Tru64 UNIX

Object File and Symbol Table Format Specification

Part Number: ObjSpec

August 2000

Product Version: Tru64 UNIX Version 5.1

This book describes the organization and usage of object files and images that are built on Tru64 UNIX systems.

Compaq Computer Corporation
Houston, Texas

Contents

6 Dynamic Loading Information

10 Programming Examples

Index

Figures

Tables

About this Manual

This book describes the organization and usage of object files and images that are built on Tru64 UNIX systems.

Audience

This manual is targeted for compiler and debugger writers and other developers who must access or manipulate object files. A familiarity with basic program development and symbol table concepts is assumed.

Necessity

This manual is designed to fill a need for technical information for back-end developers working on the Tru64 UNIX operating system. It supplements or replaces information that has previously been available in the *Assembly Language Programmer's Guide*.

Organization

This manual is organized as follows:

Chapter 1	Provides background information on the development environment and describes the high-level organization and usage of object files.
Chapter 2	Describes the header sections of the object file.
Chapter 3	Describes the contents of the "raw data" sections of the object file.
Chapter 4	Describes the relocation process and related structures stored in the object file.
Chapter 5	Describes the symbol table.
Chapter 6	Describes the object file sections containing dynamic loading information.
Chapter 7	Describes the format and usage of the object file comment section.
Chapter 8	Describes the archive file format.
Chapter 9	Provides examples that illustrate symbol table representations.
Chapter 10	Provides programming examples to illustrate object file and symbol table access.

Related Documents

This manual discusses the object file format from the perspective of tools that produce or use object files. Understanding the purpose of these tools is a prerequisite, but this information is touched upon briefly in this document. The primary source for information on system programs in the development environment is the *Programmer's Guide*. The default debugger on Tru64 UNIX is the ladebug debugger, which is treated separately in the *Ladebug Debugger Manual*.

The contents of object files are also tied to the Alpha architectural implementation. The *Assembly Language Programmer's Guide* provides an architectural overview that focuses on assembly level instructions and directives. Architectural documentation is also available in the *Alpha Architecture Reference Manual*.

The *Calling Standard for Alpha Systems* also contains material related to this manual. The calling standard defines the interface and other requirements for procedure calls on Alpha platforms.

Icons on Tru64 UNIX Printed Books

The printed version of the Tru64 UNIX documentation uses letter icons on the spines of the books to help specific audiences quickly find the books that meet their needs. (You can order the printed documentation from Compaq.) The following list describes this convention:

G	Books for general users
S	Books for system and network administrators
P	Books for programmers
D	Books for device driver writers
R	Books for reference page users

Some books in the documentation help meet the needs of several audiences. For example, the information in some system books is also used by programmers. Keep this in mind when searching for information on specific topics.

The *Documentation Overview* provides information on all of the books in the Tru64 UNIX documentation set.

Reader's Comments

Compaq welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

- Mail:

  Compaq Computer Corporation
  UBPG Publications Manager
  ZKO3-3/Y32
  110 Spit Brook Road
  Nashua, NH 03062-2698

  A Reader's Comment form is located in the back of each printed manual. The form is postage paid if you mail it in the United States.

Please include the following information along with your comments:

- The full title of the book and the order number. (The order number is printed on the title page of this book and on its back cover.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate Compaq technical support office. Information provided with the software media explains how to send problem reports to Compaq.

Conventions

The following conventions are used in this manual:

%	
$	A percent sign represents the C shell system prompt. A dollar sign represents the system prompt for the Bourne, Korn, and POSIX shells.
#	A number sign represents the superuser prompt.
% **cat**	Boldface type in interactive examples indicates typed user input.
file	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|]	
{ | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
⋮	A vertical ellipsis indicates that a portion of an example that would normally be present is not shown.
cat(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the cat command in Section 1 of the reference pages.
Return	In an example, a key name enclosed in a box indicates that you press that key.
Ctrl/*x*	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, Ctrl/C).
Alt *x*	Multiple key or mouse button names separated by spaces indicate that you press and release each in sequence. In examples, each key in the sequence is enclosed in a box (for example, Alt Q).
Colored ink	Colored ink indicates information that you enter from the keyboard or a screen object that you must choose or click on.

1

Introduction

This specification is the official definition of the object file and symbol table formats used for Tru64 UNIX object files. It also describes the legal uses of the formats and their interpretation.

New or retired features of the object file and symbol table formats are identified throughout this document by Version Notes. Table entries and structure fields may also be marked with a range of version stamps in parenthesis and bold type. This indicates that the marked feature is valid for the indicated range of operating system or format versions. The examples that follow illustrate the three kinds of version stamps and the four types of ranges.

(V5.1 -) Indicates that the marked feature is valid in Tru64 UNIX for releases V5.1 and greater.

(- OV3.12) Indicates that the marked feature is valid for all object format versions up to and including V3.12.

(SV3.10 - SV3.13) Indicates that the marked feature is valid for symbol table format versions V3.10 through V3.13 inclusive.

(OV3.13) Indicates that the marked feature is only valid for object format version V3.13.

Operating system, object format, and symbol table format versions (see Section 1.4.5) will be used to identify new or retired features. Compiler and tool versions can also affect what features may be used or supported, but this information will be provided in documentation accompanying the compiler or tool.

This document treats in detail the file formats for object files and archive files. These files are described as follows:

Object File An object file is a binary file produced by a compiler, assembler, and/or linker from high-level-language source files or other object files. Object files can be executable programs, shared libraries, or relocatable object files. One or more relocatable object files can be linked together to form executable programs or shared libraries.

Symbol Table A symbol table is contained within an object file. It is used to convey linking and debugging information describing the contents of the object file.

Archive File An archive file is a single file which contains many object or text files that are managed as a group. Archive files can serve as libraries that are searched by the linker. A special symbol table is included in the archive file for this purpose. The archiver (`ar`(1)) is the tool used to create and update archive files.

Tools that create, use, or otherwise interact with object or archive files should conform to the formatting and usage conventions outlined in this specification.

1.1 Definitions

This section defines terms that are used throughout this document.

address — If not otherwise specified, an address is a location in virtual memory.

alignment — The positioning of data items or object file sections in memory so that the starting address is evenly divisible by a given factor.

absolute file offset — See file offset.

API — Application Programming Interface.

application — A user-level program.

base address — The lowest-numbered location of an object file mapped in virtual memory.

byte boundary — The alignment factor.

common storage class symbol — A global symbol that can be legally multiply defined. Storage space for common storage class symbols is typically allocated when relocatable object files are linked.

constant — A variable or value that cannot be overwritten.

dynamic executable — A call-shared application or program. A dynamic executable is linked with shared libraries and loaded by the dynamic loader.

dynamic loader — A system program that maps dynamic executables and shared libraries into virtual memory so that they can be executed.

entry point — The first instruction that is executed in a program or procedure.

executable — An object file that can be executed. Also referred to as a program, image, or executable object. Executables can be static or dynamic.

file offset — The distance in bytes from the beginning of an on-disk file to an item within the file. Also referred to as an absolute file offset.

hashing — A search technique typically used in performance-sensitive programs.

image	A program mapped in memory for execution. A shared process image includes mappings of shared libraries used by the program.
linker	The system utility `ld`. This utility is the primary producer of executable object files and shared libraries.
literal	A value represented directly.
locally stripped	Stripped of "local" symbol information used primarily for debugging.
namespace	A scope within which symbol names should all be unique.
relative file offset	The distance in bytes from a given position in an on-disk file to another item within the file.
relative index	An index represented as an offset from a base index.
relocatable object	An object file that includes the information required to link it with other object files.
section	The primary unit of an object file.
segment	A portion of an object file that consists of one or more sections and can be loaded into virtual memory.
shared library	An object file that provides routines and data used by one or more dynamic executables.
shared object	A dynamic executable or shared library.
static executable	An object file that contains all of the executable code and data required to create a runnable program image.
symbol preemption	A mechanism by which all references to a multiply defined symbol are resolved to the same instance of the symbol.

1.2 History and Applicability

The object file format described in this specification originated from the System V COFF (Common Object File Format). Implementation-dependent varieties of the COFF format are used on many UNIX systems. Tru64 UNIX has altered and extended the object file format to serve as the basis for program development on Alpha systems. This extended version of COFF is referred to in this document as eCOFF.

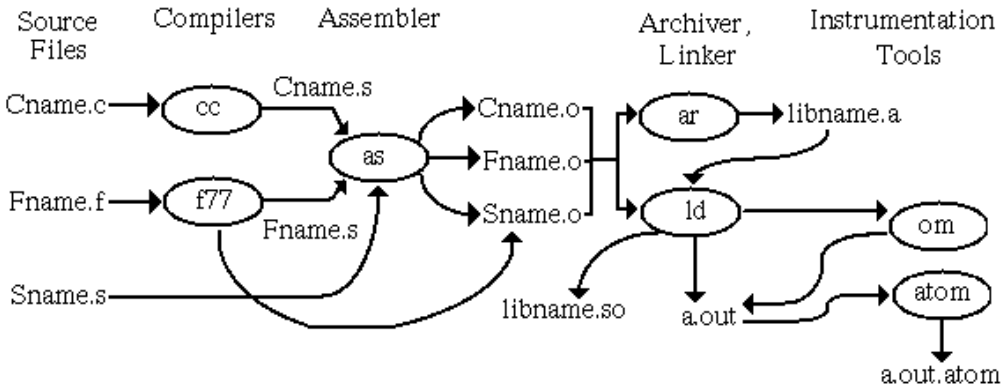
All systems based on the Alpha architecture and running Tru64 UNIX employ the eCOFF object file format.

1.3 Producers and Consumers

Many tools interact with objects and archives in the development environment. Object file producers create object files, and object file consumers read object files. A tool may be both a producer and a consumer. Figure 1–1 provides one view

of the program development process from source files through executable object file production.

Figure 1–1: Object File Producers and Consumers



A summary of the functions of relevant system utilities and their relationship to objects and archives follows. Detailed information is available in reference pages.

1.3.1 Compilers

Compilers are programs that translate source code into either intermediate code that can be processed by an assembler or an object file that can be processed by the linker (or executed directly). Accordingly, compilers may be direct or indirect producers of object files, depending on the compilation system. The compiler creates the initial symbol table.

1.3.2 Assemblers

Assemblers also produce object files. An assembler converts a compiler's output from assembly language (the intermediate form) into binary machine language. The result is traditionally a non-executable object file (`.o` file). The assembler lays out the sections of the object file and assigns data elements and code to the various sections. It also lays the groundwork for the relocation process performed by linkers.

1.3.3 Linkers

A linker (or link-editor) accepts one or more object files as input and produces another object file, which may be an executable program. The linker performs relocation fixups and symbol resolution. It merges symbolic information and searches for referenced symbols in shared libraries and archive libraries. Linkers are producers and consumers of object files, and consumers of archive files.

The selection of command-line options determines what type of object the linker produces. A final link produces an executable object file or shared library. A partial link produces a relocatable object that can be included in a future link.

1.3.4 Loaders

Loaders (sometimes referred to as dynamic linkers) load executable object files and shared libraries into system memory for execution. A loader may perform dynamic relocation and dynamic symbol resolution. It may also provide run-time support for loading and unloading shared objects and on-the-fly symbol resolution. The loader is a consumer of executable object files and shared libraries.

1.3.5 Debuggers

Debuggers are utilities designed to assist programmers in pinpointing errors in their programs. Debuggers are object file consumers, and they rely heavily on the debug symbol table information contained in object files.

1.3.6 Object Instrumentation Tools

Object instrumentation tools are both consumers and producers of object files. Their input is an executable object and, possibly, the shared libraries used by that executable object. Their output is the instrumented version of the executable program. Instrumentation involves modifying the application by adding calls to analysis procedures at basic block, procedure, or instruction boundaries. Depending on the tool, the aim may be to optimize the program or gather data to enable future optimizations.

1.3.6.1 Post-Link Optimizers

The `om` and `spike` object modification tools perform post-link optimizations such as removal of unneeded instructions and data.

The `cord` tool is a post-link tool that rearranges procedures in an executable file to facilitate improved cache mapping.

These tools are object file consumers and producers.

1.3.6.2 Profiling Tools

UNIX profiling tools (such as Compaq's programmable profiling and program analysis tool, `Atom`) are object file producers and consumers. These tools examine an executable object and the shared libraries it uses and report information such as basic block counts and procedure calling hierarchies. They may also restructure the program to improve performance. Output includes files that store profiling data generated during execution of the instrumented application.

1.3.7 Archivers

An archiver is a tool that produces and maintains archive files. It is a producer and a consumer of archive files and a consumer of object files.

1.3.8 Miscellaneous Object Tools

1.3.8.1 Object Dumpers

Tools are available that read object files and dump (print) their contents in human-readable form. Examples are `nm`, `odump`, `stdump`, and `dis`. These tools are object file consumers.

1.3.8.2 Object Manipulators

The tools `ostrip` and `strip` reduce the size of an object file by removing certain portions of the file. The `mcs` tool modifies the comment section only. These tools are both consumers and producers of object files.

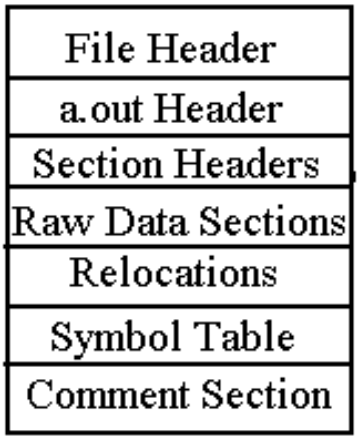
1.4 Object File Overview

1.4.1 Main Components of Object Files

This document is organized to correspond to a conceptual breakdown of an object file's contents. The main components of an object file are described briefly in the remainder of this section.

A high-level view of the eCOFF object file contents is depicted in Figure 1–2.

Figure 1–2: Object File Contents



1.4.1.1 Object File Headers

Header structures serve as a roadmap for navigating portions of the object file. They provide information about the size, location, and status of various sections and about the object as a whole. See Chapter 2 for more information.

1.4.1.2 Instructions and Data

Instructions and data are located in loadable segments of the object file. Instructions consist of all executable code. Data consists of uninitialized and initialized data, constants, and literals. Instructions and data are laid out in sections that are arranged into segments. The segments are then loaded to form part of the program's final image in memory. See Chapter 3 for more information.

1.4.1.3 Object File Relocation Information

The purpose of relocation is to defer writing the address-dependent contents of an object file until link time. Relocation entries are created by the compiler and assembler, and the necessary address adjustments are calculated by the linker. Information relevant to relocation is stored in section relocation entries and in the symbol table. In some instances, the loader subsequently performs dynamic relocation. See Chapter 4 and Chapter 6 for more details.

1.4.1.4 Symbol Table

The symbol table contains information that describes the contents of an object file. Linkers rely on symbol table information to resolve references between object files. Debuggers use symbol table information to provide users with a source language view of a program's execution and its execution image. See Chapter 5 for more details.

1.4.1.5 Dynamic Loading Information

Dynamic sections are utilized by the loader to create a process image for an executable object. These sections are present in shared object files only. Information is included to enable dynamic symbol resolution, dynamic relocation, and quickstarting of programs. See Chapter 6 for more details.

1.4.1.6 Comment Section

The comment section is a non-loadable section of the object file that is divided into subsections, each containing a different kind of information. This section is designed to be a flexible and expandable repository for supplemental object file data. See Chapter 7 for more information.

1.4.2 Kinds of Object Files

There are four principal types of object files:

Relocatable objects	Relocatable objects are object files that contain full relocation information. They are usually not executable. Pre-link producers (generally compilers and assemblers) always generate relocatable objects. The linker can also generate relocatable objects, but does not do so by default. See Chapter 4 for more details.
Static (non-shared) executables	An object file is executable if it has no undefined symbol references. Executable objects can be static or dynamic. Static executables are object files that are linked `-non_shared`. They use archive libraries only. They are fully resolved at link time and are loaded by the kernel's program execution facility.
Dynamic (call-shared) executables	Dynamic executables are object files that are linked `-call_shared`. They may use shared libraries, archive libraries or both. A dynamic executable is the compilation system's default output. The system loader performs dynamic linking, dynamic symbol resolution, and memory mapping for dynamic executables and the shared libraries they use.
Shared libraries	Shared libraries are object files that provide collections of routines that can be used by dynamic executables. Although it contains executable code, a shared library by itself is not usually executable. Advantages of shared libraries include the ability to use updated libraries without relinking and a reduction in disk requirements. The reduction in disk requirements is achieved by providing a single copy of routines and data that might otherwise be duplicated in many executable object files.

Object file types can often be differentiated by their file name extension. Typically, relocatable objects have a `.o` file extension and shared libraries have a `.so` file extension. The default name for an executable object file is `a.out`. User-named executable files often do not have an extension.

It is important to be aware of which type of file is under discussion because the usage, content, and format of each kind of object file can vary significantly.

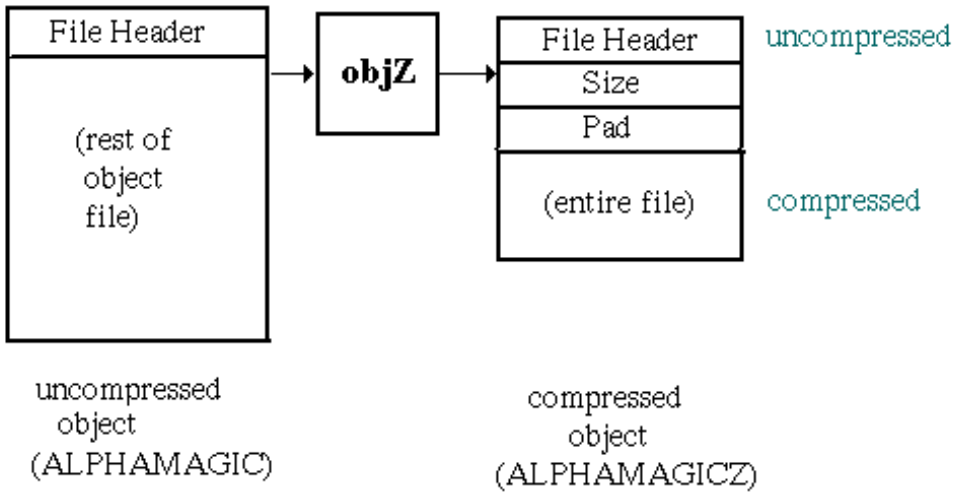
1.4.3 Object File Compression

File compression is used widely on all kinds of files to save disk space. Similarly, object files can be compressed to save space. However, not all objects are candidates for compression and not all tools that handle objects also support compressed object files.

Decompressed data can be, at most, eight times the size of the compressed data. This rate of compression is the best case possible. At worst case, a compressed object will actually be larger than the decompressed version. Typically, however, a reduction of 50% to 75% in size is achieved.

When an object is compressed, the file header in uncompressed form precedes the compressed object file. The uncompressed file header's magic number indicates whether the remainder of the file contains a compressed object.

Figure 1–3: Object File Compression



The value of "size" is the size of the uncompressed object in bytes. The archiver uses the "pad" value to indicate the bytes of padding it inserted. Both fields are 8-byte unsigned integers.

The most commonly compressed objects are archive members. Both the archiver and the linker support compressed objects used as archive members.

Executable objects and shared libraries cannot be compressed because the dynamic loader does not support compressed objects. To decompress an image, the loader would need to allocate space where it could write the decompressed image. Serious system penalties would be incurred because no part of the image would be shareable. However, a compressed object file can subsequently be decompressed and then loaded; this might be a way to temporarily save disk space in some circumstances.

The tool `objZ` is a Tru64 UNIX compression utility designed for object files. See the `objZ`(1) man page for details.

