1

E

Entry point 2-6
Error messages B-1
Executable object file 1-1, 1-3
  default filename and permission 3-16
  linker output 1-3
  memory map 2-5
  relocation information 2-7
  strip symbolic information 2-7
Expressions, directive file 3-2, 3-14, A-1
  assignment operators A-2
  binary operators A-2
  integer syntax A-1
  special functions A-3
  unary operators A-2

F

Function, linker 1-3, 1-4
  allocation of output sections 1-5, 3-1, 5-1
  relocation of memory address 1-5, 6-1
  resolution of symbolic references 1-4, 4-1

G

Gaps, creating 3-15
GNX Target Setup (gts) 2-4
Grouping output sections 3-13

H

Header magic number 3-16

I

Initialization table 2-8, 3-8, 5-7
Input file specification 2-4, 2-5, 2-6, 3-1, 3-3
Input, linker 1-2
  directive file 1-3
  library file 1-3
  partially linked object file 1-2
  simple object file 1-2
Input section 1-2, 3-1, 3-5, 5-1, 5-8, 6-2
  specification 3-6
Integer syntax A-1
  decimal A-1
  hexadecimal A-1
  octal A-1
INTO, allocation option 3-9, 5-2
Invocation line 2-1
  options 2-3
  UNIX environment 2-1
  VMS environment 2-2
Invocation options 2-3
  issue 2-9
  keep relocation information 2-7
  request initialization table 2-8
  request memory map 2-5
  retain relocation information 2-6
  specify directive file 2-4
  specify entry point 2-6
  specify fill value 2-8
  specify library directory 2-5
  specify library filename 2-4
  specify output filename 2-3
  specify undefined symbol 2-8
  specify version stamp 2-10
  strip symbolic information 2-7
  suppress error message 2-9
  suppress warning message 2-9
  version information 2-10
Issue warning for defined common data 2-9

L

Library file 1-2, 1-3, 2-1, 4-3
  directory search 2-5
  member 2-8, 4-3
  specification 2-4, 3-3
  symbol directory 4-3
  UNIX specification 2-2
  VMS specification 2-3
Line number information 2-7
Linker
  error messages B-1
  functions 1-4
  input 1-2
  introduction 1-1
  output 1-3
linker.def file 2-4

M

Memory
  allocation 1-1, 1-3, 1-5, 3-7, 4-4, 5-2
  configuration 1-5, 3-1, 3-4, 5-2
Memory address 1-1, 1-4, 3-1, 3-4
  assign 3-9, 3-10, 3-11, 3-14, 5-1, 5-2
  relocation 1-5, 6-1
Memory map 1-3, 1-4, 5-8
  specification 2-5
MEMORY statement 3-1, 3-4, 3-14, 5-2
  attribute letter 3-4
Modularity support 5-2
Module table 5-7
  entry 3-7, 6-1

O

Options, allocation 3-8
  BIND 3-8
  INTO 3-8
  ROMBIND 3-8
  ROMINTO 3-8
Options, invocation 2-3
  issue warning for defined common data 2-9
  keep relocation information 2-7
  request initialization table 2-8
  request memory map 2-5
  retain relocation information 2-6
  specify directive file 2-4
  specify entry point 2-6
  specify fill value 2-8
  specify library directory 2-5
  specify library filename 2-4
  specify output filename 2-3
  specify undefined symbol 2-8
  specify version stamp 2-10
  strip symbolic information 2-7
  suppress error message 2-9
  suppress warning message 2-9
  version information 2-10
Options, output file 3-2, 3-16
  default filename and permission 3-16
  header magic number 3-16
Output file
  options, specifying 3-16
Output, linker 1-3
  executable object file 1-3
  memory map 1-4
  options 3-2, 3-16
  partially linked object file 1-3
  specification 2-3, 2-5
Output section 1-3
  aligning 3-11
  allocation 1-5, 3-1, 3-4, 3-5, 3-8, 5-1
  assigning an address 5-2
  creating 3-1, 3-5, 5-1
  creating gaps 3-15

grouping 3-13
specification 3-5

P

Partially linked object file 1-2
linker input 1-2
linker output 1-3
specification 2-6, 3-3

R

Relocation information 2-6, 6-1
keep relocation information 2-7
Relocation of memory address 1-5, 6-1
Relocation table 6-1
Request initialization table 2-8, 2-11, 2-12
Request memory map 2-5
Request output memory map 2-11, 2-12
Resolution of symbolic references 1-4, 4-1
Retain invocation line option 2-6
Retain relocation information 2-6, 2-11, 2-12

S

Section 1-1
Section gaps 2-8
Section header 3-12
SECTIONS statement 3-1, 3-5, 3-14, 4-5, 5-1, 5-2
aligning a section 3-11
allocating a section to memory 3-8
grouping output sections 3-13
input section specification 3-6
output section specification 3-5
setting the section type 3-12
symbol assignment 3-15
Setting the section type 3-12
Simple object file 1-2
linker input 1-2
specification 3-3
Special functions A-3
file address function A-4
highest memory address function A-4
memory address function A-4
next address function A-4
size of output function A-3
Specify directive file 2-4, 2-11, 2-12
Specify fill value 2-8, 2-11, 2-12
Specify library directory 2-5, 2-11
Specify library filename 2-4, 2-11
Specify output filename 2-3, 2-11, 2-12
Specify program entry point 2-6, 2-11, 2-12
Specify undefined symbol 2-8, 2-11, 2-12
Specify version stamp 2-10, 2-11, 2-12
Strip symbolic information 2-7, 2-11, 2-12
Suppress error message 2-9, 2-11, 2-12
Suppress size warning 2-9, 2-11, 2-12
Symbol 1-4, 3-14, 4-1
assignment 3-15
directive file 4-5
external 1-4, 2-8, 4-1, 4-5
library file 4-3
linker defined 4-6
Symbol directory 4-3
Symbol table 1-4, 2-7, 4-2
Symbolic reference 6-1
common data 4-4
resolution 1-1, 1-4, 4-1

T

.text section 1-1, 2-6

U

Unary operators A-2

V

Version information 2-10, 2-11, 2-12
Version invocation line option 2-10

Series 32000®

GNX — Version 4.4
Support Libraries
Reference Manual

Customer Order Number 424010508-004

June 1992

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
4.0	May 1990	First Release. Addition of the **rename** and **getenv** system calls.
4.1	September 1990	Bug fixing.
4.2	February 1991	Synchronization revision. No changes.
4.3	August 1991	Synchronization revision. No changes.
4.4	June 1992	Synchronization revision. No changes.

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.

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

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
4.0	May 1990	First Release. Addition of the **rename** and **getenv** system calls.
4.1	September 1990	Bug fixing.
4.2	February 1991	Synchronization revision. No changes.
4.3	August 1991	Synchronization revision. No changes.

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.

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

CONTENTS

Chapter 1 OVERVIEW

1.1 INTRODUCTION 1-1
1.2 OPERATING SYSTEM CALL SIMULATION 1-2
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 Getenv 2-12
2.3.6 Lseek 2-13
2.3.7 Open 2-15
2.3.8 Read 2-17
2.3.9 Rename 2-18
2.3.10 Sbrk 2-19
2.3.11 Unlink 2-20
2.3.12 Write 2-22

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

Chapter 5 FPEE LIBRARY

5.1 INTRODUCTION 5-1
5.2 FPEE LIBRARY CONFIGURATIONS 5-2
5.3 INTEGRATING FPEE WITH AN APPLICATION 5-2
5.3.1 Integrating FPEE with Series 32000/UNIX Applications 5-2
5.3.2 Cross Application FPEE Integration 5-3
5.3.3 FPEE Library and the Math Library Integration 5-3
5.3.4 FPEE Error Handling Routines 5-4
5.4 FPEE OPERATIONAL DETAILS 5-4
5.4.1 Operational Overview 5-5
5.4.2 FPEE Enhancements to the FPU 5-6
5.4.3 NS32081 FPU, NS32381 FPU and FPEE 5-7
5.4.4 FPEE Program Control 5-8
5.4.5 FPEE Comparisons 5-10
5.4.6 FPEE Exception Handling 5-11
5.4.7 FPEE Rounding Modes 5-13