1.4.4 Object Archives

Archiving is a method used to enable manipulation of a large number of files as a single group, which may ease the task of file management. Any file can be

archived. However, the archive files of primary interest in program development are archived object files that are used as libraries for static executables.

Object archives provide a means of working with a collection of objects simultaneously. System libraries such as `libc.a` and `libm.a` are object archives. Each library collects a set of related objects which provide a service in the form of callable APIs. Benefits of using archives in this fashion include the grouping of related functions and shorter build commands.

Another benefit of archive libraries is selective linking, whereby the linker extracts only needed objects from a library, instead of mapping the entire library with the image. For example, suppose the library `libEx.a` contained the objects `x.o`, `y.o`, and `z.o`. If the executable `a.out` depended on `x.o` to define a referenced symbol, but not on the other objects in the archive, only `x.o` would become part of the final executable object.

Another typical use for object archives is to subdivide large builds into subsystems, each of which is implemented as an archive that is eventually included in the final link.

Most tools that read objects will also read object archives. The linker applies special semantics in its handling of object archives, while other utilities treat an object archive as simply a list of object files.

Object archive members can also be compressed. In this case, each object that is an archive member is compressed as shown in Section 1.4.3. The archive file's administrative information is not compressed. Also, an archive file may contain both compressed and uncompressed file members.

More information on archives can be found in Chapter 8.

1.4.5 Object File Versioning

The object file and symbol table formats are versioned. This versioning scheme is independent of the operating system or hardware versions. It is not designed to be visible to end-users.

The object file and symbol table versions are each stored as a two-byte version stamp, with major and minor components of one byte each. The object file version is stored in the `vstamp` field of the `a.out` header, and the symbol table version is stored in the symbolic header's `vstamp` field. The minor version is incremented when new features or compatible structure changes are introduced. The major version is incremented when an incompatible or semantically very significant change is made.

The object file version stamp covers the following structures:

- File header (`filehdr.h`)
- `a.out` header (`aouthdr.h`)
- Section header (`scnhdr.h`)
- Relocations (`reloc.h`)
- .comment data (`scncomment.h`)
- Dynamic loading information structures (`coff_dyn.h`)

The symbol table version covers all symbol table structures and values defined in the header files `sym.h`, `symconst.h`, and `linenum.h`.

The object file and symbol table versions can differ.

This document covers object file format V3.13 and symbol table format V3.13.

Tool-specific version information for object file consumers may also be stored in the on-disk object file. If present, this information is stored in the comment section. See Chapter 7 for details.

1.4.6 Object File Abstract Data Types

A consistent set of basic abstract data types are used to build object file, symbol table, and dynamic loading structures. These names are defined in the header file `coff_type.h`.

The use of abstract types for all elements of these structures facilitates cross-platform builds. To build a tool to run on another platform, redefine the COFF basic abstract types for the new platform. This is done by inserting the new definitions and "`#define ALTERNATE_COFF_BASIC_TYPES`" prior to any object file or symbol table header files.

Table 1–1: COFF Basic Abstract Types

Name	Size	Alignment	Purpose
`coff_addr`	8	8	Unsigned program address
`coff_off`	8	8	Unsigned file offset
`coff_ulong`	8	8	Unsigned long word
`coff_long`	8	8	Signed long word
`coff_uint`	4	4	Unsigned word
`coff_int`	4	4	Signed word
`coff_ushort`	2	2	Unsigned half word
`coff_short`	2	2	Signed half word
`coff_ubyte`	1	1	Unsigned byte
`coff_byte`	1	1	Signed byte

Another data representation that is currently used exclusively in the optimization symbol table is LEB (Little Endian Byte) 128 format. This is a variable-length format for numeric data. The low-order seven bits of each LEB byte are interpreted as an integer value. The high bit, if set, indicates a continuation to the next byte. An LEB byte is illustrated in Figure 1–4. This format takes advantage of the likelihood that most numbers will be small. To form a large number, concatenate the 7-bit segments of the LEB128 bytes, as shown in Figure 1–5.

Figure 1–4: LEB 128 Byte

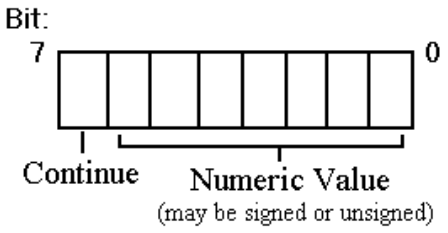
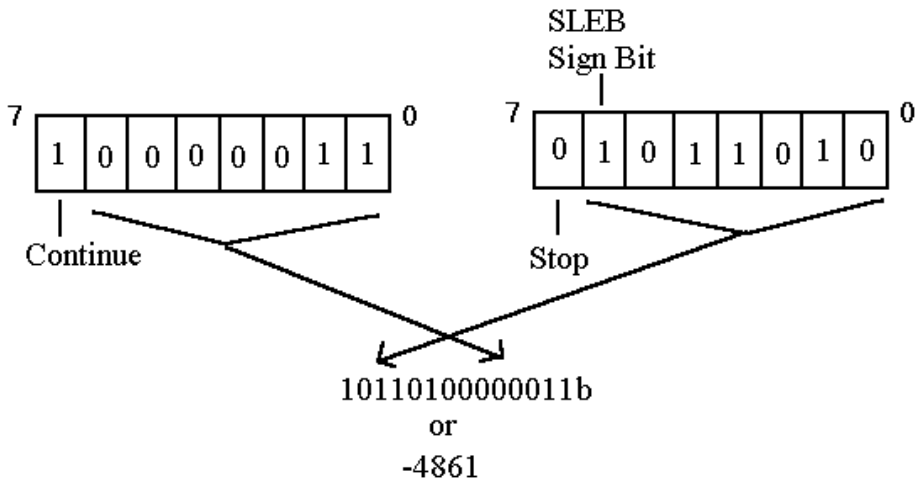


Figure 1–5: LEB 128 Multi-Byte Data



A value represented in LEB 128 format may be signed (SLEB) or unsigned (LEB). The second-highest bit in the final byte of an SLEB value is the sign bit. This means that the signed value has to be propagated only within one byte.

The program example in Section 10.2 includes subroutines that read LEB 128 data.

1.5 Source Language Support

Object files originate from source files that may be coded in any of several high-level languages. The Tru64 UNIX eCOFF object file format supports the programming languages C, C++, Fortran, Bliss, Fortran90, Pascal, Cobol, Ada, PL1, and assembly. The choice of source language primarily impacts the symbol table, which includes the type and scope information used by the debugger. See Section 5.3.2 for more information.

The UNIX system is closely tied to the C programming language, and many tools that work with objects do not fully support non-C languages. Reference the specific tool's documentation for details.

1.6 System Dependencies

Certain characteristics of the object file format are dependent on the Tru64 UNIX operating system. This section highlights those features and provides references to more detailed information.

The address space and image layout information covered in Chapter 2 are dependent on the operating system's virtual memory organization.

The kernel's virtual memory manager ensures that multiple processes can share all text and data pages. As soon as a process writes to one of those pages, it receives its own copy of that page. Because text pages are always mapped read-only, they are always shared for the lifetime of the process.

The virtual memory manager uses additional shareable pages, known as Page Table pages, to record the memory layout of a process. The linker's default address selection and the system library addresses are designed to maximize sharing of page table pages, which are implemented as "wired" memory, a limited system resource.

As part of this implementation, the text and data segments of shared libraries are usually separated in the address space. This separation allows many shared library text segments to be mapped in one area of memory. The Page Table pages

used to describe an area of memory containing only text segments are shared by all processes that map one or more of those text segments into their address space. This sharing can result in significant savings in wired memory used by the system.

The GP-relative addressing technique is unique to Tru64 UNIX. See Section 3.3.2.

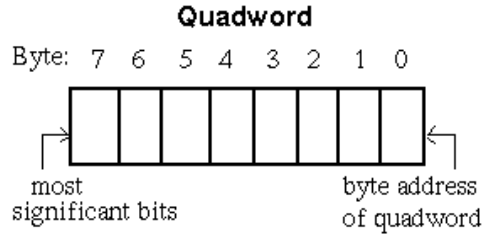
The operation of the system dynamic loader as described in Chapter 6 is system-dependent. Other loaders may behave differently.

The discussion of system shared library implementation using weak symbols is unique to Tru64 UNIX. See Section 6.3.4.1.

1.7 Architectural Dependencies

The 64-bit Alpha architecture defaults to using the little-endian byte-ordering scheme. In little-endian systems, the address of a multibyte data element is the address of its least significant byte, and the sign bit is located in the most significant bit. Bytes are numbered beginning at byte 0 for the lowest address byte, as shown in Figure 1–6.

Figure 1–6: Little Endian Byte Ordering



A big-endian byte order can be inferred by assuming all structure fields would be byte-swapped in a big-endian object. For example, big-endian byte order can be inferred from Figure 1–6 by reversing the byte numbering and moving the "byte address of quadword" label to the new location of byte 0. In a big-endian representation, bit numbering within a byte is also reversed. This document will only identify differences in the big-endian representation that either do not follow convention or are not obvious.

As discussed in Section 2.3.5, hardware constraints dictate text and data alignment. Unaligned references can cause fatal errors or negatively impact performance. For instance, on Alpha systems, dereferencing a pointer to a longword- or quadword-aligned object is more efficient than dereferencing a pointer to a byte- or word-aligned object. Special instructions exist for unaligned data memory accesses. The default assumption is that data is aligned.

TASO, the Truncated Address Space Option, is a migration path for applications with 32-bit assumptions onto 64-bit Alpha platforms. This topic is discussed in Section 2.3.3.2.

Relocation entries are heavily dependent on the Alpha instruction format. See Chapter 4 for details.

See the *Assembly Language Programmer's Guide* and *Alpha Architecture Reference Manual* for additional information about the Alpha Architecture.

1.8 Relevant Header Files

Object and archive file structure declarations and value definitions are contained in the following header files in the `/usr/include` directory:

```
aouthdr.h
```

```
ar.h
coff_type.h
coff_dyn.h
cmplrs/cmrlc.h
cmplrs/stsupport.h
filehdr.h
linenum.h
pdsc.h
reloc.h
scnhdr.h
sym.h
symconst.h
scncomment.h
stamp.h
```

To access object file structures, it is preferable to use defined APIs. APIs provide a constant interface to an underlying structure which will evolve over time. See the `libst_intro`(3) manpage for reference.

2

Headers

Headers serve as a cover page and table of contents for the object file. They contain size descriptions, magic numbers, and pointers to other sections.

The object file components covered in this chapter are the file header, `a.out` header, and section headers:

- The file header identifies the object file and indicates its type.
- The `a.out` header provides the size, location, and addresses of the object's segments.
- Section headers store the name, size, and mapped address of their sections and contain the locations of the section's raw data and relocation entries. Each object file section that is not part of the symbol table has a section header.

An object file may contain other header sections that are used to navigate the symbol table and dynamic loading information. The symbolic header and dynamic header are discussed in Chapter 5 and Chapter 6 respectively.

2.1 New or Changed Header Features

Tru64 UNIX V5.1 includes the following new or modified features:

- A new section header definition that uses reserved bits for specifying section alignment (see Section 2.2.3).

2.2 Structures, Fields, and Values for Headers

2.2.1 File Header (`filehdr.h`)

```
struct filehdr {
        coff_ushort     f_magic;
        coff_ushort     f_nscns;
        coff_int        f_timdat;
        coff_off        f_symptr;
        coff_int        f_nsyms;
        coff_ushort     f_opthdr;
        coff_ushort     f_flags;
};
```

SIZE - 24 bytes, ALIGNMENT - 8 bytes

File Header Fields

`f_magic` File magic number (see Table 2–1). Used for identification.

`f_nscns` Number of section headers in the object file.

`f_timdat` Time and date stamp. This field is implemented as a signed 32-bit quantity that acts as a forward or backward offset in seconds from midnight on January 1, 1970. The resulting date range is approximately 1902-2038.

`f_symptr` File offset to symbolic header. This field is set to zero in a stripped object.

f_nsyms	Size of symbolic header (in bytes). This field is set to zero in a stripped object.
f_opthdr	Size of a.out header (in bytes).
f_flags	Flags (see Table 2–2) that describe the object file. Note that the file header flags cannot be treated as a bit vector because some values are overloaded.

Table 2–1: File Header Magic Numbers

Symbol	Value	Description
ALPHAMAGIC	0603	Object file.
ALPHAMAGICZ	0610	Compressed object file.
ALPHAUMAGIC	0617	**(- V4.0x)** Ucode object file.

Table 2–2: File Header Flags

Symbol	Value	Description
F_RELFLG	0x0001	File does not contain relocation information. This flag applies to actual relocations only, not compact relocations.
F_EXEC	0x0002	File is executable (has no unresolved external references).
F_LNNO	0x0004	Line numbers are stripped from file.
F_LSYMS	0x0008	Local symbols are stripped from file.
F_NO_SHARED	0x0010	Currently unused.
F_NO_CALL_SHARED	0x0020	Object file cannot be used to create a -call_shared (dynamic) executable file.
F_LOMAP	0x0040	Allows a static executable file to be loaded at an address less than VM_MIN_ADDRESS (0x10000). This flag cannot be used by dynamic executables.
F_SHARABLE	0x2000	Shared library.
F_CALL_SHARED	0x3000	Dynamic executable file.
F_NO_REORG	0x4000	Tells object consumer not to reorder sections.
F_NO_REMOVE	0x8000	Tells object consumer not to remove NOP instructions.

2.2.2 a.out Header (aouthdr.h)

The a.out header is also referred to as the "optional header". Note that "optional" is a misnomer because the header is actually mandatory.

```
typedef struct aouthdr {
        coff_ushort     magic;
        coff_ushort     vstamp;
        coff_ushort     bldrev;
        coff_ushort     padcell;
        coff_long       tsize;
        coff_long       dsize;
        coff_long       size;
        coff_addr       entry;
        coff_addr       text_start;
        coff_addr       data_start;
        coff_addr       bss_start;
        coff_uint       gprmask;
        coff_word       fprmask;
        coff_long       gp_value;
} AOUTHDR;
```

SIZE - 80 bytes, ALIGNMENT - 8 bytes

`a.out` Header Fields

`magic` Object-file magic numbers (see Table 2–4).

`vstamp` Object file version stamp. This value consists of a major version number and a minor version number, as defined in the `stamp.h` header file:

Symbol	Value	Description
`MAJ_OBJ_STAMP`	3	Current major object format version
`MIN_OBJ_STAMP`	13	Current minor object format version

This version stamp covers all parts of the object file exclusive of the symbol table, which is covered by an independent version stamp stored in the symbolic header

See Section 1.4.5 for a description of object file versioning.

`bldrev` Revision of system build tools. This value is defined in `stamp.h` and is updated for each major release of the operating system. The values for Tru64 UNIX releases to date are shown below. This field is not meaningful to users.

Table 2–3: Build Revision Constants

Release	bldrev
V1.2	—
V1.3	2
V2.0	4
V3.0	6
V3.2	8
V4.0	10
V5.0	12

`tsize` Text segment size (in bytes) padded to 16-byte boundary; set to zero if there is no text segment.

For `ZMAGIC` object files, this value includes the size of the header sections (file header, `a.out` header, and all section headers). See Section 2.3.2 for more information.

`dsize` Data segment size (in bytes) padded to 16-byte boundary; set to zero if there is no data segment..

`bsize` Bss segment size (in bytes) padded to 16-byte boundary; set to zero if there is no bss segment.

`entry` Virtual address of program entry point. This field is meaningful primarily for executable objects. For shared libraries, it contains the starting address of the first procedure. For pre-link objects, it is typically set to zero.

text_start, data_start, bss_start	Base address of text, data, and bss segments, respectively, for this file. Alignment requirements are discussed in Section 2.3.2.
gprmask	Unused.
fprmask	Unused.
gp_value	The initial GP (Global Pointer) value used for this object. The kernel loads this value into the GP register ($gp) when a program is executed. The program entry point identified by the entry field will load its GP value into the GP register, which may or may not be different than the value in this field for objects with multiple GP ranges. See Section 2.3.4. This value is also used by the linker as a basis for relocation adjustments in objects. See Section 4.3.3.2.

Table 2–4: a.out Header Magic Numbers

Symbol	Value	Description
OMAGIC	0x107	Impure format. The text segment is not write-protected or shareable; the data segment is contiguous with the text segment. An OMAGIC file can be a relocatable object or an executable.
NMAGIC	0x108	Shared text format. NMAGIC files are static executables. This layout is rarely used but supported for historical reasons.
ZMAGIC	0x10b	Demand-paged format. The text and data segments are separated and the text segment is write-protected and shareable. The object can be a dynamic or static executable, or a shared library. All shared objects use a ZMAGIC layout.

2.2.3 Section Headers (scnhdr.h)

Version Note

The following structure definition is for Tru64 UNIX **V5.1** and greater. It is compatible with object format **V3.13** and greater. New fields are identified in the field descriptions following the structure.

```
struct scnhdr {
        char            s_name[8];
        coff_addr       s_paddr;
        coff_addr       s_vaddr;
        coff_long       s_size;
        coff_off        s_scnptr;
        coff_off        s_relptr;
        coff_ulong      s_lnnoptr;
        union {
            struct {
                coff_ushort  _s_nreloc;
                coff_ushort  _s_nlnno;
            } _s;
            struct {
                coff_uint    _s_nreloc:16;
                coff_uint    _s_alignment:4;
                coff_uint    _s_reserved:12;
            } _b;
        }               s_u;
        coff_uint       s_flags;
};
#define s_nreloc     s_u._s._s_nreloc
#define s_nlnno      s_u._s._s_nlnno
#define s_alignment  s_u._b._s_alignment
#define s_reserved   s_u._b._s_reserved
```

SIZE - 64 bytes, ALIGNMENT - 8 bytes

Section Header Fields

`s_name` — Section name (see Table 2–5); null-terminated unless exactly 8 bytes. Long section names are truncated to 8 bytes and are not null-terminated. Unused bytes are zero filled.

`s_paddr` — Base virtual address of section in the image. Although this field contains the same value as `s_vaddr`, normally `s_vaddr` is used and `s_paddr` is ignored.

`s_vaddr` — Base virtual address of a loadable section in the image.

This field is set to zero for nonloadable sections such as `.comment`.

For the sections `.tlsdata` and `.tlsbss`, this field contains an offset from the beginning of the object's dynamically allocated TLS region.

`s_size` — Section size rounded to 16-byte multiple.

`s_scnptr` — File offset to beginning of raw data for the section. The raw data pointed to by this field, and described by the `s_size` field, is mapped at `s_vaddr` (if non-zero) in the process image.

For sections with no raw data, such as `.bss`, this field is set to zero.

`s_relptr` — File offset to relocations for the section; set to zero if the section has no relocations.

`s_lnnoptr` — In `.lita` section header, indicates number of GP ranges used for the object:

Value	Meaning
0	Object has one GP range.
1	Invalid value.
2 or higher	Object has this number of GP ranges.

For sections with GP relative relocations, this field contains the number of `R_GPVALUE` relocation entries for that section. In `.pdata` this field contains the number of run-time procedure descriptors.

For other sections, the field is reserved and must be zero.

Version Note

For object formats less than **V3.13** the value of this field may not be zero and should be ignored.

`s_nreloc` — Number of relocation entries; `0xffff` if number of entries overflows size of this field (see Table 2–6).

s_nlnno — Not used. This field overlays the s_alignment and s_reserved fields.

s_alignment — **(V5.1 -)** Contains a power-of-two biased alignment factor. The alignment is calculated by adding 3 to this value and interpreting the sum as a power of two. The value 0 is interpreted as 16 byte alignment because this is the minimum section rounding allowed. The maximum value that can be represented is 15 which is 256k byte alignment.

Version Note

For object formats less than **V3.13** the value of this field may not be zero and should be ignored.

s_reserved — **(V5.1 -)** Reserved. Must be zero.

Version Note

For object formats less than **V3.13** the value of this field may not be zero and should be ignored.

s_flags — Flags identifying the section (see Table 2–6). Not all of these flag values are single bit masks. See Section 2.3.6 for information on testing section flags.

Table 2–5: Section Header Constants for Section Names

Symbol	Field Contents	Description
_TEXT	.text	Text section
_INIT	.init	Initialization text section
_FINI	.fini	Termination (clean-up) text section
_RCONST	.rconst	Read-only constant section
_RDATA	.rdata	Read-only data section
_DATA	.data	Large data section
_LITA	.lita	Literal address pool section
_LIT8	.lit8	8-byte literal pool section
_LIT4	.lit4	4-byte literal pool section
_SDATA	.sdata	Small data section
_BSS	.bss	Large bss section
_SBSS	.sbss	Small bss section
_UCODE	.ucode	**(obsolete)** Ucode section
_GOT[1]	.got	Global offset table
_DYNAMIC[1]	.dynamic	Dynamic linking information
_DYNSYM[1]	.dynsym	Dynamic linking symbol table
_REL_DYN[1]	.rel.dyn	Relocation information
_DYNSTR[1]	.dynstr	Dynamic linking strings
_HASH[1]	.hash	Dynamic symbol hash table

Table 2–5: Section Header Constants for Section Names (cont.)

Symbol	Field Contents	Description
_MSYM[1]	.msym	Additional dynamic linking symbol table
_LIBLIST[1]	.liblist	Shared library dependency list
_CONFLICT[1]	.conflict	Additional dynamic linking information. (This name is truncated to .conflic when stored in the s_name field of the section header.)
_XDATA[2]	.xdata	Runtime procedure descriptors and GP range information
_PDATA[2]	.pdata	Code range descriptors
_TLS_DATA	.tlsdata	Initialized TLS data
_TLS_BSS	.tlsbss	Uninitialized TLS data
_TLS_INIT	.tlsinit	Initialization for TLS data
_COMMENT	.comment	Comment section

Table Notes:

1. These sections exist only in dynamic executables and shared libraries and are used during dynamic linking. See Chapter 6 for details.
2. The .xdata and .pdata sections contain exception-handling data. See the *Calling Standard for Alpha Systems* for details. Other sections are described in Chapter 3.

Table 2–6: Section Flags (s_flags field)

Symbol	Value	Description
STYP_REG	0x00000000	Regular section: allocated, relocated, loaded. User section flags have this setting.
STYP_TEXT	0x00000020	Text only
STYP_DATA	0x00000040	Data only
STYP_BSS	0x00000080	Bss only
STYP_RDATA	0x00000100	Read-only data only
STYP_SDATA	0x00000200	Small data only
STYP_SBSS	0x00000400	Small bss only
STYP_UCODE	0x00000800	**(obsolete)** Ucode
STYP_GOT[1]	0x00001000	Global offset table
STYP_DYNAMIC[1]	0x00002000	Dynamic linking information
STYP_DYNSYM[1]	0x00004000	Dynamic linking symbol table
STYP_REL_DYN[1]	0x00008000	Dynamic relocation information
STYP_DYNSTR[1]	0x00010000	Dynamic linking symbol table
STYP_HASH[1]	0x00020000	Dynamic symbol hash table
STYP_DSOLIST[1]	0x00040000	Shared library dependency list
STYP_MSYM[1]	0x00080000	Additional dynamic linking symbol table
STYP_CONFLICT[1]	0x00100000	Additional dynamic linking information
STYP_FINI	0x01000000	Termination text only
STYP_COMMENT	0x02000000	Comment section

Table 2–6: Section Flags (`s_flags` field) (cont.)

Symbol	Value	Description
STYP_RCONST	0x02200000	Read-only constants
STYP_XDATA	0x02400000	Runtime procedure descriptors and GP range information
STYP_TLSDATA	0x02500000	Initialized TLS data
STYP_TLSBSS	0x02600000	Uninitialized TLS data
STYP_TLSINIT	0x02700000	Initialization for TLS data
STYP_PDATA	0x02800000	Code range descriptors
STYP_RESTEXT	0x02900000	**(not supported)** Resident text
STYP_LITA	0x04000000	Address literals only
STYP_LIT8	0x08000000	8-byte literals only
STYP_EXTMASK	0x0ff00000	Identifies bits used for multiple bit flag values.
STYP_LIT4	0x10000000	4-byte literals only
S_NRELOC_OVFL[2]	0x20000000	Indicates that section header field s_nreloc overflowed
STYP_INIT	0x80000000	Initialization text only

Table Notes:

1. These sections exist only in dynamic executables and shared libraries and are used during dynamic linking. See Chapter 6 for details.
2. The S_NRELOC_OVFL flag is used when the number of relocation entries in a section overflows the s_nreloc field in the section header. In this case, s_nreloc contains the value 0xffff and the s_flags field has the S_NRELOC_OVFL flag set. The actual relocation count is in the first relocation entry for the section.

Version Note

The value STYP_RESTEXT is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX.

General Notes:

The system linker uses the s_flags field instead of s_name to determine the section type. User-defined sections (see Section 3.3.10) constitute an exception; they are identified exclusively by section name.

Each section header must be unique within the object file. For system-defined sections, both the section name and flags must be unique. For user-defined sections, the name must be unique.

2.3 Header Usage

2.3.1 Object Recognition

Object file consumers use the file header to recognize an input file as an object file. Other tools that do not support objects may use the file header to determine that they cannot process the file. The file command can also identify an object by means of the file and a.out headers.

A file is identified as an object in its first 16 bits. These bits correspond to the magic number field in the file header. Objects built for the Alpha architecture are identified by the magic number ALPHAMAGIC; equivalent compressed objects are identified by ALPHAMAGICZ. Foreign objects, which are objects built for other architectures, may also be positively identified. However, once a foreign object is recognized, it is not considered to be a linkable or executable object file on the Alpha system.

In addition to providing basic identification, the file header also provides a high-level description of the object file through its flags field. File header flags store the following information: whether the object is executable, whether symbol table sections have been stripped, whether the file is suitable for creation of a shared library, and more. See Table 2–2 for a list of all flags.

The a.out header magic numbers also contribute important information about the file format. The magic numbers signify different organizations of object file sections and indicate where the image will be mapped into memory (see Section 2.3.2).

2.3.2 Image Layout

The a.out header stores run-time information about the object. Its magic number field indicates how the file is to be organized in virtual memory. Note that the contents and ordering of the sections of the image can be affected by compilation options and program contents in addition to the MAGIC classification.

The possible image formats are:

Impure Format (OMAGIC)	OMAGIC files are typically relocatable object files. They are referred to as "impure" because the text segment is writable.
Shared Text Format (NMAGIC)	NMAGIC files are static executables that use a different organization from the default ZMAGIC layout. The NMAGIC format is historical and offers no special advantages. This format can be selected by using the linker option -n or -nN in conjunction with -non_shared. In an NMAGIC file, the text segment is shared.
Demand Paged Format (ZMAGIC)	ZMAGIC files are executable files or shared libraries. This format is referred to as demand-paged because its segments are blocked on page boundaries, allowing the operating system to page in text and data as needed by the running process. By default, the linker aligns ZMAGIC segments on 64K boundaries, the maximum possible page size on Alpha systems.

The ordering of sections within segments is flexible. Diagrams in this section depict the default ordering as laid out by the linker.

The default segment ordering, which places the text segment before the data segment, is flexible. However, the bss segment is required to contiguously follow the data segment, wherever the data segment is located.

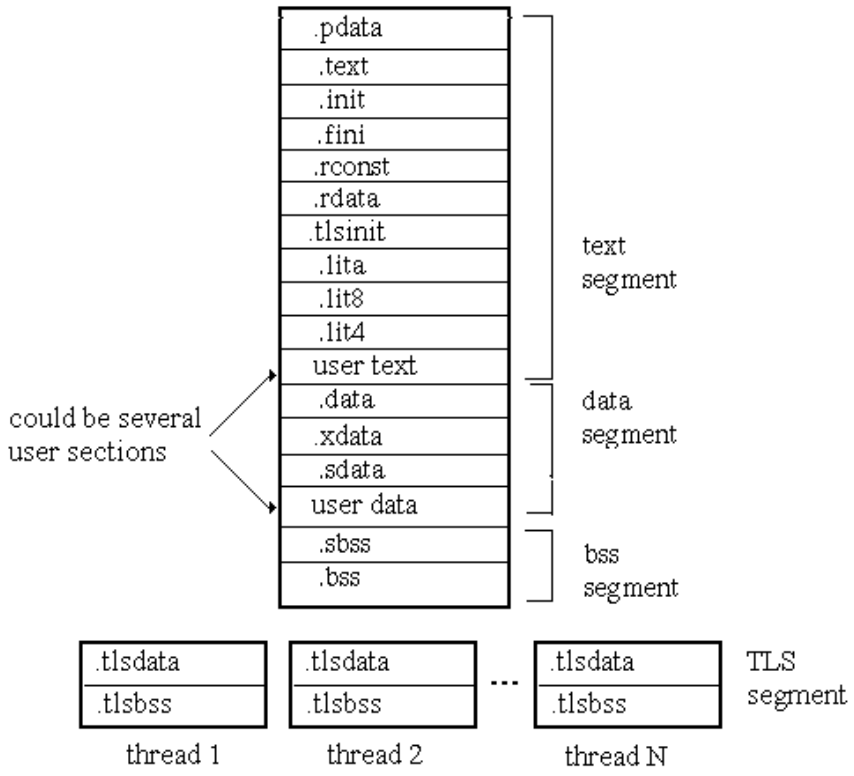
All three formats are constrained by the following restrictions:

- Segments must not overlap.
- The bss segment must follow the data segment.
- All text addresses in the object file must be within two gigabytes (0x7fff8000) of all data addresses in the file.

2.3.2.1 OMAGIC

The OMAGIC format typically has the following layout and characteristics:

Figure 2–1: OMAGIC Layout



- Segments must not overlap.
- The bss segment must follow the data segment.
- All text addresses in the object file must be within two gigabytes (0x7fff8000) of all data addresses in the file.
- Starting section addresses are aligned on a 16-byte boundary.
- Pre-link OMAGIC objects are zero-based, with the data segment contiguous to the text segment. The default text segment address for partially linked objects is 0x10000000, and the data segment follows the text segment.
- Usually contain relocation information.
- Cannot be a shared object.

Starting addresses can be specified for the text and data segments using -T and -D options. These addresses can be anywhere in the virtual address space but must be aligned on a 16-byte boundary.

OMAGIC layout is most commonly used for pre-link object files produced by compilers. Post-link OMAGIC files tend to be used for special purposes such as loadable device drivers and om input objects.

Loadable device drivers must be built as OMAGIC files because the kernel loader kloadsrv relies upon relocation information in order to link objects into the kernel image.

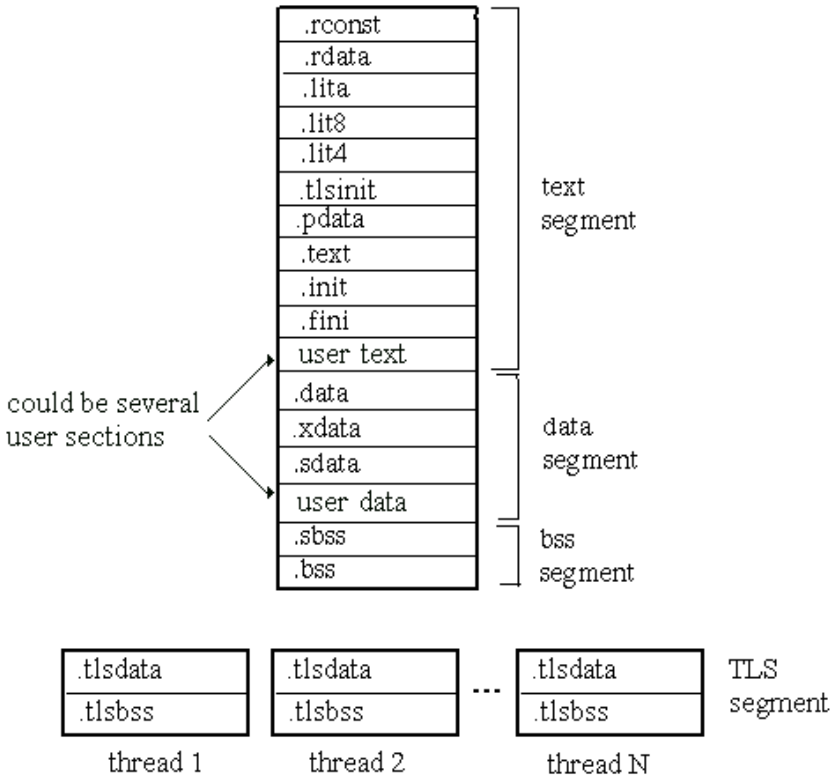
OMAGIC files can also be executable. An important example of an OMAGIC executable file is the kernel, /vmunix. A programmer might also choose to use an OMAGIC format for self-modifying programs or for any other application that has a reason to write to the text segment.

2.3.2.2 NMAGIC

The NMAGIC file format is of historical interest only.

The NMAGIC format typically has the following layout and characteristics:

Figure 2–2: NMAGIC Layout



- Segments must not overlap.
- The bss segment must follow the data segment.
- All text addresses in the object file must be within two gigabytes (0x7fff8000) of all data addresses in the file.
- Text and data segment addresses fall on page-size boundaries. The bss segment is aligned on a 16-byte boundary.
- By default, the starting address of the text segment is 0x20000000 and the starting address of the data segment is 0x40000000.
- Cannot be a relocatable object, partially linked object, or a shared object.

Addresses can be specified for the start of the text and data segments using -T and -D options. These addresses may be anywhere in the virtual address space but must be a multiple of the page size.

2.3.2.3 ZMAGIC

The ZMAGIC format typically has the following layout and characteristics:

Figure 2–3: ZMAGIC Layout for Shared Object

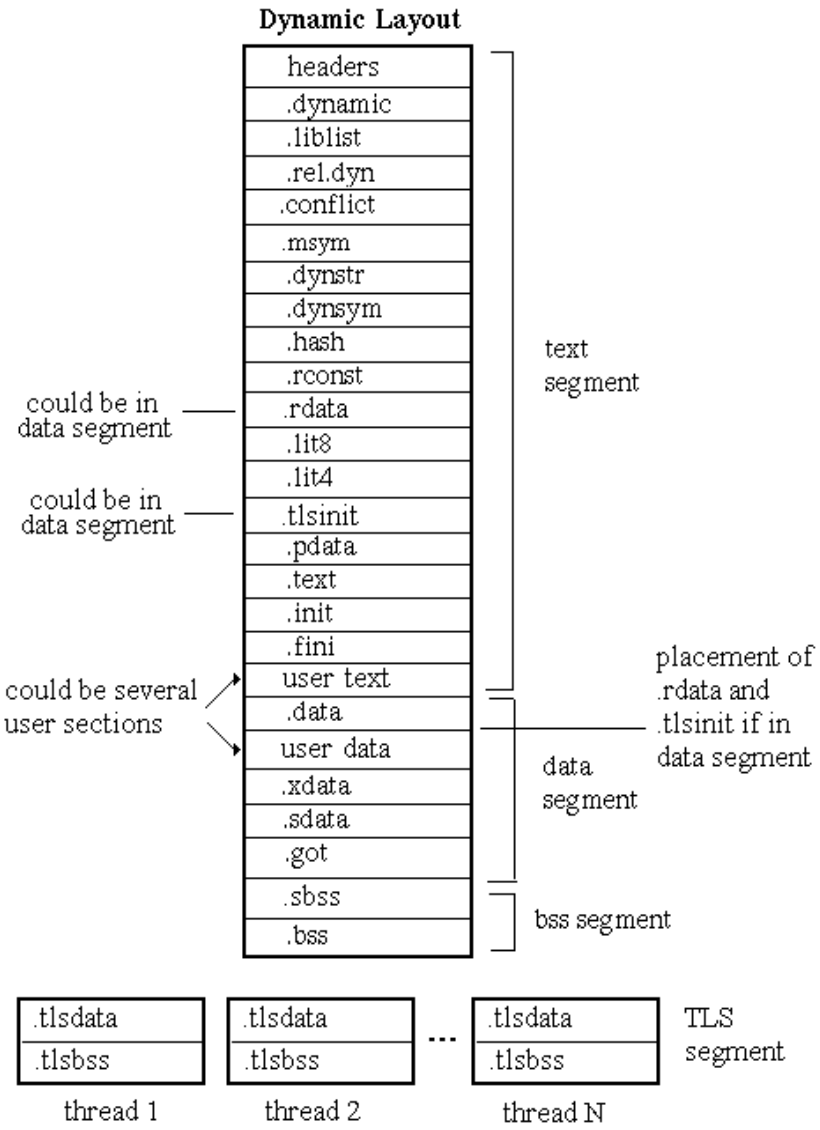
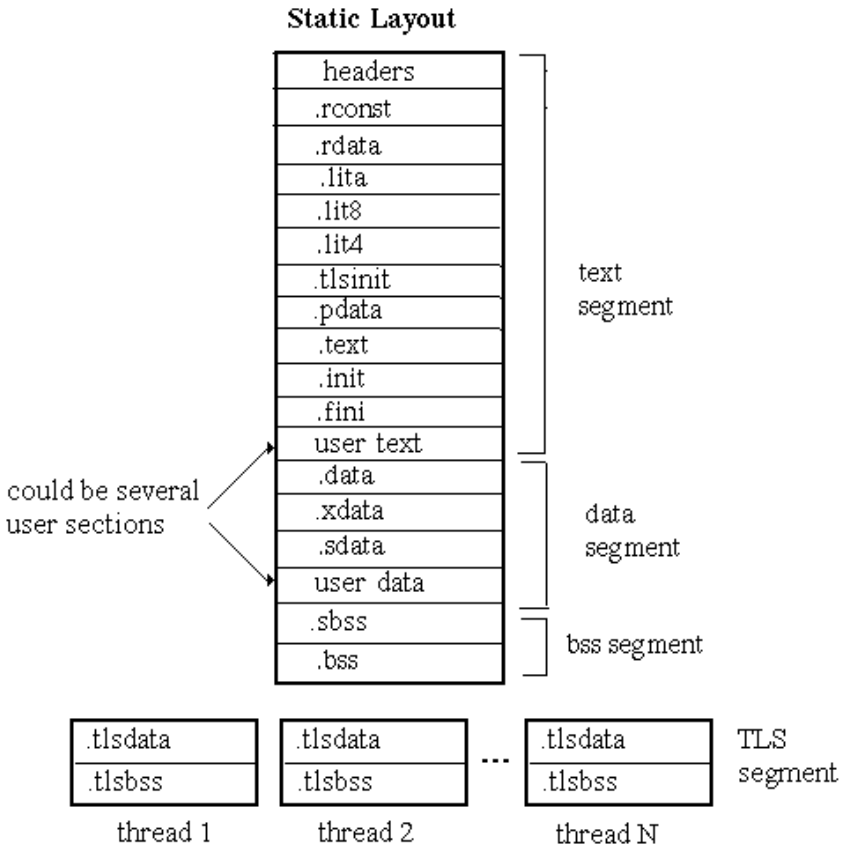


Figure 2–4: `ZMAGIC` Layout for Static Executable Objects



The `.rdata` and `.tlsinit` sections are shown as part of the text segment. However, it is possible that one or both of those sections might be in the data segment. They are placed in the data segment only if they contain dynamic relocations.