Chapter 6 `libHfp` - HIGH-SPEED FP EMULATION LIBRARY

6.1 INTRODUCTION 6-1
6.2 THE `libHfp` LIBRARY VS THE FPEE LIBRARY 6-1
6.3 HOW TO USE THE `libHfp` LIBRARY 6-2
6.4 `libHfp` TECHNICAL SPECIFICATIONS 6-3
6.4.1 Compatibility and Conformity to IEEE/754 Standards . 6-4
6.4.2 Use of the Mathematical Library 6-5
6.4.3 The `libHfp` Interface 6-5
6.4.4 Exception Handling 6-8
6.5 EXAMPLES 6-11

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 5-1.	Instruction Codes	5-5
Table 5-2.	FPEE Library-Implemented IEEE 754 Operations	5-7
Table 5-3.	Default Return Values for Overflow Exceptions	5-14
Table 6-1.	Instructions Emulated By Calls to `libHfp` Routines	6-7
Table 6-2.	Instructions Emulated Inline	6-8
Table 6-3.	Mapping of Floating-Point Registers	6-8
Table 6-4.	Exception Handling Routines	6-9

INDEX

Chapter 1

OVERVIEW

1.1 INTRODUCTION

The GNX—Version 4 Support Libraries provide run-time support for the C, Pascal, 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`, `mon532`, `moncg16`, `moncg160`, `mongx32`, `mongx320`, or `mongx32e` 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 High-speed Floating-Point Emulation Library (**`libHfp`**) is a library of very fast floating-point routines that provide an efficient low-cost floating-point solution on FPU-less systems, by emulating the NS32081 and NS32381 floating-point instructions in software.

The location and the names of these libraries may vary with the host operating system and are discussed in the *Series 32000 GNX — Version 4 Commands and Operations Manual* provided with the GNX tools.

These libraries are similar to the standard C, Pascal, 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 DBUG (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, 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`, `rename`, `sbrk`, and `unlink`. These system calls are dependent on the development board monitors, DBUG 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.

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.

Chapter 6 describes the High-speed floating-point emulation library.

Appendix A describes the *Series 32000* standard calling conventions.

See the *Series 32000 GNX — Version 4 Pascal Optimizing Compiler Reference Manual*, or the *Series 32000 GNX — Version 4 FORTRAN 77 Optimizing Compiler Reference Manual* for a description of the functions for Pascal, 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,

```
(dbug) <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 ERROR in line number 208 incorrect number of fields line is: .if t 10^9 ERROR in line number 209 incorrect number of fields line is: .if n 1.0E9 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
ldexp	3.14	localtime	3.6	longjmp	3.30
malloc	3.20	printf	3.21	putchar	3.22
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
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` and `DBUG` 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 Getenv

NAME

getenv – get the value of an environmental name

SYNOPSIS

```
char *getenv(name)
char *name;
```

DESCRIPTION

The getenv function searches an environment list, provided by the host environment, for a string that matches the string pointed to by *name*.

The getenv function returns a pointer to a string associated with the matched list member. The array pointed to shall not be modified by the program, but may be overwritten by a subsequent call to the genenv function. If the specified *name* cannot be found, a null pointer is returned.

2.3.6 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

2.3.7 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*

2.3.8 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

2.3.9 Rename

NAME

rename – changes the name of a file

SYNOPSIS

```
int rename(old, new)
char *old;
char *new;
```

DESCRIPTION

Rename causes the file whose name is the string pointed to by *old* to be henceforth known by the name given by the string pointed to by *new*. The file named *old* is effectively removed. If a file named by the string pointed to by *new* exists prior to the call to the **rename** function, the behavior is undefined.

RETURN VALUE

The rename function returns zero if the operation succeeds, nonzero if it fails. The global variable errno indicates the reason for the failure.

2.3.10 Sbrk

NAME

sbrk – allocate memory in heap

SYNOPSIS