- Segments must not overlap.
- The bss segment must follow the data segment.
- All text addresses in the object file must be within two gigabytes (`0x7fff8000`) of all data addresses in the file.
- Text and data segments are blocked; the blocking factor is the page size.
- By default the starting address of the text segment is `0x120000000` and the starting address of the data segment is `0x140000000`. The bss segment follows the data segment.
- Can be either a shared or static object, but not a relocatable object.

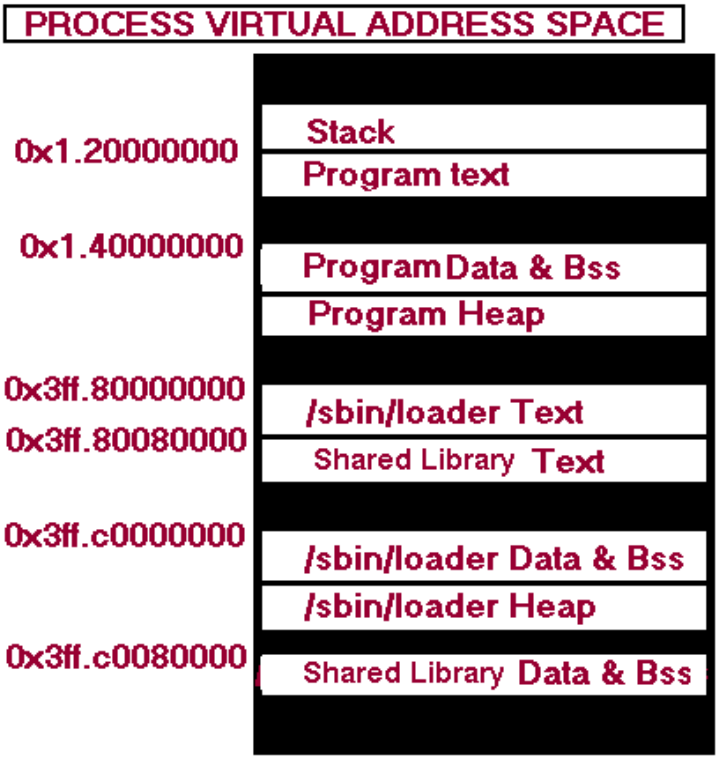
Addresses can be specified for the start of the text and data segments using `-T` and `-D` options. Those addresses can be anywhere in the virtual address space but must be a multiple of the page size.

2.3.3 Address Space

At load time, an executable object is mapped into the system's virtual memory using one of the formats detailed in Section 2.3.2. The user can choose where the object, transformed into the program image, will be loaded, but system-specific constraints exist. This section discusses the general layout of the address space and the various considerations involved in choosing memory locations for object file segments.

Figure 2–5 shows the default memory scheme for a dynamic image.

Figure 2–5: Address Space Layout



The stack is used for storing local variables. It grows toward zero. The stack pointer (stored in register `$sp`) points to the top of the stack at all times. In generated code, items on the stack are often referenced relative to the stack pointer.

The program heap is reserved for system memory-allocation calls (`brk()` and `sbrk()`). TLS sections are allocated from the heap. The heap begins where the bss segment of the program ends, and the special symbol `_end` indicates the start of the heap. The heap's placement can also be calculated using the starting addresses and sizes of segments in the `a.out` header. The mapping of shared libraries may impose an upper bound on the heap's size. Some programs do not have a heap.

The dynamic loader and shared libraries reside in memory during program execution. See Section 6.3.2 for details.

User programs can request additional memory space that is dynamically allocated. One way to request space is through an anonymous `mmap()` call. This system call creates a new memory region belonging to the process. The user can attempt to specify the address where the region will be placed. However, if it is not possible to accommodate that placement, the system will rely on environment variables to dictate placement. See the `mmap`(2) man page for details.

The usable address range for user mode addresses is `0x0 - 0x40000000000`. Attempts to map object file segments outside this range will fail, and the defaults will be invoked or execution aborted.

2.3.3.1 Address Selection

Several mechanisms permit the user to select addresses for loadable objects or assist the user in choosing viable addresses. Unless there is a good reason to do otherwise, it is preferable to rely on system defaults, which are designed to enhance performance and reduce conflicts.

The linker's -T and -D options may be used to specify the starting addresses for the text and data segments of an executable, respectively. Use of these options may be appropriate for large applications with dependencies on many shared libraries that need to explicitly manage their address space. Programs relying in any way on fixed addresses may also need to control the segment placement.

Another use of the address selection options is to place an application in the lowest 31 bits of the address space. To restrict an application to this part of the address space, the -T and -D switches may be used in conjunction with the -taso option (see Section 2.3.3.2) or separately.

The default placement of the text and data segments at 0x120000000 and 0x140000000 for executables means the default maximum size of the text segment is 0x20000000 bytes, or approximately 500MB. If this space is insufficient, the -D option can be used to enlarge it by specifying a higher starting address for the data segment.

The -T and -D options can also be used to change the segment ordering. Some applications, such as those ported from other platforms onto the Alpha platform, may rely upon the data segment being mapped in lower addresses than the text segment.

If only -T or only -D is specified on the link line, system defaults are used for the nonspecified address. If a given address is not properly aligned, the linker rounds the value to the applicable boundary. If inappropriate addresses are chosen, such as addresses for the text and data segments that are too far apart, linking may fail. Alternatively, linking may succeed, but execution can abnormally terminate if addresses are incompatible with the system memory configuration.

The linker option -B, which specifies a placement for the bss segment, is available for partial links only. For executable objects, the bss segment should be contiguous with the data segment, which is the system default. As a general rule, the -B option should not be used.

Another mechanism permits address selection for shared libraries. A registry file, by default named so_locations, stores shared library segment addresses and sizes. The so_locations directives, described in the *Programmer's Guide*, can be used to control the linker's address selection for shared libraries.

2.3.3.2 TASO Address Space

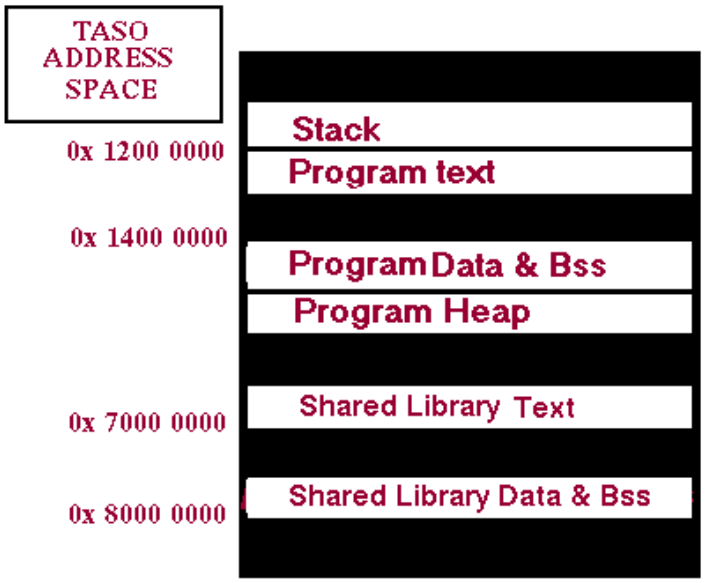
The TASO (Truncated Address Space Option) address space is a 32-bit address-space emulation that is useful for porting 32-bit applications to 64-bit Alpha systems. Selection of the -taso linker option causes object file segments to be loaded into the lower 31 bits of the memory space. This can also be accomplished, in part, by using -T and -D. If the -taso option is used in conjunction with the -T or -D options, the addresses specified with -T and -D take precedence.

Use of the -taso option also causes shared libraries linked outside the 31-bit address space to be appropriately relocated by the loader. All executable objects and shared libraries will be mapped to the address range 0x0 - 0x7fffffff.

The default segment addresses for a TASO executable are 0x12000000 for the text segment and 0x14000000 for the data segment, with the bss segment directly following the data segment. The -T and -D options can be used to alter the segment placement if necessary.

Figure 2–6 is a diagram of the TASO address space layout.

Figure 2–6: TASO Address Space Layout



A TASO shared object is marked as such with the `RHF_USE_31BIT_ADDRESSES` flag in the `DT_FLAGS` entry in the dynamic header. The loader recognizes dynamic executable objects marked with the TASO flag and maps their shared library dependencies to the TASO address space. A TASO static executable is not explicitly identified.

2.3.4 GP (Global Pointer) Ranges

Programs running on Tru64 UNIX obtain the addresses of procedures and global data by means of a GP (Global Pointer) and an address table. Address ranges and address-table sections (`.lita` and `.got`) are described further in Section 3.3.2 and Section 6.3.3. However, several important pieces of information concerning GP-relative addressing are contained in the headers.

During program execution, the global pointer register (`$gp`) contains the active GP value. This value is used to access run-time addresses stored in the image's address-table section. Addresses are specified in generated code as an offset to the GP.

There are several reasons for using this GP-relative addressing technique:

- Alpha instructions support only 16-bit relative addressing, but the generated code must be able to quickly and efficiently access arbitrary 64-bit addresses.
- The generated code must be position independent.
- The addressing method must support symbol preemption (see Section 6.3.4).

A GP range is the set of addresses reachable from a given GP. The size of this range is approximately 64KB, or 8K 64-bit addresses.

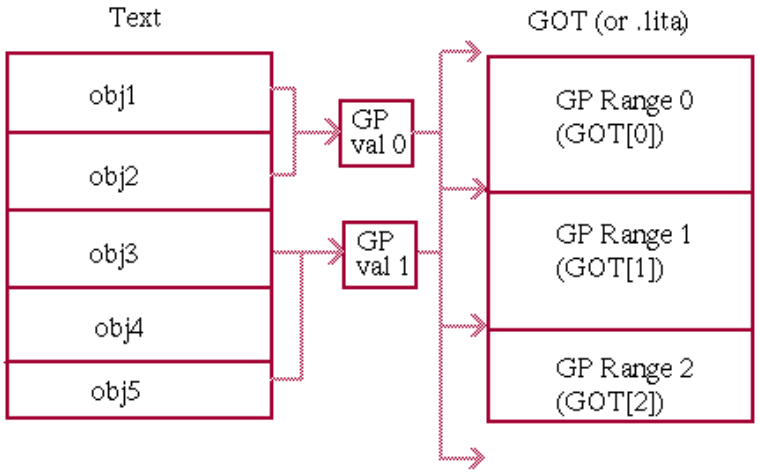
Although only one GP value is active at any time, a program can use several GP values. A program's text can be divided into ranges of addresses with a different GP value for each range. The linker will start a new GP range at a boundary

between two input object file's section contributions. As a result, a GP range will rarely be filled before a new GP range is started. Regardless of how much of a GP range is actually used, the linker always sets the GP value associated with that range as follows:

```
GP value = GP range start address + 32752
```

Figure 2–7 is a depiction of the use of GP values and ranges.

Figure 2–7: GP (Global Pointer) Ranges



Objects can share a GP range, as shown in Figure 2–7, or use more than one GP range, depending on the amount of program data. However, the *Calling Standard for Alpha Systems* specifies that a single procedure can use only one GP value. The `a.out` header's `gp_value` field contains either the GP value of the object (if there is only one) or the first one the program should use (if there are multiple GP ranges).

How the number of GP ranges is represented in an object depends on the object's type:

- For objects with a `.lita` section, the section header field `s_nlnnoptr` indicates the number of GP ranges, as explained in Section 2.2.3.
- In a relocatable object (`OMAGIC` file), a new GP range is signaled by a `R_GPVALUE` relocation entry. See Section 4.3.4.18 for details.
- In shared objects, multiple GP ranges are indicated by entries in the dynamic header section (`.dynamic`), which are described in Section 6.2.1.

2.3.5 Alignment

Alignment is an architectural issue that must be dealt with in the object file at several levels: object file segments, object file sections, and program variables all have alignment requirements.

Data alignment refers to the rounding that must be applied to a data item's address. For natural alignment, a data item's address must be a multiple of its size. For example, the natural alignment of a character variable is one byte, and the natural alignment of a double-precision floating-point variable is 8 bytes.

On Alpha systems, all data should be aligned on proper boundaries. Unaligned references can result in substantially slower access times or cause fatal errors. The compiler and the user have some control over the alignments through the use of assembler directives and compilation flags (see the *Programmer's Guide* and *Assembly Language Programmer's Guide*). When designing alignment attributes, however, the architectural cost of loading unaligned values should be considered.

Object file segments are, by default, aligned as indicated in Section 2.3.2. Segment alignment can be impacted by section alignment. The segment alignment must be evenly divisible by the highest alignment factor for sections contained in that segment.

For shared libraries that are not mapped at their quickstart addresses the loader will map segments with a minimum alignment of 8K bytes. If any section in the shared library requires an alignment greater than 8K bytes, the loader will map the text segment with 64K byte alignment. The linker is responsible for assigning segment addresses with a distance that is a 64K byte multiple. This will allow the loader to align the data segment address which is mapped at a fixed distance from the text segment.

Object file sections may have a power-of-two alignment factor specified in their section headers (see Section 2.2.3). The default section alignment is 16 bytes.

Version Note

Power-of-two section alignment is supported in object format **V3.13** and greater for Tru64 UNIX **V5.1** and greater.

The default alignment boundary for raw data is 16 bytes. Smaller alignments can be applied to individual data items allocated in raw section data. If a data item must be aligned with greater than 16 byte alignment, the section in which it is allocated must be aligned with a power-of-two alignment factor that is greater than or equal to the data item's required alignment.

Individual data items should meet the following minimum requirements. Structure members and array elements are aligned according to the minimum requirements in order to minimize pad bytes between members. Other data items are typically aligned with 8 or 16 byte rounding due to alignment requirements imposed by the generated code used to access data addresses.

- Atomic data items are aligned using natural alignment.
- Structures are aligned based on the size of their largest member.
- Arrays are aligned according to the alignment requirements of the array element.
- Procedures are aligned on a 16-byte (quadruple instruction word) boundary. This preserves the integrity of multiple-instruction issue established by the instruction scheduling phase of code generation.
- Common storage class symbols must be aligned when they are allocated. The `value` field for a common storage class symbol indicates its size and determines which section it will be allocated in (`.bss` or `.sbss`). The `alignment` field for the common storage class symbol indicates the required power-of-two alignment biased by 2^3. If `alignment` is zero, the default alignment is based on the symbol's size. Common storage class symbols with a size of 16-bytes or greater are aligned to octaword (16-byte) boundaries, otherwise they are aligned to quadword (8-byte) boundaries. The maximum alignment supported for allocating common storage class symbols is 64K bytes. This is represented in the `alignment` field as the value "13".

Version Note

The definition of a power-of-two alignment field in external symbol table entries is supported in Tru64 UNIX **V5.1** and greater. Objects built by compilers that do no support the alignment field will appear to have the alignment set to 0 which will yield the desired default behavior.

Sections are padded wherever necessary to maintain proper alignment. Padding is done with zero bytes in the data and bss sections. In the text segment, each routine is padded with NOP instructions to a 16-byte boundary. The section sizes reported in the section headers and the segment sizes reported in the a.out header reflect this padding.

2.3.6 Section Types

The primary unit of an object file is a section, and the sections in an object are identified, located, and broadly characterized by means of the section headers. Object files are organized into sections primarily to enable the linker to combine multiple input objects into an executable image. At link time, sections of the same type are concatenated or merged. The sectional breakdown also provides the linker flexibility in segment mapping; the linker has a choice in assigning sections to segments for memory-mapping and loading.

Section headers include flags that describe the section type. These flags identify the section type and attributes. See Table 2–6 for a complete listing of section flags. Note that the s_flags field cannot be treated as a simple bit vector when testing or accessing section types because some of the flag values are overloaded. The algorithm below illustrates how to test for a particular section type using the s_flags field.

```
if (type & STYP_EXTMASK)
    FOUND = ((SHDR.s_flags & STYP_EXTMASK) == type)
else
    FOUND = (SHDR.s_flags & type)
```

Sections can be mapped or unmapped. A mapped section is one that is part of the process image as well as the object file. An unmapped section is present only in the on-disk object file.

Raw data, organized by section and segment, is part of the process image. For a ZMAGIC file, all header sections in the object are also mapped into memory as part of the text segment.

2.3.7 Special Symbols

Some special symbol names are reserved for use by the linker or loader. The majority of these special symbols correspond to locations in the image layout.

Table 2–7 describes the special symbols and indicates whether they are reserved for the linker or the loader. Additional special symbols for debug information are described in Section 5.3.9.

Table 2–7: Special Symbols

Linker Reserved Symbols	
Symbol	**Description**
_BASE_ADDRESS[3]	Base address of text segment.
_cobol_main	First COBOL main symbol; undefined if not a COBOL program.
_DYNAMIC	Starting address of .dynamic section if present; otherwise, zero.
_DYNAMIC_LINK	Enumeration value identifying module type: 0 = static executable, 1 = dynamic executable, 2 = shared library.
_ebss	End of bss segment.
_edata	End of data segment.

Table 2–7: Special Symbols (cont.)

Linker Reserved Symbols	
Symbol	**Description**
edata[1]	Weak symbol for end of data segment.
_end	End of bss segment.
end[1]	Weak symbol for end of bss segment.
_etext	End of text segment.
etext[1]	Weak symbol for end of text segment.
_fbss	First location of bss data. Usually the virtual address of either the .sbss or .bss section.
_fdata	First location of initialized data. Usually the virtual address of the .data section and data segment.
_fpdata	Start of .pdata section.
_fpdata_size	Number of entries in .pdata. The exception-handling object file sections (.pdata and .xdata) are included in the output object if this symbol is referenced.
__fstart	Start of .fini section.
_ftext	First location of executable text. Usually the virtual address of the .text section.
_ftlsinit	The address of the .tlsinit section.
_GOT_OFFSET[3]	Starting address of .got section if present; otherwise, zero.
_gp[3]	GP value stored in a.out header.
_gpinfo	Table of GP ranges used exclusively by exception handling code.
__istart	Start of .init section.
_procedure_string_table[2]	String table for run-time procedures
_procedure_table[2]	Run-time procedure table.
_procedure_table_size[2]	Number of entries in run-time procedure table.
__tlsbsize	Size of the .tlsbss section.
__tlsdsize	Size of the .tlsdata section.
__tlskey	The value of this symbol is the address of the GOT or .lita entry of the tlsoffset symbol.
__tlsoffset	Offset in the TSD array of the TLS pointer for a particular object. For static executables, this value is set at link time. For shared objects, the value is set to 0 at link time and filled in at run time.
__tlsregions	The number of TLS regions (TSD entries) that are used by an executable or library.
Loader Reserved Symbols	
_ldr_process_context	Points to loader's data structures.
ldr_process_context[1]	Weak symbol pointing to loader's data structures.
_rld_new_interface	The generic loader entry point servicing all loader function calls.

Table Notes:

1. These symbols are not defined under strict ANSI standards. They are weak symbols that are retained for backward compatibility. See Section 6.3.4.2 for further discussion of weak aliasing to strong symbols.
2. These symbols relate to the run-time procedure table, which is a table of `RPDR` structures (their declaration is in the header file `sym.h`). The table is a subset of the procedure descriptor table portion of the symbol table with one unused field, `exception_info`, that is set to zero. The run-time procedure table is maintained for historical reasons. It is not used by the system's exception handling software, nor any other Tru64 UNIX runtime support.
3. These symbols are recorded as `scAbs` symbols in the external symbol table, but their values are relocatable addresses that are not absolute values in a shared library. This misclassification is maintained partly for historical reasons, and partly because the values of these symbols cannot be described as an offset within a specific section. The equivalent dynamic symbol table entries identify these symbols as text (`SHN_TEXT`) or data symbols (`SHN_DATA`) rather than absolute symbols (`SHN_ABS`).