```
char *sbrk(incr)
int incr;
```

DESCRIPTION

Sbrk allocates *incr* bytes of memory from the unallocated memory heap and returns the address of its lowest byte. The heap is defined as a continuous area which resides between the addresses of two dummy symbols: _HEAP$_START and _HEAP$_MAX. These two symbols may be redefined in a linker-directive file by the user. By default, _HEAP$_START is set to the first memory location above the program data area (_end) and _HEAP$_MAX is set to the value 0x00ffffff. Allocated memory can not exceed the address of _HEAP$_MAX. Also, when the stack pointer value is between _HEAP$_START and _HEAP$_MAX, the upper limit of the heap must be at least 1024 bytes below the current value of the stack pointer. Sbrk checks if memory exists, writing and verifying the contents of the highest byte allocated. There is no way to de-allocate this memory.

RETURN VALUE

Sbrk returns a pointer 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 is allocated if the following is true:

[ENOMEM] Insufficient memory exists to support the expansion (either _HEAP$_MAX will be exceeded or the heap will encroach on the stack).

SEE ALSO

malloc, nmeld (the linker-directive language and file).

2.3.11 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

2.3.12 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 4 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 ERROR in line number 1386 incorrect number of fields line is: .if n pi ERROR in line number 1387 incorrect number of fields line is: .if t 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 *strncpy(s1, s2, n)
char *s1, *s2;                char *s1, *s2;

char *strncat(s1, s2, n)      strlen(s)
char *s1, *s2;                char *s;

strcmp(s1, s2)                char *strchr(s, c)
char *s1, *s2;                char *s, c;

strncmp(s1, s2, n)            char *strrchr(s, c)
char *s1, *s2;                char *s, c;

char *strcpy(s1, s2)          char *index(s, c)
char *s1, *s2;                char *s, c;

char *rindex(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, NS32181 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 and NS32181 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, or FORTRAN.

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).

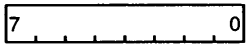
4.2.1 Number Formats

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

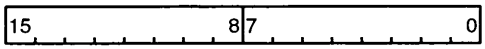
The most significant bit in the integer format is a sign bit used to implement negative integers in two's-complement representation. The math library operates on integers in three formats.

Byte Format:



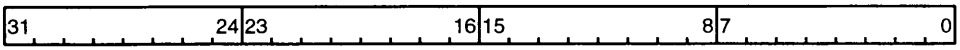
Byte format represents values from negative 128 through positive 127.

Word Format:



Word format represents values from negative 32768 through positive 32767.

Double-word Format:



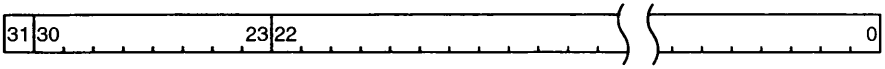
Double-word format represents values from negative 2147483648 through positive 2147483647.

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:

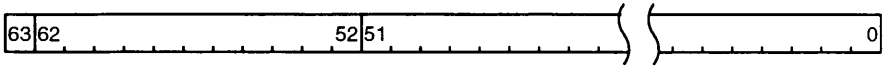
- Sign
- Exponent
- Fraction

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.

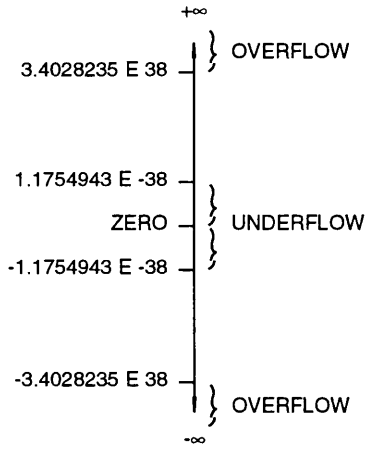
Though zero is a valid floating-point number, it is not a normalized number. A zero is represented by all 0's in the exponent and fraction. The sign bit can be either 0 (positive zero) or 1 (negative zero). Normally, positive and negative zero are equivalent, but in special cases, such as divide by zero, 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

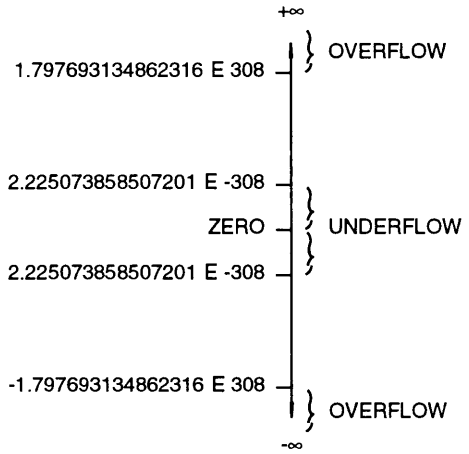
NOTE: 1. These values are positive. The sign bit does not affect the absolute magnitude of the limits. The negative limits can be determined by adding a sign bit to the hexadecimal value and affixing a negative sign in front of the floating-point value.

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, NS32181 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 ERROR in line 888 String notation in .if didn't parse line:.if 'Modula'true' , that do not have a predefined list of math function names, programs call the math library functions directly.

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.

ERROR in line 41 String notation in .if didn't parse line:.if '081lib'supported'The FPEE library is provided in source form and as a binary The FPEE library is provided as a binary library suitable for its particular GNX tool-set environment. The FPEE library is configured to enhance/emulate the NS32381 FPU and the NS32081 FPU (see Section 5.4.3).

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

ERROR in line 70 String notation in .if didn't parse line:.if '081lib'supported'All FPEE binary libraries provided with the GNX development All FPEE binary libraries provided with the GNX development tool-set are configured for the NS32381 (see Section 5.4.3).

ERROR in line 76 String notation in .if didn't parse line:.if '081lib'supported'

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 `fprinit`, 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 `fprint` is in the `fpinit.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).

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

5.4.1 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.

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.

ERROR in line 515 String notation in .if didn't parse line:.if '081lib'supported'The FPEE library does distinguish between NS32081 FPU instruction emulation

The FPEE library enhances/emulates the NS32381.

The NS32381 FPU FPEE library implements eight 64-bit registers and supports the RMB bit of the FSR. 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:

X X X X 1 X 1 X X X X X X X X X	Invalid Operation trap
X X X X 1 0 0 X X X X X X X X X	Return NAN **
X X X X 1 1 0 X X X X X X X X X	Invalid-Operation trap

** If the invalid operand is a denormalized number, the FPEE software returns a normalized value.

Division by zero exception:

X X 1 X X X 1 X X X X X X X X X	Division by zero trap
X X 1 0 X X 0 X X X X X X X X X	Return infinity
X X 1 1 X X 0 X X X X X X X X X	Division by zero trap

Overflow signaled:

1 X X X X X 1 X X X X X X X X X	Overflow trap
1 0 X X X X 0 X X X X X X X X X	Result according rounding mode
1 1 X X X X 0 X X X X X X X X X	Overflow trap

See Section 5.4.7 on rounding mode for results.

Overflow on conversion from float to integer:

X X X X X X 1 X X X X X X X X X	Overflow trap (non-IEEE 754 Standard)
X X X X X 0 0 X X X X X X X X X	Return -1
X X X X X 1 0 X X X X X X X X X	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)

Chapter 6

libHfp - HIGH-SPEED FP EMULATION LIBRARY

6.1 INTRODUCTION

The High-Speed FP Emulation Library (**libHfp**) is used to create floating-point programs for those *Series 32000* systems that lack floating-point unit hardware (FPU). The **libHfp** is a library of very fast floating-point routines. It provides an efficient low-cost floating-point solution for systems without an FPU, by emulating the NS32081/NS32181/NS32381 floating-point instructions in software.

The high-speed **libHfp** emulation library is approximately *ten* times faster than the the FPEE library, which is also used for floating-point emulation. This is because unlike the FPEE library routines, which are invoked through a hardware trap mechanism, the **libHfp** routines are invoked by procedure calls embedded in your software programs by the assembler at compile-time. Thus by using the **libHfp** you do not incur the runtime cost associated with the FPEE library.

This chapter describes the **libHfp** library. Sections 6.2 and 6.3 explain when and how to use the **libHfp** library. Section 6.4 describes technical details, compatibility issues and exception handling. Section 6.5 presents several examples of usage.

6.2 THE libHfp LIBRARY VS THE FPEE LIBRARY

The high-speed **libHfp** emulation library provides the same precision and functionality as the NS32081/NS32181/NS32381 floating-point hardware (except for the conditions mentioned below), providing the best solution for floating-point code generation for those *Series 32000*-based systems without an FPU.

However, the **libHfp** does not support the full flexibility advocated by the ANSI "IEEE Standard for Binary Floating-Point Arithmetic" (ANSI/IEEE Std 754-1985), such as a selection of four rounding modes. Issues of compatibility and conformity to IEEE/754 standards are discussed in Section 6.4.1. If IEEE/754 functionality is required, either the FPEE library can be used or the **libHfp** can be used in conjunction with the FPEE library.

The FPEE library, described in Chapter 5, is an enhanced binary compatible emulation of the NS32081 and NS32381 floating-point units. As a high-level language programmer (or a writer of assembly code), you do not need to know if an FPU is installed on your target-system. All you need to do is link the FPEE library with your object code. If the resulting execuTable program is then run on a system with an FPU, the

floating-point instructions are executed by the FPU. If on the other hand the program runs on a system that has *no* FPU, the *Series 32000* trap mechanism catches the floating-point instructions (ILL, *illegal trap*) and the floating-point operation is then performed by software. This trap mechanism has a significant runtime performance cost. The enhancements of the FPEE library to be fully conformant to ANSI/IEEE 754 even increase this cost. FPEE emulation time is 70-120 times slower than FPU execution time.

On the other hand, the High-speed **libHfp** library was designed to avoid any unnecessary performance penalties. The emulation routines were rewritten for optimal performance. **libHfp** emulation time is only 6-9 times that of the FPU, or about 10-12 times faster than FPEE emulation.

Using **libHfp** has its price: the program object code size increases. Each floating-point instruction in your program roughly triples in size, not counting the one-time cost of the **libHfp** library itself (3 - 6K bytes). Overall, code-size increases by 10 - 50%. Therefore, if code-size is more important than execution speed, the FPEE library will perhaps yield a smaller execuTable file.

For most uses, **libHfp** is an efficient and adequate solution.

There is currently no *modular* version of **libHfp**.

6.3 HOW TO USE THE libHfp LIBRARY

In order to use **libHfp**, you need to re-compile (or re-assemble) your source program with the `emulation` option. This option instructs the GNX assembler to *replace* floating-point instructions in your program by integer instructions and by calls to **libHfp** emulation routines. This makes the transition from an FPU chip or from FPEE to the use of **libHfp** relatively painless, even for assembly programs; a program source file requires no changes and needs only to be re-assembled. The resulting execuTable program will run the same way, whether or not an FPU is installed.

You have the following three options to create your execuTable program:

1. Use only the **libHfp** library, as follows. This is the simplest option, and will result in the fastest solution.

    a. Compile (or assemble) your source program with the `emulation` option, in one of two ways. You can use the GTS utility[1] to set the `FPU` entry in the GNX target configuration file to `emulation`. Or you can compile (or assemble) with the appropriate command-line switch:

1. See Chapter 4 of the "GNX Commands and Operations Manual"

under UNIX: `-KFemulation`
under VMS: `/TARGET=(FPU=EMULATION)`.

b. Link your code with **libHfp** and other floating-point emulating language libraries. The **libHfp** library is called `libHfp.a`; the other libraries have an additional "H" in their name, i.e. `libc.a` becomes `libHc.a`, `libm.a` becomes `libHm.a`, etc.

   If you use a GNX cross-compiler to compile and link your program under UNIX, then the compiler will automatically determine for you which libraries are appropriate. You can inspect this with the `-v` or `-vn` switches.

For a detailed example, please see example 1 in Section 6.5.

2. If you need the full functionality of the FPEE library, or if you have object programs that already contain floating-point instructions, then you can use the FPEE mechanism; compile (or assemble) your source program *as if* an FPU were present, link your program with the FPEE library, and rely on hardware traps to emulate the floating-point instructions (if you use a GNX compiler on UNIX to compile, assemble and link, all this is accomplished with the `-f` switch). You will find more details in Chapter 5.

3. Use a combination of both libraries. Compile performance-critical code under `emulation`, and link with both the **libHfp** and the FPEE libraries. The performance-critical code will thus use the efficient **libHfp** emulation, and the rest will trap into the FPEE routines as described before.

   Example 2 in Section 6.5 describes a possible scenario.

6.4 libHfp TECHNICAL SPECIFICATIONS

The **libHfp** library consists of fast emulation routines for the *Series 32000* floating-point instructions, global variables to which floating-point registers are mapped, and replaceable exception-handling routines.

When a program is compiled (assembled) with the `emulation` option, the GNX assembler replaces each floating-point instruction with a sequence of integer instructions. Simple instructions (such as `movf` and `movl`) are emulated inline, but most instructions are replaced by procedure-calls to the **libHfp** routines that emulate the instruction. Floating-point registers are mapped onto global variables in memory. You can examine these transformations, summarized in Tables 6-1 to 6-3, by using the assembler's `List (-L)` switch.

The emulation routines are re-entrant; floating-point code may therefore be used in signal and interrupt handlers.

6.4.1 Compatibility and Conformity to IEEE/754 Standards

The **libHfp** library is arithmetically compatible to the *round-to-nearest* rounding mode of the NS32081/NS32181/NS32381 FPU. It implements single and double precision floating-point numbers, using the NS32081/NS32181/NS32381 formats, obtaining the same results as the FPU.

In particular, the following NS32081/NS32181/NS32381 and ANSI/IEEE 754 features are emulated:

- basic single precision format (float)
- basic double precision format (long)
- signed zero
- round to nearest
- division-by-zero exception
- invalid operation exception
- overflow exception
- underflow exception
- exception handling for division by zero, reserved operands, overflow and underflow (see further Section 6.4.4).

Following are the IEEE/754 features that are not supported by **libHfp**. If correct execution of your program depends on any of these, **libHfp** should be used in combination with FPEE, as discussed in Section 6.3.

- special (NaN, denormalized, infinity) arithmetic (this is also not supported by the NS32081/NS32181/NS32381)
- round towards + ∞, round towards - ∞, round towards zero
- unordered compare
- inexact exception
- reserved operand exception by the `cmpf` and `cmpl` instructions

In addition, the **libHfp** emulation routines have several other attributes which may impact the way your code works:

1. Floating-point emulation is reentrant and interrupTable. This means that interrupt handlers may use floating-point instructions too. However, the access to double precision floating-point variables is not always an atomic operation. This means that some code (such as an interrupt handler) may cease to work, if it relies on the fact that the high-order four bytes and the low-order four bytes of a global double precision variable are *atomically* consistent (Floating-point data cannot have the *volatile* property).
2. Most floating-point instructions are emulated by **libHfp** routine calls. This means that the stack *above* the stack-pointer will be corrupted. This is normally not a problem, of course.

3. Floating-point emulated code size is significantly larger than the original. Beware of displacement overflow in your assembly code, such as in `caseb` instructions.
4. The following NS32381/NS32181 instructions `dotf`, `dotl`, `polyf` and `polyl` are accurate to within 1 *ulps* (units in least place of floating point number) with regards to the FPU hardware.
5. The emulation code for `fsqrt` (single precision square root) and `sqrt` (double precision square root) uses a different algorithm from the respective floating-point square root routines in the mathematical library (`libm`). They may occasionally compute results that differ in the least-significant bit.
6. The FSR register is not updated by **`libHfp`**. In fact, the `lfsr` and `sfsr` instructions do not affect the state of computation.
7. The global library names, ending with two underscore characters (such as `addf__`) are reserved. A complete list of reserved names is found in Tables 6-1, 6-3 and 6-4.

6.4.2 Use of the Mathematical Library

Chapter 4 of this manual describes the GNX mathematical library (`libm`). When using emulation mode and the **`libHfp`** library, all of the following routines may be used:

```
acos, acosh, asin, asinh, atan, atan2, atanh,
cabs, cbrt, ceil, compound, copysign, cos, cosh, drem,
erf, exp, exp2, expm1, fabs, facos, facosh, fasin, fasinh,
fatan, fatan2, fatanh, fcabs, fcbrt, fceil, fcompound,
fcopysign, fcos, fcosh, fdrem, ferf, fexp, fexp2, fexpm1,
ffabs, ffinite, ffloor, ffmod, fhypot, finf, finite,
flog, flog10, flog1p, flog2, floor, fmod, fmodf, fneg,
fpi, fpow, frelation, frem, frint, fsin, fsinh, fsqrt,
ftan, ftanh, gamma, hypot, inf, initrand, j0, j1, jn, log,
log10, log1p, log2, neg, nextdouble, nextfloat, pi, pow,
randomx, relation, rem, rint, sin, sinh, sqrt, tan, tanh
```

In order to use *other* `libm` routines, your code must be linked with the FPEE library.

6.4.3 The `libHfp` Interface

The GNX assembler calls the **`libHfp`** routines in the following manner:

1. The operands of the floating-point instruction are pushed on top of the user-stack, first the right operand (often the *destination*), then the left (*source*) operand. If the operand has access-class *write* or *rmw*, then its address is pushed, otherwise (access-class *read*), its value is pushed. Values smaller than

4 bytes are sign-extended and aligned to a full double-word. Four-byte values are pushed using one `movd` instruction, eight-byte values by two `movds`.

2. The appropriate emulation routine is called. Before returning it writes the result into the expected location and clears the parameters off the stack.

Example: `roundfw 16(fp), _i`

is transformed into:

```
addr   _i, tos          # push the address of the destination
movd   16(fp), tos      # push the value of the source
bsr    roundfw__        # call the emulation routine
```

Example: `addl tos, _j`

is transformed into:

```
addr   _j, tos          # push the address of the destination
movd   8(sp), tos       # push low bytes of source
movd   8(sp), tos       # push high bytes of source
bsr    addl__           # call the emulation routine
adjspb $-8              # pop the stack as required
```

Note the complex interaction between `tos` and `sp`-relative addresses.

The floating-point instructions mentioned in Table 6-2 are emulated inline.

Example: `movl  $0.0, xyz`

is transformed into

```
movqd   $0, xyz+4       # copy the high-order part
movqd   $0, xyz         # copy the low-order part
```

Table 6-3 lists the names of the variables that serve as registers under emulation. These names are reserved by the **libHfp** package. Note that these are the *same* names as used by FPEE.

Example: `movf  f1, _a[r2:d]`

is transformed into

```
movd    F1__, _a[r2:d]XUX
```

Table 6-1. Instructions Emulated By Calls to `libHfp` Routines

EMULATED INSTRUCTION	EMULATION ROUTINE	EMULATED INSTRUCTION	EMULATION ROUTINE
addf	addf__	mulf	mulf__
addl	addl__	mull	mull__
cmpf	cmpf__	polyf	polyf__
cmpl	cmpl__	polyl	polyl__
divf	divf__	roundfw	roundfw__
divl	divl__	roundfw	roundfw__
dotf	dotf__	roundfd	roundfd__
dotl	dotl__	roundlb	roundlb__
floorfb	floorfb__	roundlw	roundlw__
floorfw	floorfw__	roundld	roundld__
floorfd	floorfd__	scalbf	scalbf__
floorlb	floorlb__	scalbl	scalbl__
floorlw	floorlw__	subf	subf__
floorld	floorld__	subl	subl__
logbf	logbf__	truncfb	truncfb__
logbl	logbl__	truncfw	truncfw__
movbf	movbf__	truncfd	truncfd__
movwf	movwf__	trunclb	trunclb__
movdf	movdf__	trunclw	trunclw__
movbl	movbl__	truncld	truncld__
movwl	movwl__		
movdl	movdl__		
movfl	movfl__	-	_fsqrt
movlf	movlf__	-	_sqrt

Table 6-2. Instructions Emulated Inline

INSTRUCTION	INSTRUCTION
absf	negf
absl	negl
movf	lfsr
movl	sfsr

Table 6-3. Mapping of Floating-Point Registers

REGISTER	VARIABLE
fsr	FSR__
f0	F0__
f1	F1__
:	:
f7	F7__
l0	F0__
l1	L1__
:	:
l7	L7__

6.4.4 Exception Handling

libHfp recognizes the following exceptions:

- An operation is attempted with an *invalid* operand (such as divide by zero).
- A computation results in *overflow*.
- A computation results in *underflow*.

When an exception is detected by a **libHfp** routine, it calls an exception routine whose name corresponds to the exception (e.g. `overflow_in_addf__`). Default exception handling routines that handle these exception in a way similar to the emulated hardware, are provided as part of the **libHfp** library (see Table 6-4). A hardware trap is generated artificially by executing the `bpt` instruction. You can suppress these default exception handlers with your own code, by linking routines of the same name with your program. See example 3 in Section 6.5.

The following are two of the default exception handlers:

```
overflow_in_mulf__::
    # overflow is not supposed to happen
    bpt                                 # abort execution
    ret       $(0)

underflow_in_mull__::
    .set  SIGN_BIT, 0x80000000

    save    [r0]                        # SAVE scratch register r0
    sprd    upsr,tos                    # SAVE psr

    .set  SP_OFFSET, 12                 # stack-offset of first param: 12 =
                                        # 4 psr + 4 r0 + 4 return address
    # dest := src.sign xor dest.sign (+0 or -0)
    movd    SP_OFFSET+4(sp), r0         # take src.hi
    xord    4(SP_OFFSET+8(sp)), r0      # xor with dest.hi
    andd    $SIGN_BIT, r0               # leave only sign (set rest to zero)
    movd    r0, 4(SP_OFFSET+8(sp))      # copy result to dest.hi
    movqd   $0, 0(SP_OFFSET+8(sp))      # set dest.lo to zero

    lprd    upsr, tos                   # restore psr
    restore [r0]
    ret     $(0)
```

Table 6-4. Exception Handling Routines
Sheet 1 of 2

FUNCTION NAME†	CALLED FROM	DEFAULT ACTION/RESULT
overflow_in_addf__	addf__, subf__	abort
underflow_in_addf__	addf__, subf__	± 0
reserved_to_addf__	addf__, subf__	abort
overflow_in_addl__	addl__, subl__	abort
underflow_in_addl__	addl__, subl__	± 0
reserved_to_addl__	addl__, subl__	abort
overflow_in_divf__	divf__	abort
underflow_in_divf__	divf__	± 0
reserved_to_divf__	divf__	abort
zero_divisor_to_divf__	divf__	abort
overflow_in_divl__	divl__	abort
underflow_in_divl__	divl__	± 0
reserved_to_divl__	divl__	abort
zero_divisor_to_divl__	divl__	abort
overflow_in_floorfI__	floorfI__	abort
reserved_to_floorfI__	floorfI__	abort
overflow_in_floorlI__	floorlI__	abort
reserved_to_floorlI__	floorlI__	abort

† In the above Table "I" stands for "b", "w" or "d" respectively.

Table 6-4. Exception Handling Routines
Sheet 2 of 2

FUNCTION NAME†	CALLED FROM	DEFAULT ACTION/RESULT
overflow_in_logbf__	logbf__	abort
underflow_in_logbf__	logbf__	± 0
reserved_to_logbf__	logbf__	abort
overflow_in_logbl__	logbl__	abort
underflow_in_logbl__	logbl__	± 0
reserved_to_logbl__	logbl__	abort
reserved_to_movfl__	movfl__	abort
overflow_in_movlf__	movlf__	abort
underflow_in_movlf__	movlf__	± 0
reserved_to_movlf__	movlf__	abort
overflow_in_mulf__	mulf__	abort
underflow_in_mulf__	mulf__	± 0
reserved_to_mulf__	mulf__	abort
overflow_in_mull__	mull__	abort
underflow_in_mull__	mull__	± 0
reserved_to_mull__	mull__	abort
overflow_in_roundfl__	roundfl__	abort
reserved_to_roundfl__	roundfl__	abort
overflow_in_roundll__	roundll__	abort
reserved_to_roundll__	roundll__	abort
overflow_in_mulf__	mulf__	abort
underflow_in_mulf__	mulf__	± 0
reserved_to_mulf__	mulf__	abort
overflow_in_mull__	mull__	abort
underflow_in_mull__	mull__	± 0
reserved_to_mull__	mull__	abort
overflow_in_scalbf__	scalbf__	abort
underflow_in_scalbf__	scalbf__	± 0
reserved_to_scalbf__	scalbf__	abort
overflow_in_scalbl__	scalbl__	abort
underflow_in_scalbl__	scalbl__	± 0
reserved_to_scalbl__	scalbl__	abort
neg_to_fsqrt__	fsqrt__	abort
neg_to_sqrt__	sqrt__	abort
reserved_to_sqrt__	sqrt__	abort
overflow_in_truncfl__	truncfl__	abort
reserved_to_truncfl__	truncfl__	abort
overflow_in_truncll__	truncll__	abort
reserved_to_truncll__	truncll__	abort

6.5 EXAMPLES

1. Compile and link the `whetstone.c` benchmark program with **libHfp**.

   UNIX (cross-compiler)
   ```
   nmcc whetstone.c -O -KFemulation -o whetstone
   ```

   VMS
   ```
   nmcc /optimize /target=(fpu=emulation) whetstone.c
     nmeld whetstone.obj, gnxdir:crt0, -
           gnxdir:libHm.a,gnxdir:libHc.a,gnxdir:libHfp.a
   ```

2. Emulate the main algorithm with the **libHfp**, while the assembly floating-point code is emulated by the FPEE library. Assume that you have a Fortran main program to be linked with a special purpose assembly routine. This routine performs NS32381 floating point computations in a rounding mode not supported by **libHfp** (for instance, rounding towards + ∞). This can be done by:

   UNIX (cross-compiler)
   ```
   nasm -KF381 special.s
   nf77 -f -KFemulation fft.f special.o
   ```

   UNIX (SYS32)
   ```
   as -KF381 special.s
   nf77 -f -KFemulation fft.f special.o
   ```

   VMS
   ```
   nasm/target=(fpu=381) special
   nf77 /target=(fpu=emulation) fft
   nmeld/exe=fft fft, special, gnxdir:fcrt0, gnxdir:libHF77.a,-
      gnxdir:libHI77.a, gnxdir:libHm.a, gnxdir:libfpe.a,-
      gnxdir:libHc.a, gnxdir:libHfp.a
   ```

3. Replace an exception handler. The normal, default action of **libHfp** is to abort in case of an overflow exception. Say, that you want to avoid this abortion in case of `mulf`, and instead return ∞ or -∞, as required by IEEE 754. Write your own exception handler (see below), and then compile and link as follows:

   UNIX (cross-compiler)
   ```
   nmcc my_program.c ovf_handler.s IEEE_ovf.c \
       -KFemulation -o my_program
   ```

   VMS
   ```
   nmcc /target=(fpu=emulation) my_program.c
   nmcc /target=(fpu=emulation) IEEE_ovf.c
   nasm /target=(fpu=emulation) ovf_handler.asm
   nmeld my_program.obj, ovf_handler.obj, IEEE_ovf.obj, -
       gnxdir:crt0.obj, gnxdir:libHm.a, -
       gnxdir:libHc.a, gnxdir:libHfp.a
   ```

Here is how your exception handler may look:

```
#-------------------------------------------------------------------------
overflow_in_mulf__::
#-------------------------------------------------------------------------
    .set  NUM_REGS,    3              # No. of regs to be pushed
    .set  SP_OFFSET, 4*NUM_REGS+4+4   # sp-offset of first param (4 bytes
                                      # per register plus PSR plus return
                                      # addr to emulation routine, mulf__
    save    [r0,r1,r2]                # SAVE scratch registers
    sprd    upsr, tos                 # SAVE psr
    movd    SP_OFFSET+8(sp), tos      # push second param (addr of dest)
    movd    SP_OFFSET+8(sp), tos      # push first param  (src)

    bsr    _my_ovf_handler            # C routine that computes result
    adjspb    $(-8)                   # clear the operand stack

    lprd    upsr, tos                 # restore psr
    restore [r0,r1,r2]                # restore scratch registers
    ret     $(0)
```

This is how the corresponding `_my_ovf_handler` will look:

```
/*-------------------------------------------------------------------------
 * my_ovf_handler: return plus/minus infinity in case of overflow.
 *-------------------------------------------------------------------------*/
typedef union {
    long     hex;
    float    val;
} FLOAT_OF_32000;