Version Note

Prior to Tru64 UNIX **V5.1** the system linker records these symbols as absolute symbols (`SHN_ABS`) in the dynamic symbol table, and they are not relocated correctly by the dynamic loader.

The linker defines special symbols only if they are referenced.

The majority of these symbols have local binding in a shared object's dynamic symbol table. Consequently, a shared object can only reference its own definition of these symbols. However, several special symbols have global scope. The linker-defined symbols `end`, `_end`, `__istart`, and `_cobol_main` are global, which implies that each has a unique value process-wide. The symbol `_end` and its weak counterpart `end` are used by `libc.so` to identify the start of the heap in memory. The symbol `_cobol_main` gives a COBOL program's main entry point.

Special symbols in addition to those listed in Table 2–1 are defined by the linker to represent object file section addresses:

```
.bss
.comment
.data
.fini
.init
.lit4
.lit8
.lita
.pdata
.rconst
.rdata
.sbss
.sdata
.text
.xdata
```

The value of the symbol is the starting address of the corresponding section. These symbols generally are not referenced by user code. For shared objects, they may appear in the dynamic symbol table.

2.3.7.1 Accessing

A user program can reference, but not define, reserved symbols. An error message is generated if a user program attempts to define a symbol reserved for system use.

A special symbol is a label, and thus its value is its address. Interpreting a label's contents as its value may lead to an access violation, particularly for those linker-defined symbols that are not address locations within the image (for example, `_DYNAMIC_LINK` or `_procedure_table_size`).

The following example shows how linker-defined labels are referenced in code:

```
$ cat gprange.c
#include <stdio.h>
#include <excpt.h>

extern unsigned long _gpinfo[];
extern unsigned long _ftext;
extern unsigned long _fdata;

main(){
    int i;
    unsigned long tstart, tend;
    unsigned long gpval;

    if (!_gpinfo || _gpinfo[0] != GPINFO_MAGIC) {
        printf("No GP range info\n");

    } else {
        for (i=1; _gpinfo[i] != GPINFO_LAST; i+=3){

            tstart = (unsigned long)&_ftext + _gpinfo[i];
            tend = tstart + _gpinfo[i+1];

            gpval = (unsigned long)&_fdata + _gpinfo[i+2];

            printf("GP=0x%lx for Text Range [0x%lx - 0x%lx]\n",
                   gpval, tstart, tend);
        }
    }
}

$ cc gprange.c
$ a.out
GP=0x1400080c0 for Text Range [0x120000fe0 - 0x120001440]
```

This example prints out the GP ranges recorded in the `.xdata` section. See Section 3.3.8 for a description of the GP range info.

2.4 Language-Specific Header Features

The linker-defined symbol `_cobol_main` is set to the symbol value of the first external symbol encountered by the linker with its `cobol_main` flag set. COBOL programs use this symbol to determine the program entry point.

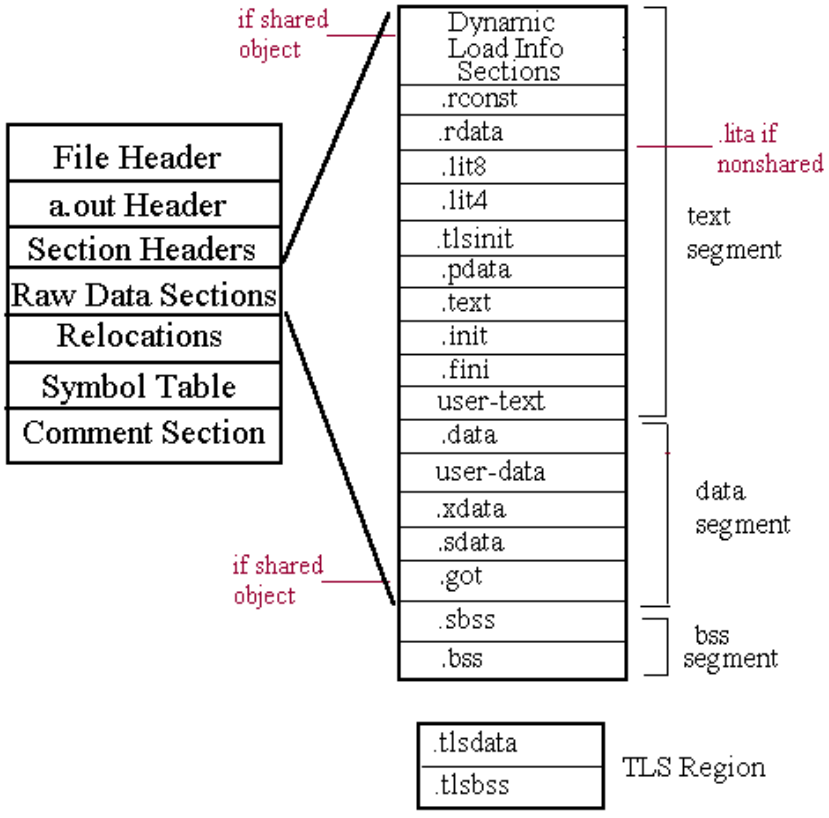
3

Instructions and Data

Instructions and data are the portions of the object file that are logically copied into the final process image. Instructions include all executable machine code. Data includes initialized and zero-initialized data, constant data, exception-handling data structures, and thread local storage (TLS) data. The breakdown of the instructions and data into object file sections is shown in Figure 3–1.

Object file sections are organized into three loadable segments: text, data, and bss. Multiple TLS regions may also be loaded. The mapping of sections into segments is principally determined by segment access permissions and object file. Figure 3–1 illustrates the layout of a typical dynamic executable file. See Section 2.3.2 for details.

Figure 3–1: Raw Data Sections of an Object File



The object file sections containing dynamic load information are covered separately in Chapter 6. Chapter 7 describes the `.comment` section data. This chapter covers all other raw data sections.

3.1 New or Changed Instructions and Data Features

Version 5.1 of Tru64 UNIX adds new fields to the code range descriptor (see Section 3.2.1) and the run-time procedure descriptor (see Section 3.2.2).

Version 5.0 of Tru64 UNIX supports a new name-recognition mechanism for ordering subsystem-generated initialization and termination routines. See Section 3.3.5.2.4 for details.

Version 3.13 of the object file format does not introduce any new features for the instructions or data contained within the object file.

3.2 Structures, Fields, and Values for Instructions and Data

Section 3.2.1 and Section 3.2.2 contain structure declarations for the exception-handling data structures as stored in the `.xdata` and `.pdata` object file sections. These are the only two sections covered in this chapter that contain structured data. Text sections containing machine instructions use the Alpha instruction formats and other sections contain binary and character data.

3.2.1 Code Range Descriptor (pdsc.h)

The `.pdata` section contains a table of code range descriptors ordered by address.

```
typedef unsigned int            pdsc_mask;
typedef unsigned int            pdsc_space;
typedef int                     pdsc_offset;

union pdsc_crd {
        struct {
            pdsc_offset    begin_address;
            pdsc_offset    rpd_offset;
        } words;
        struct {
            pdsc_mask      context_t             :1;  (V5.1 - )
            pdsc_mask      context_s             :1;  (V5.1 - )
            pdsc_offset    shifted_begin_address :30;
            pdsc_mask      no_prolog             :1;
            pdsc_mask      memory_speculation    :1;
            pdsc_offset    shifted_rpd_offset    :30;
        } fields;
}
```

SIZE - 8 bytes, ALIGNMENT - 4 bytes

Version Note

The fields marked "V5.1" in the preceding structure definition are new fields for Tru64 UNIX **V5.1** and greater. The new fields take the place of a reserved field so there is no change in the structure size.

See the *Calling Standard for Alpha Systems* for a full description.

3.2.2 Run-time Procedure Descriptor (pdsc.h)

The `.xdata` section contains run-time procedure descriptors. These descriptors are not necessarily sorted, and may be intermixed with unstructured exception-handling data.

```
typedef unsigned char           pdsc_uchar_offset;
typedef unsigned short          pdsc_ushort_offset;
typedef unsigned int            pdsc_count;
typedef unsigned int            pdsc_register;
typedef unsigned long           pdsc_address;

typedef union pdsc_rpd {

        struct pdsc_short_stack_rpd {
            pdsc_mask          flags:8;
```

```
        pdsc_uchar_offset  rsa_offset;
        pdsc_mask          fmask:8;
        pdsc_mask          imask:8;
        pdsc_count         frame_size:16;
        pdsc_count         sp_set:8;
        pdsc_count         entry_length:8;
    } short_stack_rpd;

    struct pdsc_short_reg_rpd {
        pdsc_mask          flags:8;
        pdsc_space         reserved1:3;
        pdsc_register      entry_ra:5;
        pdsc_register      save_ra:5;
        pdsc_space         reserved2:11;
        pdsc_count         frame_size:16;
        pdsc_count         sp_set:8;
        pdsc_count         entry_length:8;
    } short_reg_rpd;

    struct pdsc_long_stack_rpd {
        pdsc_mask          flags:11;
        pdsc_register      entry_ra:5;
        pdsc_ushort_offset rsa_offset;
        pdsc_count         sp_set:16;
        pdsc_count         entry_length:16;
        pdsc_count         frame_size;
        pdsc_mask          reserved:2;        (V5.1 - )
        pdsc_offset        return_address:30; (V5.1 - )
        pdsc_mask          imask;
        pdsc_mask          fmask;
    } long_stack_rpd;

    struct pdsc_long_reg_rpd {
        pdsc_mask          flags:11;
        pdsc_register      entry_ra:5;
        pdsc_register      save_ra:5;
        pdsc_space         reserved1:11;
        pdsc_count         sp_set:16;
        pdsc_count         entry_length:16;
        pdsc_count         frame_size;
        pdsc_mask          reserved2:2;       (V5.1 - )
        pdsc_offset        return_address:30; (V5.1 - )
        pdsc_mask          imask;
        pdsc_mask          fmask;
    } long_reg_rpd;

    struct pdsc_short_with_handler {
        union {
            struct pdsc_short_stack_rpd short_stack_rpd;
            struct pdsc_short_reg_rpd   short_reg_rpd;
        } stack_or_reg;
        pdsc_address       handler;
        pdsc_address       handler_data;
    } short_with_handler;

    struct pdsc_long_with_handler {
        union {
            struct pdsc_long_stack_rpd  long_stack_rpd;
            struct pdsc_long_reg_rpd    long_reg_rpd;
        } stack_or_reg;
        pdsc_address       handler;
        pdsc_address       handler_data;
    } long_with_handler;

} pdsc_rpd;
```

SIZE - 40 bytes, ALIGNMENT - 8 bytes

Version Note

The fields marked "V5.1" in the preceding structure definition are new fields for Tru64 UNIX **V5.1** and greater. The new fields take the place of a reserved field so there is no change in the structure size.

See the *Calling Standard for Alpha Systems* for a full description.

3.3 Instructions and Data Usage

3.3.1 Minimal Objects

Many sections may be missing from a still-viable object file. Sections may not be present due to the type of the object file or to the contents of a particular program.

The `.init` and `.fini` sections of the text segment are typically not present in relocatable objects. They contain code generated during final link.

The allocation of data in the "small" and "large" writable data sections (`.sdata`, `.data`, `.sbss`, `.bss`) can be controlled by the user in some situations. See Section 3.3.6 for more details.

The `.lit4` and `.lit8` sections, which hold 4- and 8-byte literal values respectively, may be omitted from an object file. Compilers may choose not to emit these sections.

The `.xdata` and `.pdata` sections, which contain exception-handling information, may not be present. All pre-link objects with a non-empty text segment contain these sections because compilers are expected to provide exception-handling information for their code. Statically linked executables will only contain these sections if they include code which handles exceptions. The linker identifies exception handling code by looking for references to the `_fpdata_size` symbol. By default, shared objects will contain these sections. The `.xdata` and `.pdata` sections are required if a shared object includes exception handling code or if it is used in conjunction with another shared object that includes exception handling code.

Although most objects contain both text and data segments, only one loadable segment is required for an object to be loadable. A minimal pre-link object file may contain no sections.

3.3.2 Position-Independent Code (PIC)

Position-independent code is generated code that is not constrained to any particular location in the virtual address space. Eventually, code must be assigned to a portion of the address space where it can execute. However, on Tru64 UNIX, code is kept position-independent as long as possible.

The implementation of position-independent code in eCOFF relies upon address tables to store full virtual addresses for procedures and data locations invoked or referenced in the text segment. Programs refer to these addresses using a technique called GP-relative addressing.

Most eCOFF objects have address tables that hold 64-bit addresses. Address tables in shared objects are called Global Offset Tables (GOTs) and are found in the `.got` section. Address tables for relocatable and static objects are called literal address pools and are found in the `.lita` section.

Address table entries are accessed in code by adding a signed 16-bit offset to the currently active GP value, which is stored in the `$gp` register:

```
ldq t12,-31656(gp)
```

Multiple GP ranges can be associated with a program, each corresponding to a different portion of the address table. See Section 2.3.4 for details.

In some cases, special instruction sequences may be required to update the contents of the `$gp` register. In particular, the GP value used by a procedure may or may not be the same as the value used by the calling code. Under most circumstances, the called procedure's GP value is calculated when a procedure is invoked. Upon completion of the procedure's execution, the calling code's GP value must be reestablished. Refer to the *Calling Standard for Alpha Systems* for details.

Different kinds of objects use address tables in different ways:

- Relocatable Objects

  Pre-link objects usually have a `.lita` section with associated section relocation information. The literal address pool contains addresses that must be adjusted at link time.

- Static Executables

  Addresses in static executables are fixed at link time. The image must be loaded and executed at addresses the linker has chosen. Library addresses as well as segment base addresses are known at link time.

  Static executables store addresses in a `.lita` section that encompasses one or more GP ranges. The contents of the address table are accessed by means of the GP value or values, which are also fixed at link time.

- Shared Objects

  Each `.lita` entry in the input object files is relocated by the linker to form the GOT in the output object. The loader may need to update the GOT entries when mapping the process image. The addresses are then absolute and may be extracted at run time to obtain the final locations of referenced items.

The loader may also update GOT entries at run time, such as when it replaces lazy text stubs with resolved procedure addresses or dynamically loads new objects.

The GOT may contain entries for nonsymbolic text and data addresses. These are known as local GOT entries. The GOT may also contain entries for unresolvable symbols; which are either set to NULL or to the address of a lazy text stub routine.

Special semantics are associated with multiple GP ranges in shared objects. See Section 6.3.3.3 for details on multiple GOT representation and usage.

Code can be only partially position independent. For example, shared libraries can be mapped anywhere in the address space that is not in conflict with previously mapped objects, but executable objects must be mapped at their link-time base addresses. Dynamic executables are thus partly PIC because their own segment addresses are fixed, but the addresses of shared libraries they use are not. Static executables are position dependent (nonPIC) and can be optimized to rely on more efficient position dependent methods for accessing program addresses.

3.3.3 Lazy-Text Stubs

This section applies to shared objects only. See Section 6.3.4.5 for related information.

Final addresses may be unknown at link time for subroutines that are defined in shared libraries and called by dynamic executables. Instructions reference these routines in an address-independent manner, and the dynamic loader resolves the procedure's actual address the first time it is invoked.

Stubs are specially constructed code fragments used for this run-time symbol resolution. They serve as placeholders for the definitions of functions that cannot be resolved at static link time. The linker builds the stub for each called procedure and allocates GOT table entries that point to the stubs. The stubs themselves are inserted in the `.text` section of the shared object file by the linker.

A stub looks like this:

```
stub_xyz:
    ldq  t12, got_index(gp)           //load register with .got entry
                                      //  of lazy text resolver
    lda  $at, dynsym_index_low(zero)  //load register with external
    ldah $at, dynsym_index_high($at)  //  symbol's .dynsym index
    jmp  t12, (t12)                   //jump to lazy text resolver
```

The first time the procedure is called, its stub is invoked. The stub, in turn, calls the loader to resolve the associated symbol. The dynamic loader then replaces the stub address with the correct procedure address, which is used for subsequent calls.

The calling standard requires that when control actually reaches the procedure's entry point, register `$r27` must contain the procedure value of the newly loaded routine (as if no intermediate processing had occurred).

3.3.4 Constant Data

Constant data is data that cannot be changed over the course of program execution. It can include constants appearing in the source program, constants that are generated during the compilation process (usually addresses), and literal values (also referred to as immediate values).

Constant data may appear in any data section. It is likely to appear in the `.lit4`, `.lit8`, `.lita`, `.rconst`, and `.rdata` sections. Compilers and other object file producers may make varying choices concerning data placement in object file sections.

The literal sections contain only literal values sorted by sizes. 4-byte literals are stored in the `.lit4` section, 8-byte literals in the `.lit8` section, and 8-byte address literals in the `.lita` section. However, these sections do not necessarily contain all the literals in the program. String literals, for example, are assigned to the `.data` section (or `.rconst` section when the `-read_only_strings` compiler option is specified).

There are compile-time, link-time, and run-time constants. Examples of compile-time constants include numeric constant data such as floating-point constants and literals appearing in the source file. Examples of link-time constants include addresses that are fully resolved at link time. Examples of run-time constants include addresses established by the dynamic loader.

The linker places the `.rconst` section and all three literal sections with the text segment because they contain nonwritable data. The advantage of mapping constant data with a program's read-only segment is that it allows the data to be shared among processes.

The `.rdata` section contains constant data with values that may not be known until run time (such as global symbol addresses). For shared objects, the `.rdata` section is mapped with the data segment so the loader can perform relocations for that section without affecting the shareability of text or page table pages. If there are no dynamic relocations, the `.rdata` section may be mapped with the text segment.

3.3.5 INIT/FINI Driver Routines

Every compilation unit in an executable or shared library has the opportunity to contribute initialization or termination code to be run at startup and exit, respectively. INIT routines perform initialization actions and are run automatically at load time or by the routine `dlopen()`. FINI routines are termination functions that are executed by `dlclose()` or at program termination by `exit()`.

The `.init` and `.fini` sections consist of a series of calls to the initialization and termination routines. These calls, or drivers, are generated by the linker. They are not present in pre-link objects. The `.init` driver is invoked by a call from startup code in `/usr/lib/cmplrs/cc/crt0.o`, which must be linked into every executable object file.

The driver code in the `.init` and `.fini` sections has the following characteristics:

- No associated symbolic information

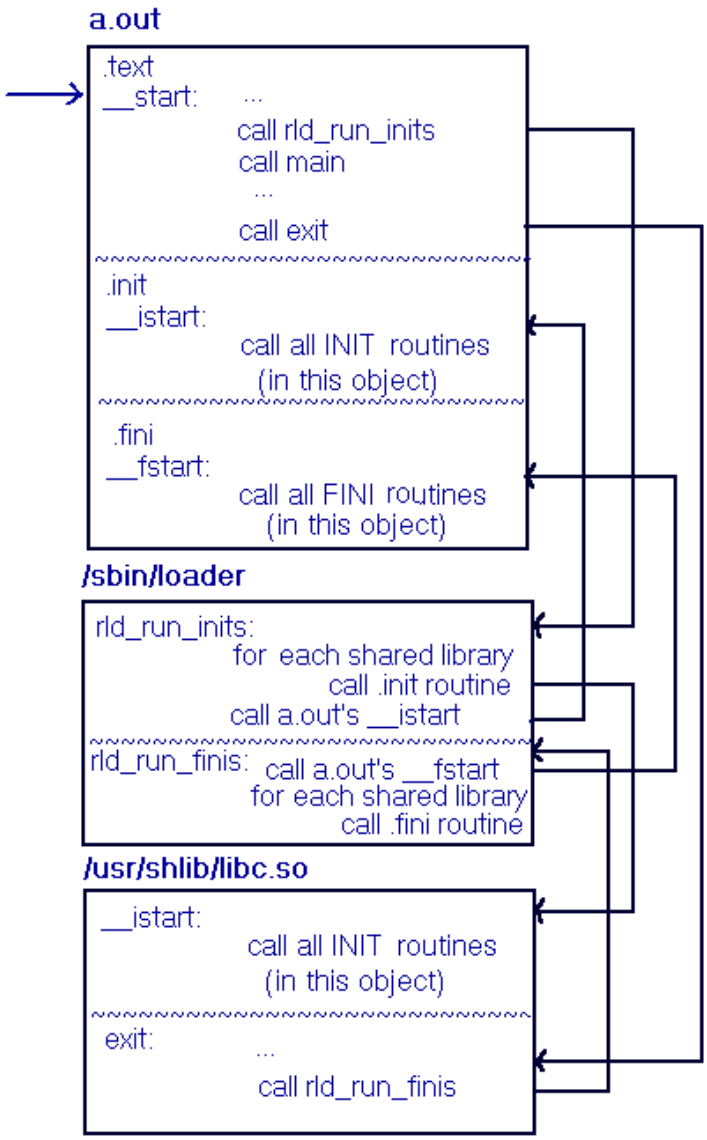
- No associated call frame information
- Use of self-relative code for jumping to the routines; therefore, no use of the GOT table or GP value

The initialization and termination routines themselves are in the `.text` section and have the following characteristics:

- No arguments
- No return value
- Defined in one of the objects or archives being linked

Figure 3–2 presents a graphical overview of the INIT/FINI mechanism for shared objects:

Figure 3–2: INIT/FINI Routines in Shared Objects



For static executables, the first call is to the main object's `__istart()` symbol instead of `rld_run_init()`. The dynamic loader is not involved.

System tools can generate initialization and termination routines. For example, global constructor and destructor routines for static objects are implemented as INIT/FINI routines by the C++ compiler.

The INIT/FINI mechanism is used for allocation and deallocation of thread-specific data. Every object using TLS has its own INIT routine to take care of the TLS data associated with that object. The purpose of this INIT routine is to allocate a TSD key that will be used for the object's TLS for the duration of the object mapping. See Section 3.3.9 for more information on TLS data.

3.3.5.1 Linking

INIT and FINI routines can be included implicitly, by prefix recognition, or explicitly, by option processing. With either linking method, as the routine's symbols are identified, a list determining the execution order is built. When the list is complete, code to invoke the routines is generated by the linker and placed in the `.init` and `.fini` sections.

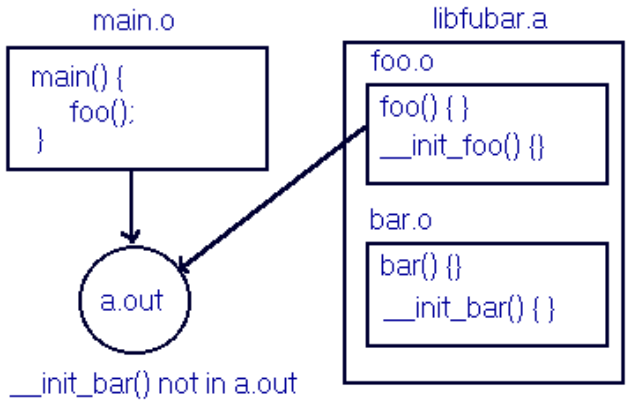
To link explicitly, the `-init` and `-fini` linker options are used with a symbol parameter. The symbol should meet the criteria listed above for INIT and FINI routines.

To link implicitly, it is necessary to conform to naming and usage conventions. A symbol is recognized as an initialization or termination symbol if:

- Automatic recognition of special symbols is not disabled.
- The symbol is defined in an object included in the link.
- The symbol bears the correct prefix (`__init_` or `__fini_`).
- The symbol is a procedure.

Library archives may contain aptly named routines that are not implicitly linked into an object as INIT or FINI routines. The reason this situation can occur is that prefix recognition alone is not sufficient cause to extract a module from an archive.

Figure 3–3: INIT/FINI Recognition in Archive Libraries



On the other hand, if the archived object is already linked into the object, prefix recognition will apply to routines contained in that module. Explicit inclusion can be used to ensure an archived routine is included as an initialization or termination routine in all cases. See the *Programmer's Guide* for more information on linking with archive libraries.

The linker's `-no_prefix_recognition` option disables implicit linking of INIT and FINI routines.

3.3.5.2 Execution Order

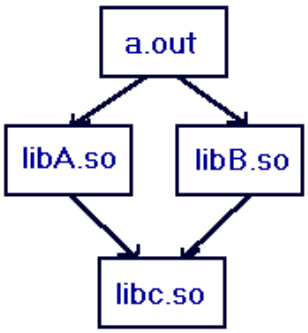
This section describes the execution order of initialization and termination routines in dynamic and static executables. It also covers the determining factors used by the linker and loader to establish this order.

3.3.5.2.1 Dynamic Executables

The INIT driver routine for each shared object is executed after INIT drivers for all of its dependencies. Dependencies are processed in a post-order traversal of the dependency graph. The dependency graphs shown in this section are based on link-line ordering (a left "sibling" appears first on the link line) as well as the shared library dependency information.

FINI drivers are executed in precisely the reverse order of INIT drivers.

Figure 3–4: INIT/FINI Example (I)

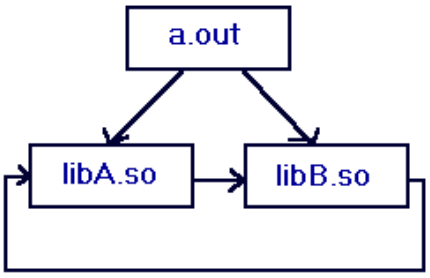


INIT order: `libc.so libB.so libA.so a.out`

FINI order: `a.out libA.so libB.so libc.so`

Cyclic dependencies are handled using a first-seen approach, while still conforming to the preceding rules. For example:

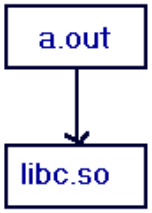
Figure 3–5: INIT/FINI Example (II)



INIT order: `libA.so libB.so a.out`

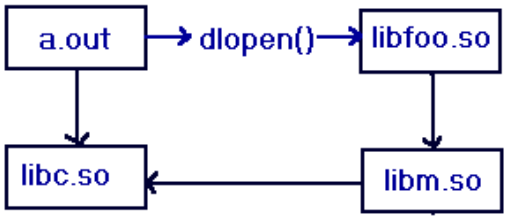
Initialization and termination routines may also be executed when shared objects are loaded and unloaded dynamically during run time. `dlopen()` runs INIT routines for any shared objects that it loads. `dlclose()` runs FINI routines for each shared object that it unloads.

Figure 3–6: INIT/FINI Example (III)



INIT order before dlopen call: `libc.so a.out`

Figure 3–7: INIT/FINI Example (IV)



INIT order after dlopen call: `libm.so libfoo.so`

FINI order after dlopen call: `libfoo.so libm.so a.out libc.so`

3.3.5.2.2 Static Executables

For static executables, the execution order for initialization and termination routines is determined at link time. The linker establishes the execution order for INIT routines by the order in which they are encountered within an object's external symbol table and by the ordering of objects on the command line. It also takes into account the ordering of archive libraries on the command line. The INIT routines from each archive are executed in the reverse order of their occurrence on the command line. For example:

```
$ ld x.o y.o z.o libm.a libfoo.a
```

INIT order: `libfoo.a libm.a x.o y.o z.o`

FINI order: `z.o y.o x.o libm.a libfoo.a`

3.3.5.2.3 Ordering Within Objects

It is also possible to have multiple INIT or FINI routines within an object. The number of initialization or termination functions that can be included from a single object is unlimited. When multiple routines are encountered in an input object, they are placed as a group within the overall ordering.

If both methods of linking are used, explicitly linked initialization routines are executed prior to the implicitly linked routines for that object. Because the FINI order is always the opposite of the INIT order, any explicitly linked termination routines are executed last.

If the linker's range-table generating routines are present, they execute first and last, respectively in INIT/FINI ordering on a per-object basis. These initialization routines set up a PC-range table that enables exception-handling. They execute first so that range information is added before other INIT routines are executed. These termination routines run last so that all others are run before range information is removed. These precautions allow other INIT and FINI routines to utilize exception handling.

3.3.5.2.4 Subsystem Control of INIT/FINI Order

Version Note

Subsystem generated initialization and termination routines are supported in Tru64 UNIX **V5.0** and greater.

Compilers may need to generate initialization and termination routines and to control the order in which they execute. For this reason, subsystem-generated INIT and FINI routines are distinguished from user INIT and FINI routines.

The linker recognizes a subsystem-generated routine by the prefixes `__INIT_` and `__FINI_`. Routines recognized with the `__INIT_` prefix always run prior to any routines recognized with the `__init_` prefix within the same executable or shared library. FINI routines recognized with the `__FINI_` prefix always run after any routines recognized with the `__fini_` prefix. Subsystem INIT and FINI routines also run, respectively, before and after any routines added by a user using the linker's `-init` and `-fini` switches.

All routines with the `__INIT_` prefix execute in alphabetic order, and all routines with the `__FINI_` prefix execute in reverse alphabetic order. For a name of the form `__INIT_`*ALPHANAME*, the *ALPHANAME* portion should be encoded as a variable-length hexadecimal string. The string will contain one or more hex digits followed by an underscore.

INIT routines generated by the linker for exception-handling, speculative execution, and thread-local storage run prior to all other INIT routines. The associated FINI routines run last.

3.3.6 Initialized Data and Zero-Initialized Data (bss)

Writable user-program data is divided between data (initialized data) and bss (zero-initialized data) sections, which may then be subdivided according to data element size. Zero-initialized data consists of program variables whose values are not specified at compile time. Initialized data includes all variables that are explicitly initialized in declaration statements.

One example of zero-initialized data is Fortran commons . Another is uninitialized C data (`int count;`).

Note that a C-global or C-static data item explicitly initialized to zero (`int count = 0;`) may be placed in an initialized data section, even though its value is the same as if it were part of bss.

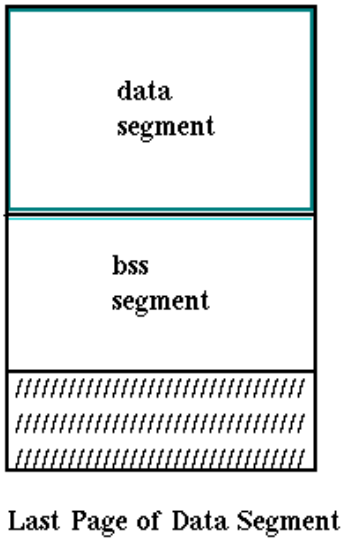
The primary advantage of separating initialized and uninitialized data is to save space in the object file. All bss data elements are set to the same value (zero). The only information required in the object file is a description of the run-time size and location of the bss sections. This description is found in the `.bss` and `.sbss` section headers.

Zero-filled memory is allocated for the bss segment when an object is mapped into memory. Because the `.bss` and `.sbss` raw data sections do not require space in

the object file, their section header size field reports the size of the section in the process image instead of in the object file.

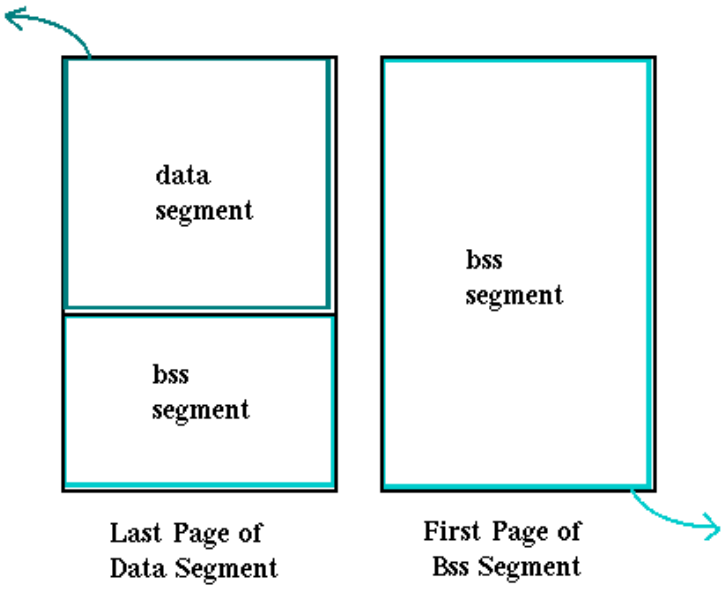
To take advantage of all available space, zero-initialized data immediately follows initialized data in the image. An object can have bss sections but no bss segment. If the data in the bss sections does not exceed the size of the leftover space in the last page of the data segment, the bss segment will be empty. This situation is illustrated in Figure 3–8.

Figure 3–8: Data and Bss Segment Layout (1)



For the same reason, some bss data can potentially be present in the data segment, even if a separate bss segment exists. This situation is illustrated in Figure 3–9.

Figure 3–9: Data and Bss Segment Layout (II)



When part or all of the bss segment is contained in the last page of a data segment, that portion of the data page must be initialized to zero in the corresponding raw data area of the object file.

The division of initialized and uninitialized data by size may split writable data into "small" (`.sdata`, `.sbss`) and "large" (`.data`, `.bss`) sections. It may be possible to exploit this division by grouping frequently used data together in a section. This strategy may enhance performance by reducing page faults. The size division may also allow post-link tools, such as `om` and `spike`, to generate more efficient code sequences for accessing data items.

The default maximum value for an item allocated in a "small" section is eight bytes. Some compilers accept a `-G` option with a parameter to specify the maximum size of a "small" data item. However, the default compilers on Tru64 UNIX do not.

When speaking of item size, note that an aggregate data item is considered as a whole. For example, a string of ten characters has a size of ten bytes.

3.3.7 Permissions/Protections

When a process image is created for a program, loadable segments are assigned access permissions. These are determined by the file's MAGIC number and the segment type.

Table 3–1: Segment Access Permissions

Image	Segment	Access Permissions
`OMAGIC`	text, data, bss	Read, Write, Execute
`NMAGIC`	text	Read, Execute
`NMAGIC`	data	Read, Write
`NMAGIC`	bss	Read, Write, Execute
`ZMAGIC`	text	Read, Execute
`ZMAGIC`	data	Read, Write
`ZMAGIC`	bss	Read, Write, Execute

3.3.8 Exception Handling Data

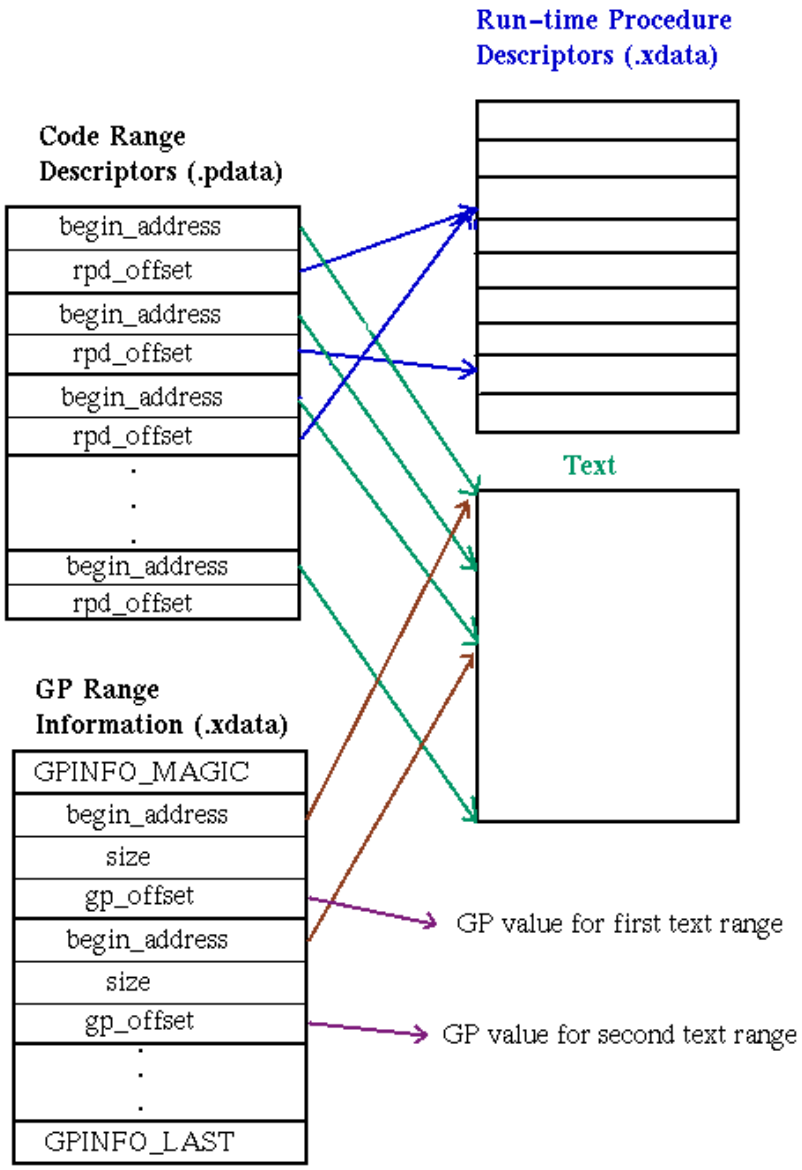
Exception handling is provided on the system to cope with unusual conditions. The object file contains two sections for storing exception-handling data structures. The declaration of these structures is shown in Section 3.2.

The object file sections `.xdata` and `.pdata` work together to provide exception-handling support. The `.xdata` section contains run-time procedure descriptors, GP range information, and user-specified exception data. The `.pdata` section contains code range descriptors. Exception information is produced for all pre-link object files. The linker produces exception information for dynamic executables and shared libraries because they will potentially be utilized in conjunction with other dynamic executables or shared libraries that rely on exception handling. The linker also produces exception information for static executables that reference `_fpdata_size`, a linker-defined symbol which represents the number of entries in the `.pdata` section.

A code range descriptor associates a contiguous sequence of addresses with a run-time procedure descriptor. The `.pdata` code range descriptors are ordered by run-time address. The ranges never overlap. The last `.pdata` entry is an end marker. It may be followed by padding.

The code range descriptor points into both the text segment and the run-time procedure descriptors, as shown in Figure 3–10. The relationship between code range descriptors and run-time procedure descriptors can be a many-to-one relationship. Also note that a code range descriptor may not have an associated run-time procedure descriptor.

Figure 3–10: Exception-Handling Data Structures



The virtual address space containing the text section of the object file is portioned into code ranges. Each code range descriptor has only one address, which indicates the beginning of the range. The range is implicitly ended just prior to the beginning address of the subsequent range. The final code range descriptor serves to end the range begun by the next-to-last descriptor, not to start a new range.

The GP range information can be accessed via the special symbol `_gpinfo` (see Section 2.3.7). It is an array of signed 64 bit integers. If the first entry is not `GPINFO_MAGIC` the GP range information should be ignored. The end of GP range information is identified by the constant `GPINFO_LAST`. (These constants can be found in /usr/include/excpt.h.) Each range of instructions with a unique GP value is represented by a set of three entries as shown in Figure 3–10.

`begin_address` The address of the first instruction in the GP range stored as an offset from `&_ftext`.

`size` Size in bytes of the GP range.

`gp_offset`	The GP value used for the GP range stored as an offset from `&_fdata`.

The *Programmer's Guide* and the *Calling Standard for Alpha Systems* provide detailed explanations of the exception-handling mechanisms supported by Tru64 UNIX. Related man pages such as `pdsc`(4) and `exception_intro`(3) are also available for quick reference.