#define SIGNMASK    0x80000000
#define INFINITY    0x7fffffff

my_ovf_handler(src, dest)
    FLOAT_OF_32000 src;
    FLOAT_OF_32000 *dest; /* address of destination */
{
    dest->hex = ((src.hex ^ dest->hex) &SIGNMASK) | INFINITY;
}
```

Appendix A

SERIES 32000 STANDARD CALLING CONVENTIONS

A.1 INTRODUCTION

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

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

A.2 CALLING CONVENTION ELEMENTS

Elements of the standard calling sequence are as follows:

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

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

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

  structures/unions use a multiple of double-words.

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

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

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

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

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

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

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

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

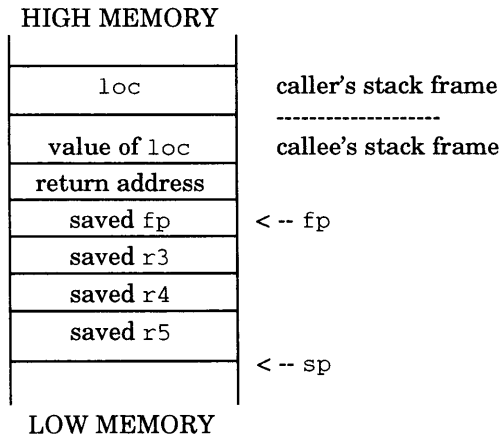
Example:

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

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

The compiler may generate the following code:

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

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

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



INDEX

32 – bit move instruction 5-8
64 – bit move instruction 5-8

A

abort routine 3-2
abs routine 3-3
Absolute
  ceiling 3-13
  floor 3-13
  value 3-13
Absolute value 3-3
Allocate memory in heap 2-19
Argument stack
  in calling sequence A-1
asctime 3-7
Assign buffering to a stream 3-44
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
Bias 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

Calling conventions
  standard A-1
calloc routine 3-24
ceil routine 3-13
clearerr 3-12
Close
  or flush a stream 3-11
Close a file 2-7
close system call 2-7
Compatibility 6-4
Conversion 3-9
Convert
  ASCII to numbers 3-4
  date and time to ASCII 3-7
CPU
  generates 5-6
creat
  system call 2-8
creat call 2-6
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

D

DB library 1-1, 2-3, 3-1
Denormalized numbers 4-8
descriptor table size 2-11
Divide by zero 4-6
  exception 5-12
Division by zero
  exception 5-10
Documentation conventions 1-3
Double-precision numbers 4-3
Double-word format 4-3
Dummy
  call restrictions 1-2
  routines, list of 1-2
Dummy implementations 1-2, 2-3

E

EACCES 2-1
EBADF 2-1
EBUSY 2-2
ecvt routine 3-9
  fcvt 3-9
  gcvt 3-9
EDOM 2-3
EEXIST 2-2
EFAULT 2-2
EFBIG 2-2
EINVAL 2-2
EIO 2-1
EISDIR 2-2
ELOOP 2-3
EMFILE 2-2
ENAMETOOLONG 2-3

ENFILE 2-2
ENODEV 2-2
ENOENT 2-1
ENOMEN 2-1
ENOSPC 2-2
ENOTBLK 2-2
ENOTDIR 2-2
ENOTTY 2-2
Environment
  control functions 4-9
ENXIO 2-1
EPERM 2-1
EROFS 2-2
errno 2-1
Errors
  EACCES 2-1
  EBADF 2-1
  EBUSY 2-2
  EDOM 2-3
  EEXIST 2-2
  EFAULT 2-2
  EFBIG 2-2
  EINVAL 2-2
  EIO 2-1
  EISDIR 2-2
  ELOOP 2-3
  EMFILE 2-2
  ENAMETOOLONG 2-3
  ENFILE 2-2
  ENODEV 2-2
  ENOENT 2-1
  ENOMEN 2-1
  ENOSPC 2-2
  ENOTBLK 2-2
  ENOTDIR 2-2
  ENOTTY 2-2
  ENXIO 2-1
  EPERM 2-1
  EROFS 2-2
  ETXTBSY 2-2
  list of 2-1
ETXTBSY 2-2
Exception
  checking 4-10
  status flag functions 4-9
  status flags 5-9
  trap functions 4-9
  traps 4-9
Exception handler 6-8
Exceptions 4-6, 6-8
exit call 2-6
exit routine 3-10
_exit system call 2-10

F

fabs routine 3-13
fclose routine 3-11
  fflush 3-11
fcvt routine 3-9
fdopen routine 3-14
feof routine 3-12
ferror routine 3-12
  clearerr 3-12
  feof 3-12
  fileno 3-12
fflush routine 3-11
ffs routine 3-5
fgetc routine 3-19
fgets routine 3-21
File descriptor 2-3
File exists 2-2
File table overflow 2-2
File too large 2-2
Filename too long 2-3
fileno 3-12
Find name of a terminal 3-23
Floating-point
  comparisons 5-10
  divide by zero 4-6
  emulation 6-1
  emulation library 1-1
  enhancement and emulation 5-1
  exception handling 5-11
  exceptions 5-11, 6-8
  format 4-3
  inexact result 4-6
  invalid operation 4-6
  library 6-1
  overflow 4-6
  range 4-4
  underflow 4-6
Floating-point numbers
  Reserved operand values 4-6
floor routine 3-13
  ceil 3-13
  fabs 3-13
fopen routine 3-14
  fdopen 3-14
  freopen 3-14
Formatted conversion
  input 3-40
  output 3-29
FPEE 5-1
  and math library integration 5-3
  enhancements to FPU 5-6
  error handling 5-4
  error mechanism 5-4
  library implements 5-6, 5-8
  operation, overview of 5-6
  operations 5-4
  program control 5-8
  rounding modes 5-13

FPEE library
  configurations 5-2
  cross application 4-6
  implements 5-13
  native application 4-6
  support of infinity 4-8
FPEE trap handler 4-9
fperrn.c 5-4
fperrx.s 5-4
fp_getexptn function 4-10
fpinit 5-3
  routine 5-2
fp_procentry function 4-9, 5-3
fprintf routine 3-29
FPU
  bit selects 5-9
  provides 5-6
  Trap 4-6
  trap handler 5-6
  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
freopen routine 3-14
frexp routine 3-17
  ldexp 3-17
  modf 3-17
fscanf routine 3-40
fseek routine 3-18
  ftell 3-18
  rewind 3-18
fsqrt 6-5
FSR 6-5
ftell routine 3-18
Function return value A-2
fwrite routine 3-16

G

gcvt routine 3-9
Generate a fault 3-2
Get a string from a stream 3-21
Get a value of an name 2-12
Get character or word from stream 3-19
getc routine 3-19
  fgetc 3-19
  getchar 3-19
  getw 3-19
getchar routine 3-19
getdtablesize 2-7, 2-16
  system call 2-11
getenv system call 2-12
getpid call 2-3
gets routine 3-21
getw routine 3-19
gmtime routine 3-7
Group ID 2-3

I

IEEE 754 5-1, 5-3, 6-4
  compliance 4-6
  enhancements to FPU 5-6
Implemented IEEE 754 operations, list of 5-7
index routine 3-47
Inexact result 4-6
  exception 5-12
Inexact result exception 5-9
Infinity 4-8
INIT__ 5-3
Initialize the FSR 5-3
initstate routine 3-36
Insert/remove element from queue 3-22
insque routine 3-22
Instruction
  codes, list of 5-5
Integer
  format 4-2
Integer formats
  byte format 4-2
  double-word format 4-3
  word format 4-3
Interrupt handler 6-4
Introduction, high-speed fp emulation
  library 6-1
Introduction, math library 4-1
Invalid
  operation exception 5-10, 5-12