C++ uses its own unique exception mechanism. An example illustrating the symbol table representation of C++ exception information can be found in Section 9.2.6.

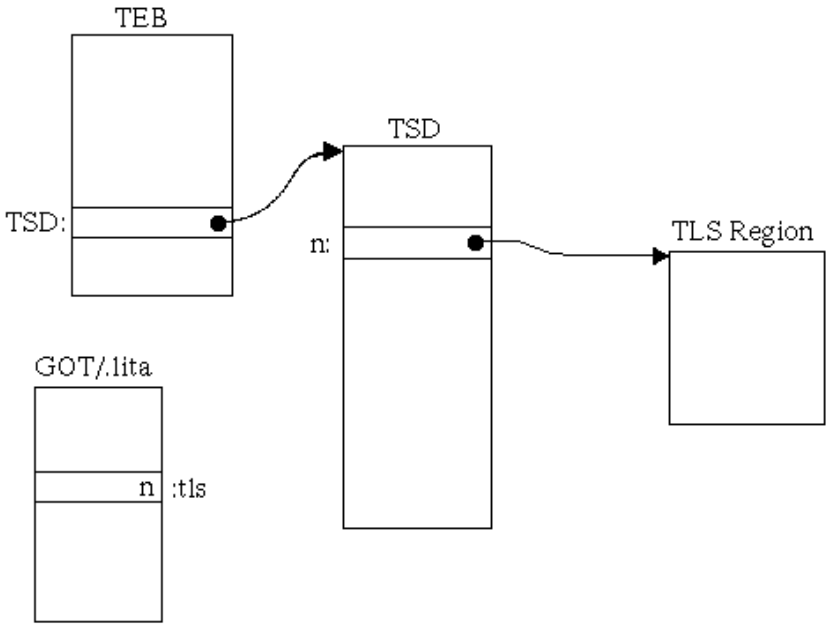
3.3.9 Thread Local Storage (TLS) Data

Threads are available on Tru64 UNIX as a way to increase processor utilization and overall application performance. Thread Local Storage (TLS) provides a way for an application writer to declare data that has multiple instances, one per thread. The object file has specific structures designed to store and manage TLS. These structures and the impact of TLS on the object file and symbol table are described here. For general information about threads programming, see the *Guide to DECthreads*.

Three object file sections are devoted to TLS data: `.tlsdata`, `.tlsbss`, and `.tlsinit`. The TLS region consists of the `.tlsdata` and `.tlsbss` sections. The `.tlsinit` section, which may be mapped with the object file's text or data segments, contains initialization information for `.tlsdata`. Objects containing TLS data are distinguished by the presence of these sections.

Structures outside the object file are used to reference TLS data. The Thread Environment Block (TEB) is an architected structure provided by system libraries. One of the fields in the TEB is the address of the Thread Specific Data (TSD) array, which contains pointers into the TLS region. Each object containing TLS will be allocated one or more TSD entries. In each thread, the TSD entries will contain the address of the start of a region of that thread's TLS area.

Figure 3–11: Thread Local Storage Data Structures



Because the TLS region is allocated dynamically and is unique per-thread, no address information can be recorded in the object file. All other attributes of the TLS region can be determined at link time and are recorded in the object file in the TLS data and TLS bss section headers.

The TLS data and bss sections occupy no space in the object file and do not have associated section relocation information.

The TLS INIT section contains the data which will be used to initialize each thread's instance of the TLS data section at run time. The TLS INIT section can contain relocation information. Only R_REFQUAD and R_REFLONG relocations are allowed, and the relocations must reference nonTLS symbols or sections.

The TLS region for a shared object consists of the initialized and zero-initialized TLS data defined by that object. The TLS region is composed of two sections: the TLS data section containing initialized TLS data (.tlsdata) and the TLS bss section (.tlsbss) containing zero-initialized TLS data.

If a shared object contains TLS data, an entry in the GOT (for the special symbol __tlsoffset) contains the offset into the TSD array to the array element that points to the TLS area. If this is a multiple-GOT shared object, the entry may be duplicated in each GOT. The value of the GOT entry is filled in at load time when the TLS initialization routine calls the loader with the allocated TSD key value.

If a static executable contains TLS data, the address of __tlsoffset will normally be accessed through a .lita entry that contains the value 2048, the offset to TSD key 256.

Special symbol types and relocation types are specific to TLS. See Chapter 5 and Chapter 4 for more information.

3.3.10 User Text and User Data Sections

The linker contains provisions for creating and relocating user-defined object file sections. This feature was implemented for a specific customer at the customer's request. It is very rarely used and minimally supported. This section is designed to provide only a general overview.

Any number of user sections can be added to an object file. See Section 2.3.2 for the placement of the user sections in the various object file layouts.

The section header for a user section has the same semantics as those used for other object file sections. The section flags are set to STYP_REG. The user creating the section chooses the section name. User text sections are distinguished from user data sections by their addresses. User text sections have text segment addresses, and user data sections have data segment addresses.

For user sections, the linker synthesizes special symbols for the start and end addresses of each section. These symbols take the form:

```
__fuser_section_SECTION_NAME
__euser_section_SECTION_NAME
```

where *SECTION_NAME* is the name in the section header. These linker-defined symbols are always strong symbols.

The linker also combines like-named user sections in multiple input files to form a single section in the output file.

User sections can only have external relocation records.

Namespace issues can arise due to the user's naming of these sections. It is the responsibility of the user to protect against and recognize errors caused by namespace issues.

3.4 Language-Specific Instructions and Data Features

Procedures with alternate entry points require multiple run-time procedure descriptors. See the *Calling Standard for Alpha Systems* for details.

C++ has exception handling facilities in addition to those discussed in this chapter.

C++ global constructors and destructors are implemented as initialization and termination routines invoked by driver code stored in the `.init` and `.fini` sections.

4

Relocation

The purpose of relocation is to identify and update storage locations that need to be adjusted when an executable image is created from input object files at link time. Relocation information enables the linker to patch addresses where necessary by providing the location of those addresses and indicating the type of adjustments to be performed. Relocation entries in the section relocation information are created by the assembler, compiler, or other object producer, and the address adjustments are performed by the linker.

The linker performs relocation fixups after determining the linked object's memory layout and selecting starting addresses for its segments. During partial links, relocation information is updated and preserved for subsequent links. Relocation updates for partial links include converting external relocation entries to local relocation entries and retargeting relocation entries to new section addresses. See Section 4.3.2.1 for details.

Relocation information contained in an object file can have four distinct representations:

- Relocation entries identified in section headers. These are the relocation entries referred to in this document as "normal" or "actual".
- Compact relocation records, produced by the linker and consumed by `om`, `spike`, and profiling tools. Compact relocations are stored in the `.comment` section.
- Linkerdef entries which are produced by the linker to identify all uses of linker-defined symbols. Linkerdef entries are stored in the `.comment` section.

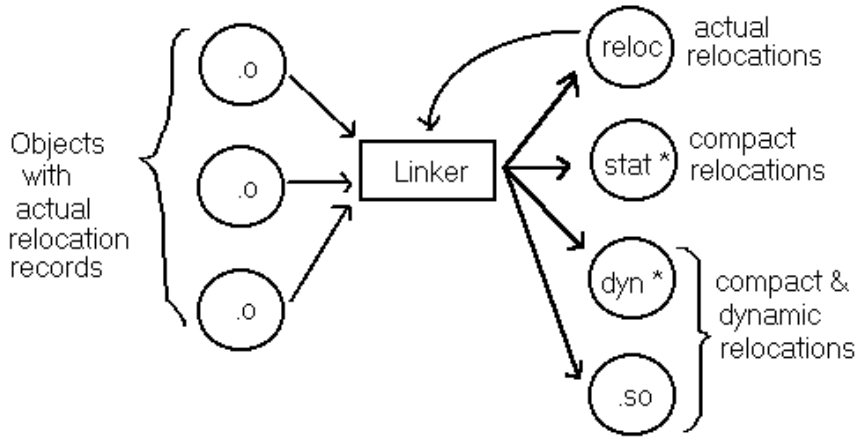
Version Note

Linkerdef entries are supported in Tru64 UNIX **V5.1** and greater for object format **V3.13** and greater.

- Dynamic relocations, which are present only in shared objects. Dynamic relocation may be performed for shared objects at load time.

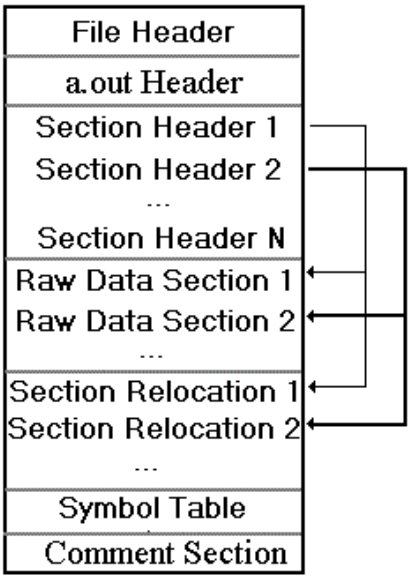
The first three forms of relocation information are discussed in this chapter. Compact relocations are discussed in Section 4.4 and linkerdef relocations are covered in Section 4.5. The fourth form is covered in Chapter 6. Figure 4–1 summarizes which kinds of objects contain which kinds of relocation information.

Figure 4–1: Kinds of Relocations



Actual relocation entries are organized by raw data section. Not all object file sections necessarily have relocation entries associated with them. For example, bss sections do not have relocation entries because they do not have raw data to relocate. Section headers for sections with relocation entries contain pointers to the appropriate section relocation information, as shown in Figure 4–2.

Figure 4–2: Section Relocation Information in an Object File



Note that the ordering of section headers does not necessarily correspond to the ordering of raw data and section relocation information. Consumers should rely on the section header to access this information.

4.1 New or Changed Relocations Features

Tru64 UNIX V5.1 introduces the following new or changed features:

- Full compact relocations. See Section 4.4.
- Linkerdef relocations. See Section 4.5.

4.2 Structures, Fields, and Values for Relocations

4.2.1 Relocation Entry (`reloc.h`)

```
struct reloc {
        coff_addr    r_vaddr;
        coff_uint    r_symndx;
        coff_uint    r_type  : 8;
        coff_uint    r_extern: 1;
        coff_uint    r_offset:6;
        coff_uint    r_reserved:11;
        coff_uint    r_size:6;
};
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

Relocation Entry Fields

`r_vaddr` — Virtual address of an item to be relocated.

If the `s_nreloc` field in the section header overflows, this field contains the number of relocation entries for the section. This possibility applies only to the first entry in a section's relocation information. See Section 4.2.4 for more information.

`r_symndx` — For an external relocation entry, `r_symndx` is an index into external symbols. For a local relocation entry, `r_symndx` is the number of the section containing the symbol. Table 4–1 lists the section numbering.

For entries of type `R_LITUSE`, this field contains a subtype. See Table 4–3.

`r_type` — Relocation type code. Table 4–2 lists all possible values.

`r_extern` — Set to 1 for an external relocation entry. Set to 0 for a local relocation entry.

`r_offset` — For an entry of type `R_OP_STORE`, `r_offset` is the bit offset of a field within a quadword. For other relocation types, the field is unused and must be zero.

`r_reserved` — Must be zero.

`r_size` — For an entry of type `R_OP_STORE`, `r_size` is the bit size of a field. For `R_IMMED_*` entries, it is a subtype. See Table 4–4. For other relocation types, the field is unused and must be zero.

Table 4–1: Section Numbers for Local Relocation Entries

Symbol	Value	Description
R_SN_NULL	0	no section
R_SN_TEXT	1	.text section
R_SN_RDATA	2	.rdata section
R_SN_DATA	3	.data section
R_SN_SDATA	4	.sdata section
R_SN_SBSS	5	.sbss section

Table 4–1: Section Numbers for Local Relocation Entries (cont.)

Symbol	Value	Description
R_SN_BSS	6	.bss section
R_SN_INIT	7	.init section
R_SN_LIT8	8	.lit8 section
R_SN_LIT4	9	.lit4 section
R_SN_XDATA	10	.xdata section
R_SN_PDATA	11	.pdata section
R_SN_FINI	12	.fini section
R_SN_LITA	13	.lita section
R_SN_ABS	14	for R_OP_*xxxx* constants
R_SN_RCONST	15	.rconst section
R_SN_TLSDATA	16	.tlsdata section
R_SN_TLSBSS	17	.tlsbss section
R_SN_TLSINIT	18	.tlsinitsection
R_SN_RESTEXT	19	**(not supported)** .restext section
R_SN_GOT	20	**(V5.1 -)**.got section

Version Note

The R_SN_RESTEXT value is reserved for Tandem big-endian systems. It is not used on Tru64 UNIX.

Table 4–2: Relocation Types

Symbol	Value	Description
R_ABS	0x0	Relocation already performed
R_REFLONG	0x1	A 32-bit reference to symbol's virtual address
R_REFQUAD	0x2	A 64-bit reference to symbol's virtual address
R_GPREL32	0x3	A 32-bit displacement from the global pointer to a symbol's virtual address
R_LITERAL	0x4	A reference to a literal in the literal address pool as an offset from the global pointer
R_LITUSE[1]	0x5	An instance of a literal address previously loaded into a register
R_GPDISP	0x6	An lda/ldah instruction pair that is used to initialize a procedure's global-pointer register
R_BRADDR	0x7	A 21-bit branch reference to the symbol's virtual address
R_HINT	0x8	A 14-bit jsr hint reference to symbol's virtual address
R_SREL16	0x9	A 16-bit self-relative reference to symbol's virtual address
R_SREL32	0xa	A 32-bit self-relative reference to symbol's virtual address
R_SREL64	0xb	A 64-bit self-relative reference to symbol's virtual address
R_OP_PUSH	0xc	A 64-bit virtual address to push on the relocation expression stack
R_OP_STORE	0xd	An address to store the value popped from the relocation expression stack

Table 4–2: Relocation Types (cont.)

Symbol	Value	Description
R_OP_PSUB	0xe	A symbol's virtual address to subtract from value at the top of the relocation expression stack
R_OP_PRSHIFT	0xf	The number of bit positions to shift the value at the top of the relocation expression stack
R_GPVALUE	0x10	A new GP value to be used for the address range starting with the address specified by the r_vaddr field
R_GPRELHIGH	0x11	The most significant 16 bits of a 32-bit from the global pointer to a symbol's virtual address
R_GPRELLOW	0x12	The least significant 16 bits of a 32-bit from the global pointer to a symbol's virtual address
R_IMMED[2]	0x13	An instruction sequence that calculates an address
R_TLS_LITERAL	0x14	The instruction that loads the TLS key
R_TLS_HIGH	0x15	The most significant 16 bits of a 32-bit from the TLS region pointer to a symbol's virtual address
R_TLS_LOW	0x16	The least significant 16 bits of a 32-bit from the TLS region pointer to a symbol's virtual address

Table Notes

1. The r_symndx field for the relocation type R_LITUSE is a subtype. The valid entries for this field and their meanings are summarized in Table 4–3.
2. The r_size field for the relocation type R_IMMED is a subtype. The valid entries for this field and their meanings are summarized in Table 4–4.

Table 4–3: Literal Usage Types

Symbol	Value	Description
R_LU_BASE	1	The base register of a memory format instruction (except ldah) contains a literal address
R_LU_BYTOFF	2	Should not be used
R_LU_JSR	3	The target register of a jsr instruction contains a literal address

Table 4–4: Immediate Relocation Types

Symbol	Value	Description
R_IMMED_GP_16	1	16-bit displacement from GP value
R_IMMED_GP_HI32	2	Most significant 16 bits of 32-bit displacement from GP value
R_IMMED_SCN_HI32	3	Most significant 16 bits of 32-bit displacement from section start
R_IMMED_BR_HI32	4	Most significant 16 bits of 32-bit displacement from instruction following branch
R_IMMED_LO32	5	Least significant 16 bits of 32-bit displacement specified by last R_IMMED_*_HI32

4.2.2 Compact Relocation Records)

Compact relocation records are written into the free-form data area of the comment section. They are identified by a tag type of CM_COMPACT_RLC in the comment header. The public versions of compact relocation interfaces for producers and

consumers are located in the header file cmplrs/cmrlc.h. See Section 4.4 and Chapter 7 for more information.

4.2.3 Linkerdef Relocation Records (scncomment.h)

Linkerdef relocation records are written into the free-form data area of the comment section. They are identified by a tag type of CM_LINKDERDEF in the comment header. The Linkerdef comment subsection is an array of linker_data structures that contain information similar to the reloc structure. See Section 4.5 and Chapter 7 for more information.

Version Note

The linker_data structure is supported on Tru64 UNIX **V5.1** and greater.

```
struct linker_data {
        unsigned int    ld_scnptr;
        unsigned int    ld_base   : 6;
        unsigned int    ld_symbol : 6;
        unsigned int    ld_type   : 8;
        unsigned int    ld_size   : 6;
        unsigned int    ld_offset : 6;
};
```

SIZE - 8 bytes, ALIGNMENT - 4 bytes

Linkerdef Relocation Entry Fields

ld_scnptr	A byte offset relative to the starting file offset of the section identified by ld_base. Together, these fields identify the target address for the relocation.
ld_base	The number of the section containing the target address. See Table 4–1 for a list of valid section numbers.
ld_symbol	An enumeration value identifying a linker-defined symbol. See Section 4.2.3.1 for a list of valid values.
ld_type	A relocation type. See Table 4–2 for a list of relocation types.
ld_size	The size of a bitfield for the R_OP_STORE relocation.
ld_offset	The bit offset of a bitfield for the R_OP_STORE relocation.

4.2.3.1 Linkerdef Symbol Enumeration

Linker-defined symbols are identified by the following enumeration. Each enumeration value corresponds to the linker-defined symbol of the same name (excluding the "LDEF_" prefix).

Version Note

The LD_SYMBOL enumeration is supported on Tru64 UNIX **V5.1** and greater.

```
enum LD_SYMBOL {
      LDEF__BASE_ADDRESS            = 0,
      LDEF__cobol_main              = 1,
      LDEF__DYNAMIC                 = 2,
```

```
        LDEF__DYNAMIC_LINK              = 3,
        LDEF__ebss                      = 4,
        LDEF__edata                     = 5,
        LDEF_edata                      = 6,
        LDEF__end                       = 7,
        LDEF_end                        = 8,
        LDEF__etext                     = 9,
        LDEF_etext                      = 10,
        LDEF__fbss                      = 11,
        LDEF__fdata                     = 12,
        LDEF__fpdata                    = 13,
        LDEF__fpdata_size               = 14,
        LDEF___fstart                   = 15,
        LDEF__ftext                     = 16,
        LDEF__ftlsinit                  = 17,
        LDEF_GOT_OFFSET                 = 18,
        LDEF__gp                        = 19,
        LDEF__gpinfo                    = 20,
        LDEF___istart                   = 21,
        LDEF__procedure_string_table    = 22,
        LDEF__procedure_table           = 23,
        LDEF__procedure_table_size      = 24,
        LDEF___tlsbsize                 = 25,
        LDEF___tlsdsize                 = 26,
        LDEF___tlskey                   = 27,
        LDEF___tlsoffset                = 28,
        LDEF___tlsregions               = 29,
        LDEF_MAX
};
```

4.2.4 Section Header

The section header contains a file pointer to the section's relocation information and the number of entries. (See Section 2.2.3 for the declaration.) The number of relocation entries for a section is contained in the section header field `s_nreloc`. If that field overflows, the section header flag `S_NRELOCS_OVFL` is set and the first relocation entry's `r_vaddr` field stores the actual number of relocation entries for the section. That relocation entry has a type of `R_ABS` and all other fields are zero, causing it to be ignored during relocation.

4.3 Relocations Usage

4.3.1 Relocatable Objects

An object is relocatable if it contains enough relocation information for the linker to successfully relocate it. Relocatable objects can be produced by compiling without linking or by partial linking.

Compilers and assemblers always produce relocatable objects. By default, the relocatable object files produced are passed to the linker to produce a non-relocatable executable object. Most compilers recognize a `-c` option. The `-c` option suppresses the link operation and writes the object file in its relocatable form. For example, the following command produces a non-executable `OMAGIC` file named `pgm.o`.