Invalid argument 2-2
Invalid operation 4-6
I/O error 2-1
Is a directory 2-2
isatty routine 3-23

L

ldexp routine 3-17
libfpe.a 5-2
  installed in 5-2
LibHfp 6-1
Library
  handling of exceptions 5-11
localtime routine 3-7
longjmp routine 3-46
lseek system call 2-13

M

malloc routine 3-24
  calloc 3-24
  free 3-24
  realloc 3-24
Mantissa 4-3
Math
  environment variables 4-6
Math argument 2-3
Math environment function 4-9
Math library 1-1, 4-1
  provides 4-6, 5-3, 5-4
Math library functions
  access 4-10
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
  values, table of 4-4
modf routine 3-17
Mount device busy 2-2
Move read/wrxite pointer 2-13

N

Native
  application FPEE integration 5-2
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 5-7
NS32081/NS32381 and FPEE 5-7
NS32381
  has 5-7
  implements 5-7
NS32381 instructions 6-5
NS32580 instructions 6-5
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 stream 3-14
open
  system call 2-15
open call 2-6
Operating system
  call simulation 1-2
Overflow 4-6, 6-8
  exception 5-12
  exceptions returned values, list of 5-14
  on conversion from float to integer 5-10
  signaled 5-10

P

Permission denied 2-1
perror routine 3-28
printf routine 3-29
  fprintf 3-29
  sprintf 3-29
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

qsort routine 3-35
Quicker sort 3-35
Quiet NAN 4-7

R

Random number generator 3-36
random routine 3-36
  initstate 3-36
  setstate 3-36
  srandom 3-36

Read input 2-17
Read mode 2-3
read system call 2-17
Read-only file system 2-2
realloc routine 3-24
re_comp routine 3-38
Reentrancy 6-4
re_exec routine 3-38
Registers
  saving A-2
Regular expression handler 3-38
Regular expression routines
  re_comp 3-38
  re_exec 3-38
Remove directory entry of a file 2-20
remque routine 3-22
Rename input 2-18
rename system call 2-18
Reposition a stream 3-18
Reserved operand values 4-2
  denormalized numbers 4-8
  infinity 4-8
  Not a Number (NAN) 4-7
Reserved operand values and operations 4-6
Return codes 2-1
Returned value A-2
rewind routine 3-18
rindex routine 3-47
Rounding mode 5-9, 6-4
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
Routines for changing generators 3-36

S

Saving registers A-2
sbrk system call 2-19
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
Signals 4-6
Simulated system calls 2-6
Single-precision numbers 4-3
Split into mantissa and exponent 3-17
sprintf routine 3-29
sqrt 6-5
srandom routine 3-36
sscanf routine 3-40
Stack
in calling sequence A-1
Standard calling convention A-1
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 1-2
System calls 1-1, 2-1, 2-3
close 2-7
creat 2-6, 2-8
dummy implementations 1-2
exit 2-6
_exit 2-10
getdtablesize 2-11
getenv 2-12
implemented 1-2
lseek 2-13
open 2-6, 2-15
read 2-17
rename 2-18
routines that don't require 2-4
routines that use simulated 2-5
sbrk 2-19
simulated 2-6
summary of 2-3
unlink 2-20
write 2-22

T

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
handler 4-7
type 5-9

U

Underflow 4-6, 6-8
Underflow exception 5-9, 5-12
ungetc routine 3-50
unlink system call 2-20
User ID 2-3

V

Values from functions 2-1
Volatile 6-4

W

Word format	4-3
Write mode	2-3
Write on a file	2-22
write system call	2-22

Series 32000®

GNX — Version 4.4
COFF Programmer's Guide

Customer Order Number 424010507-004

June 1992

REVISION RECORD

VERSION	RELEASE DATE	SUMMARY OF CHANGES
4.0	May 1990	First Release. *Series 32000*® GNX — Version 4.0 COFF Programmer's Guide
4.1	Sept 1990	No updates. Version synchronization
4.2	Feb 1991	No updates. Version synchronization
4.3	Aug 1991	No updates. Version synchronization
4.4	June 1992	MS-DOS support added

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.

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

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 2-8
2.4.2 Section Header Flags 2-9
2.4.3 .bss Section Header 2-9

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 6-14
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/MS-DOS host.
For the options to the assembler, linker, and compiler for your specific host, see the *Series 32000 GNX — Version 4 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.

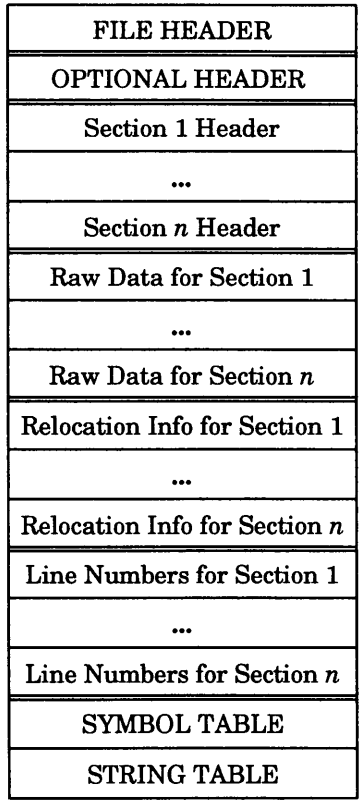


Figure 1-1. GNX Common Object File Format

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/MS-DOS 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 4 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	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 processed 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 4 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.

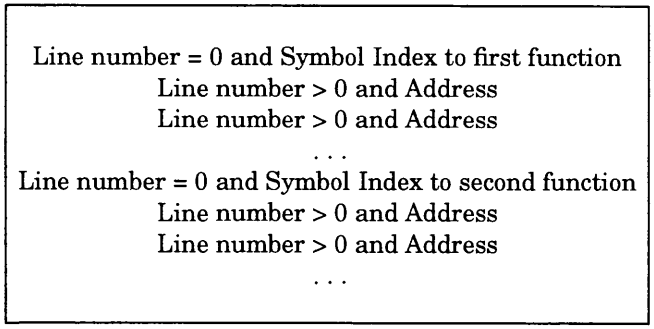


Figure 5-1. Line Number Grouping

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.

see page 6-25 to find the source line number for a function.

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.

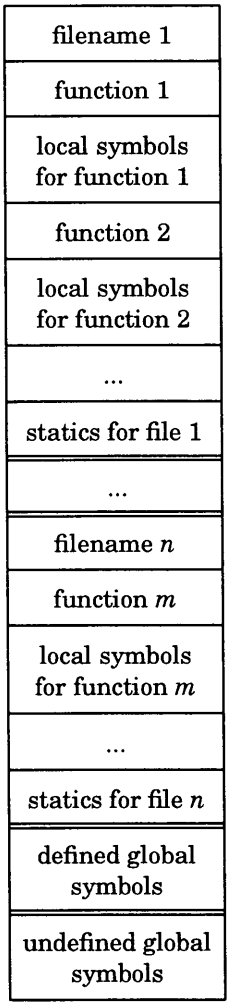


Figure 6-1. GNX COFF Symbol Table

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.

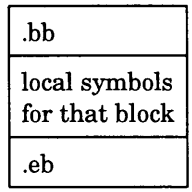


Figure 6-2. Special Symbols (.bb and .eb)

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.

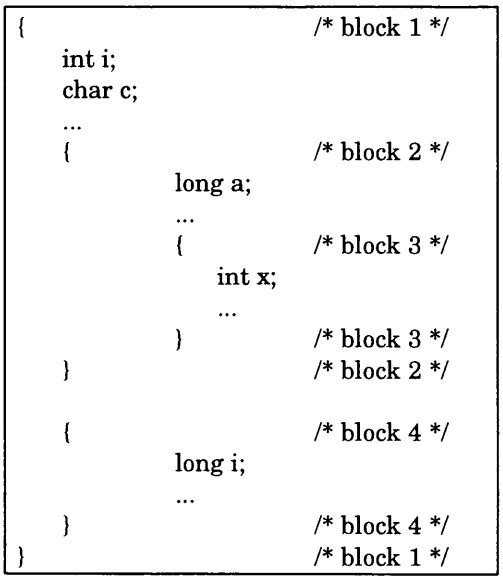


Figure 6-3. Nested Blocks

An example of a symbol table is shown in Figure 6-4.

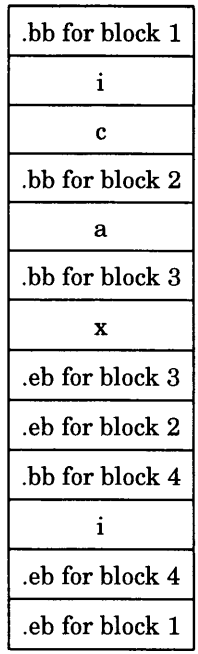


Figure 6-4. Symbol Table

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.

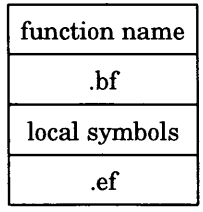


Figure 6-5. Symbols for Functions

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.

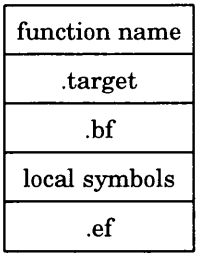


Figure 6-6. The Special Symbol .target

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.

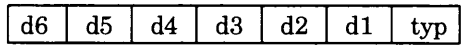
Table 6-6. Section Number and Storage Class

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:



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 *.x*fake, 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 0^{th} 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

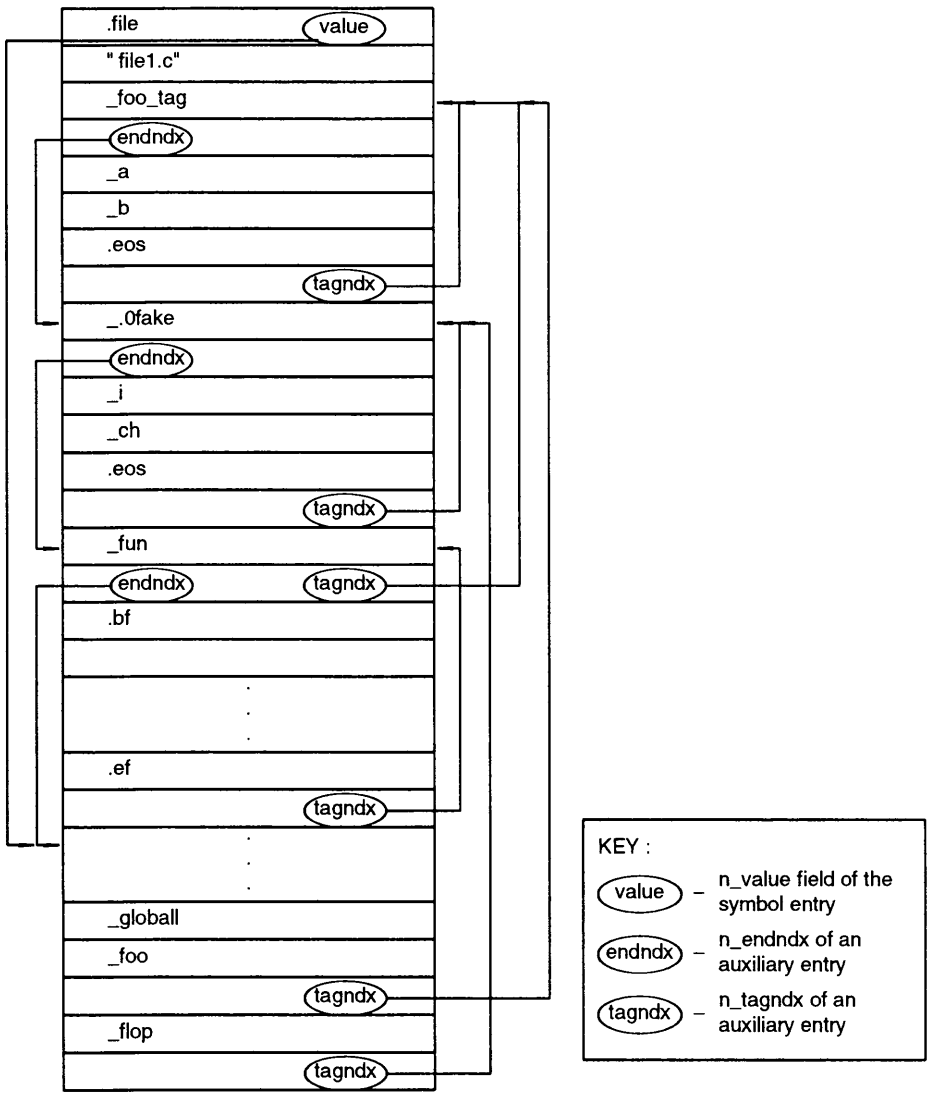
The C language structure declaration for an auxiliary symbol table entry may be found in the `syms.h` header file.

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 ()
    {
    }
```



GG-01-0-U

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 2-9

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
  figure 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
Link editor 6-13
.link section 3-1
Linked list structures in the symbol table,
  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
  loading a file with 3-1

N

Name field, figure of 6-9
Nested blocks, figure of 6-6
n_value 6-28

O

Optional header 2-1, 2-4, 3-1
  contents of 2-4
  guidelines for use 2-5
Optional header flags 2-6
  list of 2-7
Optional header magic numbers 2-5
  list of 2-6
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, 2-9, 3-1
  contents of 2-8
  use of 2-8
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, 6-28
  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, 6-1, 6-6
  list of entry fields 6-8
  order of symbols 6-2
Symbol table entries 6-7, 6-20
Symbol value field 6-11
Symbols and functions 6-6
Symbols for functions, sequence of 6-6
syms.h 6-20

T

Tag index 6-28
Tagnames 6-22
.target 6-7
Target machine 1-4
.text section 3-1
Type entries 6-15
  basic 6-15
  derived 6-15
  fundamental 6-15
  list by storage class 6-19
  storage classes 6-18
Type field 6-15

U

Unions 6-23

V

Virtual address 1-3
Vstamp 2-5

X

.xfake 6-1

NATIONAL SEMICONDUCTOR CORPORATION
GNX LANGUAGE TOOLS BINARY USER AGREEMENT

CAREFULLY READ ALL OF THE TERMS AND CONDITIONS OF THIS LICENSE AGREEMENT PRIOR TO OPENING THIS PACKAGE. BY OPENING THIS SEALED PACKAGE, YOU, THE END USER, ACCEPT THE TERMS AND CONDITIONS OF THIS AGREEMENT. IF YOU DO NOT AGREE TO BE BOUND BY THE TERMS AND CONDITIONS OF THIS AGREEMENT, DO NOT OPEN/BREAK THE SEAL ON THIS PACKAGE. PROMPTLY RETURN THE UNOPENED PACKAGE TO THE PLACE WHERE YOU OBTAINED IT FOR A FULL REFUND.

---

1. **GRANT OF LICENSE:** In consideration of the payment of the License Fee, which is a part of the price you paid for the product, National Semiconductor Corporation, as Licensor ("NSC"), grants you, as Licensee, a personal, nontransferable, nonexclusive right to use and display the enclosed Licensed Software program on a single CPU.

2. **OWNERSHIP OF SOFTWARE:** As the Licensee, you own the media on which the Licensed Software has been recorded. However, NSC retains title and ownership of the Licensed Software. Hence, whole or partial copies of the Licensed Software, in any form or on any media made by Licensee shall also be the property of NSC.

3. **COPY RESTRICTIONS:** Both the Licensed Software and the accompanying printed materials are copyrighted. Unauthorized copying of the Licensed Software or any portion thereof, including Licensed Software that has been modified, merged, or included with other software programs or of the printed materials is expressly prohibited. However, you may make a reasonable number of copies of the Licensed Software solely for back-up or archival purposes. In any such copies, you must reproduce all copyright notices and other identifying or restrictive legends that appear on the Licensed Software as received.

4. **TRANSFER RESTRICTIONS:** This Software has been licensed to you, the end user, and may not be transferred or assigned to anyone else without obtaining the prior written consent of NSC. An authorized recipient of this Licensed Software must agree to be bound by the terms and conditions of this Agreement.

5. **USE RESTRICTIONS:** Licensee may physically transfer the Licensed Software from one of Licensee's CPUs to another provided that the Licensed Software is used on only one CPU at a time. You may not electronically transfer Licensed Software from one CPU to another.

6. **REGISTRATION CARD:** Licensee must complete and return or telefax the enclosed Software Registration Card within seven (7) days of opening this package. Registration Cards on file will insure prompt service to Licensee.

7. **TERMINATION:** Provided that you complete and return the Registration Card, this License Agreement is perpetual until terminated. You may terminate this Agreement, at any time, by either returning the Licensed Software, printed materials and all copies thereof, or certifying the destruction of same to NSC. This Agreement will terminate automatically if you fail to comply with any of its provisions. In that event, you agree to return the Licensed Software, printed materials and all copies thereof or certify their destruction.

8. **DISCLAIMER OF WARRANTY:** NSC provides the Licensed Software "AS IS" without warranty of any kind. Further, NSC shall not be liable to Licensee or Licensee's customers for any direct, indirect, special, incidential, consequential, or other damages arising from the use of the Licensed Software. NSC expressly disclaims any impled warranties of merchantability and/or fitness for a particular purpose.

9. **EXPORT:** Licensee shall not export or re-export Licensed Software without the appropriate United States Department of Commerce or Department of State and foreign government licenses.

10. **APPLICABLE LAW:** This License Agreement shall be governed by the laws of the State of California.

Should you have any questions concerning this License Agreement please contact NSC in writing at:

National Semiconductor GmbH
Attn: Office Automation Software Marketing M/S 024
Industriestrasse 10
8080 Fuerstenfeldbruck
Federal Republic of Germany
Telefax: Germany + 8141 - 103 304