```
$ cc -c pgm.c
```

By means of partial linking, the linker can also produce a relocatable object. By default, the linker attempts to produce an executable `ZMAGIC` file for which all relocation entries have been processed and removed. To preserve relocation information, the linker's `-r` switch should be selected. For example, the following command produces a non-executable `OMAGIC` file named `a.out`.

```
$ ld -r pgm.o
```

Selection of the `-r` switch has other effects: common storage class symbol allocation is deferred until final link and undefined symbol error messages are suppressed.

Relocatable objects have various uses. The most obvious is as input to a subsequent partial or final link operation. All objects input to the linker are relocatable objects, regardless of how they are produced. Multiple relocatable objects can be combined during a final link to produce an executable object. The typical example of this process is when several separately compiled modules are created at different times and later linked together to produce the final executable program. For example, the following steps produce an executable ZMAGIC file named a.out.

```
$ cc -c part1.c
$ cc -c part2.c
$ cc -c part3.c
$ cc part1.o part2.o part3.o
```

Relocatable objects are also used for archives. Although files of any type may be archived, one important use of archives is for user or system libraries. An example is the system library libc.a, which is linked with many C programs. Objects in archive libraries must be relocatable to be linked with other object files to make executable programs.

Relocatable objects may be used as loadable device drivers, which are object files that are dynamically added to a running kernel. See *Reference Pages, Section 9r, Device Drivers (Volume 1)* and *Reference Pages, Section 9s, 9u, and 9v, Device Drivers (Volume 2)* for more information.

Relocatable objects can also be used by the bootlinker, which builds the kernel from object files at boot time. Information is available in the *System Administration* guide.

Some profiling tools require relocatable objects as input because they rebuild the object and require the capability of rearranging raw data. However, on Tru64 UNIX, these tools rely on compact relocations, which are an alternate form of relocation information. Compact relocations are described in Section 4.4.

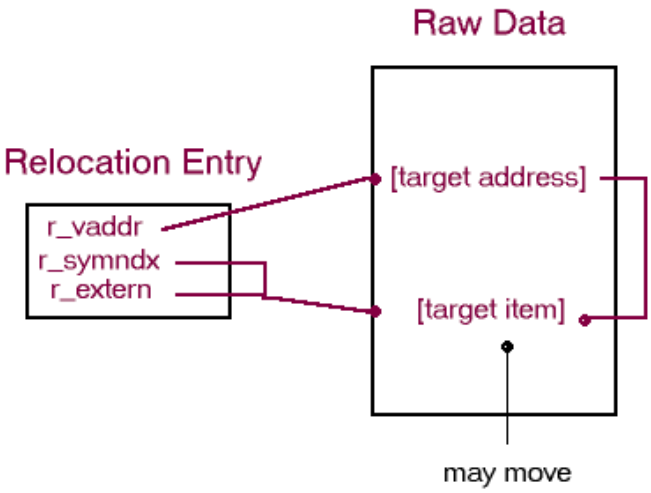
4.3.2 Relocation Processing

This section describes the generic process of relocating object files from a high-level viewpoint. It does not include details of address calculations, nor does it take into account the substantial variations in the contents of a relocation entry's fields. For specifics, see Section 4.3.4.

Relocation involves tracking and updating references as the referenced items move in memory. At a minimum, one relocation entry is required for each reference made to an item whose address may potentially change. This address, pointed to by the reloc structure field r_vaddr, is the target address of the relocation. This address is adjusted when relocation records are preserved at link time. The target address is located in one of the raw data sections of the object file.

The target address points to another item in the raw data. This item can be a data item, procedure, or any program element that will potentially be mapped to a new memory location when the linker builds the executable object.

Figure 4–3: Relocation Entry



Note that a many-to-one relationship may exist between relocation entries and target items. A target item may be addressed multiple times in an object file's raw data, and a single target address reference may be described by multiple relocation entries.

Taken together, the `r_symndx` field and `r_extern` bit track the position of the target item. If it is moved to a new location, the target address is updated accordingly.

The value of the relocation is the distance that the tracked item will move in memory.

4.3.2.1 Local and External Entries

Relocation entries are used for several purposes:

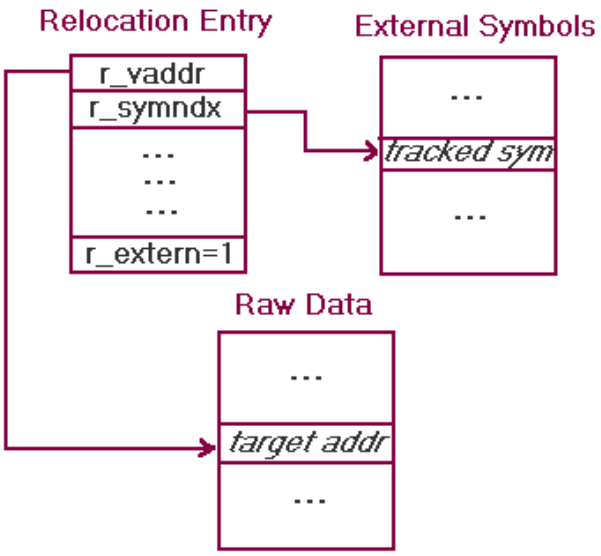
- Address references to unresolved symbols that will be imported from other objects.
- References to addresses within an object that may change when the object is linked at a different base address or linked with other object files.
- Identification of address references that may be optimized at link time.

Relocation entries may be local or external. Local relocation entries are used for references to addresses within an object. External relocation entries are used for references to any external symbols. In particular, unresolved symbol references can only be represented by external relocation entries.

The `r_extern` flag is set in external relocation entries. This flag determines the interpretation of the `r_symndx` field. For external entries, this field provides the external symbol table index of the referenced symbol.

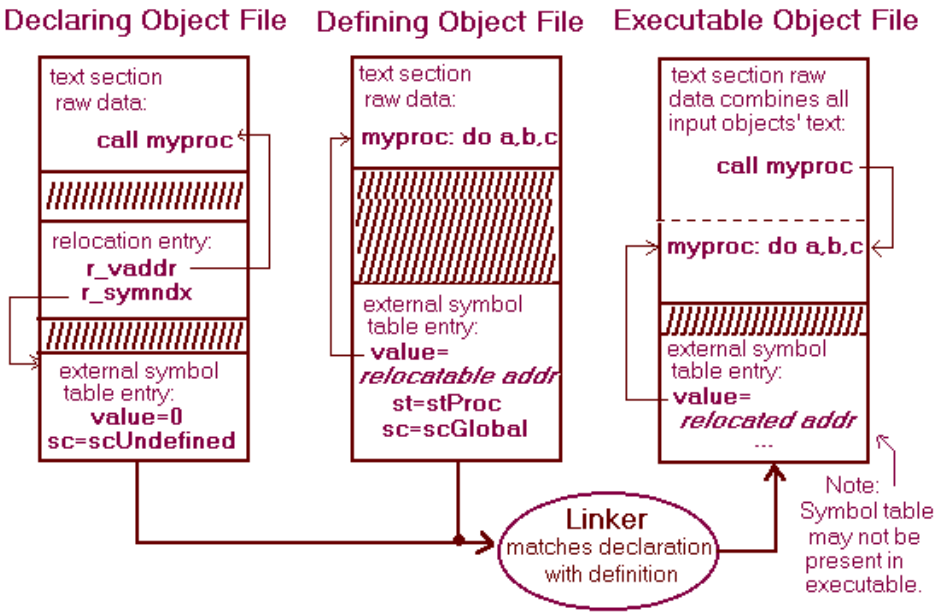
Figure 4–4 shows a sample external relocation entry.

Figure 4–4: External Relocation Entry



For an external entry, the value for relocation is the run-time address of the referenced external symbol. In cases where the symbol is undefined in an input object, it must first be resolved. Figure 4–5 depicts this process.

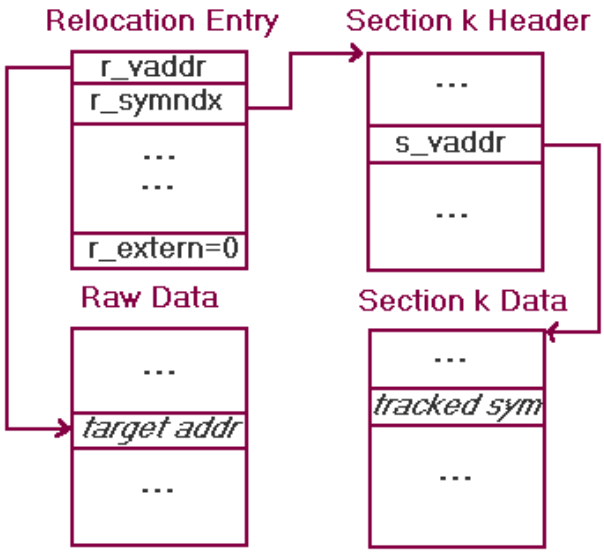
Figure 4–5: Processing an External Relocation Entry



A local relocation entry has its `r_extern` flag cleared and tracks references by section.

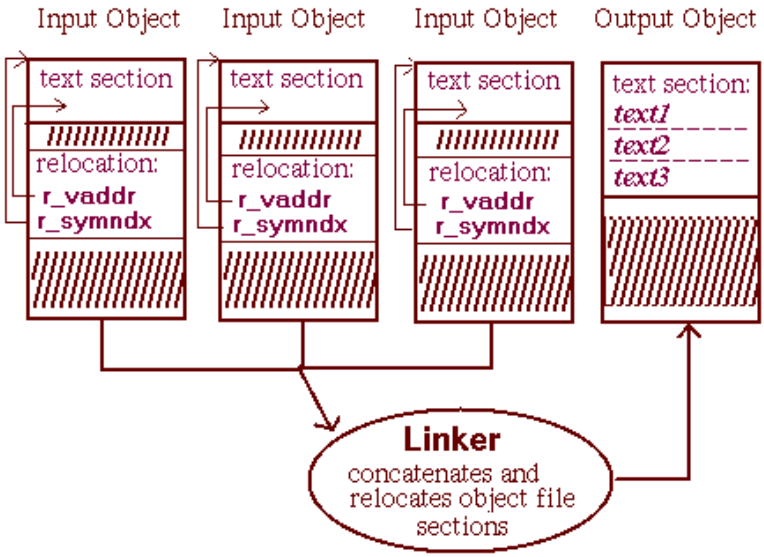
Figure 4–6 shows a sample local entry.

Figure 4–6: Local Relocation Entry



For a local entry, the value for relocation is the difference between a section's address in the input object and the address of that section's data after linking. The section is identified by a relocation section type in `r_symndx`. Figure 4–7 depicts this situation.

Figure 4–7: Processing a Local Relocation Entry

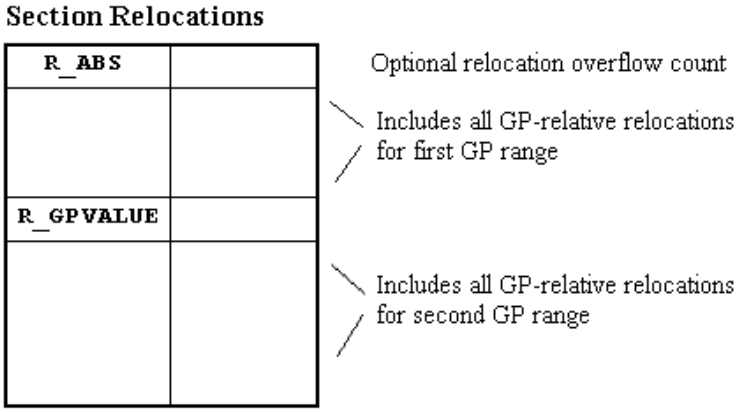


To complete relocation for all entries, the base address for the final process image is required. The linker can then use that address to patch all relocatable entries.

4.3.2.2 Relocation Entry Ordering

The ordering of relocation entries is sometimes significant. The diagram below shows the optional relocation entry count and grouping of relocation entries according to GP range.

Figure 4–8: Relocation Entry Ordering Requirements



If a section requires an optional relocation entry overflow count, it must be in the first relocation entry.

Relocation processing tools require GP-relative relocations to be grouped by GP range. R_GPVALUE entries will effectively separate the groups of GP-relative relocation entries for each GP range. For a list of GP-relative relocation types, see Section 4.3.3.2.

Some relocation types can only be used when paired with other relocation types. These relocation groupings are:

- R_GPRELHIGH, R_GPRELLOW
- R_TLSHIGH, R_TLSLOW
- R_LITERAL, R_LITUSE
- R_OP_PUSH, R_OP_PSUB, R_OP_PRSHIFT, R_OP_STORE

An R_GPRELHIGH entry must be followed by one or more R_GPRELLOW entries.

An R_TLSHIGH entry must be followed by one or more R_TLSLOW entries.

An R_LITERAL entry may be followed by zero or more R_LITUSE entries.

An R_OP_PUSH entry must be followed by exactly one R_OP_STORE entry. Zero or more R_OP_PSUB and R_OP_PRSHIFT entries may be located between the R_OP_PUSH and R_OP_STORE entries.

4.3.2.3 Shared Object Transformation

Part of the linker's preparation of loading information for shared objects is to create dynamic relocation entries from some of the actual relocation entries.

The linker must determine which relocation entries need to be converted to dynamic relocation entries. Data references (R_REFQUAD and R_REFLONG relocation types) must be represented in the .rel.dyn section if they are not in the .lita section. The .lita section is an exception because its contents are mapped directly into the GOT. All other R_REFQUAD or R_REFLONG entries have an associated dynamic relocation entry in the shared object file.

Dynamic relocation entries are not permitted for text addresses. The text segment is not mapped with write permission, so text relocation fixups cannot be performed by the dynamic loader.

4.3.3 Kinds of Relocations

Relocations types can be grouped into the following categories:

- Direct Relocations
- GP-relative Relocations
- Self-relative Relocations
- Literal Relocations
- Relocations Stack Expressions
- Immediate Relocations
- TLS Relocations

The categories often overlap.

4.3.3.1 Direct Relocations

Direct relocations are independent entries; all of the information necessary to process them is self-contained. The relocation target contains either the address of a relocatable symbol or an offset from that address. They are used for simple address adjustments; addresses in the literal address pool (`.lita` section), for example, will have associated direct relocation entries.

`R_REFQUAD` and `R_REFLONG` are direct relocation types. `R_REFQUAD` indicates a 64-bit address and thus is normally used on Alpha systems. `R_REFLONG` indicates a 32-bit address and most often occurs when the xtaso environment is in effect. These types of relocations are processed in the manner described in Section 4.3.2.

The following special requirements exist for direct relocation entries for the `.lita` section:

- Only entries of type `R_REFQUAD` or `R_REFLONG` are permitted.
- `R_REFLONG` entries pertain to the bottom 4 bytes of a `.lita` entry. The size of the entry is unchanged, but an error is generated if the result overflows 4 bytes.
- All external entries must correspond to symbols whose value is zero prior to relocation.

4.3.3.2 GP-Relative Relocations

This class of relocations requires use of the GP value as a factor in the calculation. Note that the literal relocations in Section 4.3.3.4 and Section 4.3.3.7 also fit this category.

The `R_GPREL32`, `R_GPRELHIGH`, `R_GPRELLOW`, and `R_GPDISP` relocation types are GP-relative. They typically point to instructions that calculate or load addresses using a GP value. The `R_GPRELHIGH` and `R_GPRELLOW` relocation types must be used together. The `R_GPDISP` relocation type is used for instruction pairs that load the GP value.

A special-purpose GP-relative relocation entry specifies that a new GP range is in effect. The relocation type for this entry is `R_GPVALUE`. The linker inserts `R_GPVALUE` entries at object module boundaries during a partial link (`ld -r`) when the `.lita` section it is building would otherwise overflow. Entries of this type appear in the `.text` section or the `.rdata` section. These entries are local entries because they are not tied to any symbol.

4.3.3.3 Self-Relative (PC-Relative) Relocations

This class of relocations require adjustments based on the current position in the text or data. Self-relative relocations are also referred to as PC-relative relocations.

The R_SREL16, R_SREL32, and R_SREL64 relocation types apply to 16, 32, and 64 bit target addresses, respectively.

Two more self-relative relocation types are R_BRADDR and R_HINT. R_BRADDR is used to identify branching instructions whose targets are known at link time. R_HINT is used to adjust the branch-prediction hint bits in jump instructions.

4.3.3.4 Literal Relocations

This category of relocations encompasses both literal relocations (type R_LITERAL) and literal-usage relocations (type R_LITUSE), which work together to describe text references.

A literal relocation (type R_LITERAL) occurs on a load of an address from the .lita section. Any associated R_LITUSE entries always directly follow the R_LITERAL entry.

The literal-usage entries are used for linker optimizations. Processing for these relocation entries is optional. The linker and other tools may ignore these relocation entries with no risk of producing an improperly relocated object file.

The advantage of literal-usage entries is that they enable link-time memory-access optimizations. These relocation entries identify instructions which use a previously loaded literal. With this knowledge, the linker is able to determine that certain instructions are unnecessary or can be altered to improve performance. Optimization is performed only during final link and with an optimization level setting of at least -O1.

4.3.3.5 Relocation Stack Expressions

Relocation stack expressions constitute a sequence of relocation entries that must be evaluated as a group. The purpose of stack expressions is to provide a way to represent complex relationships between relocatable addresses and store results with bit field granularity. They are currently used only for exception-handling sections.

An additional advantage of stack expressions is that they provide the capability to describe a new relocation type without requiring tool support or code modification to recognize and execute a new r_type. However, the greater flexibility of relocations expressions is offset by the fact that multiple entries are necessary to describe a single fix-up.

Special relocation types are used to build relocation expressions. The types are:

- R_OP_PUSH
- R_OP_STORE
- R_OP_PSUB
- R_OP_PRSHIFT

An R_OP_PUSH entry marks the beginning of a sequence of relocation stack expressions and an R_OP_STORE marks the end. The types of any intervening relocation entries should be either R_OP_PRSHIFT to shift the top of stack value right or R_OP_PSUB to subtract an address from the top of stack value.

An R_OP_STORE entry pops the value from the top of the expression stack and stores selected bits into a field in a word in memory. The r_offset and r_size fields of a relocation entry are used to specify the target bit field.

It is an error to cause stack underflow or to have values left on the stack when section relocation is complete.

Currently, these relocation types are used exclusively for relocating the exception-handling data in .xdata and .pdata. The reason this relocation is performed using the stack expression types is the need to shift the address by two bits. Bit field granularity cannot be specified with other relocation types unless it is implicit in the relocation type.

4.3.3.6 Immediate Relocations

Immediate relocations are used to describe the linker's optimization of literal pool references. If optimization options are in effect, the linker will replace R_LITERAL and R_LITUSE entries with R_IMMED entries wherever possible. This information is then used to generate compact relocations that sufficiently describe all relocatable storage locations.

Immediate relocations can describe instruction sequences that calculate addresses by adding either a 16-bit or 32-bit immediate displacement to a base address. R_IMMED entries always point to memory-access instructions. The displacement is obtained from the instruction.

There are five types of immediate relocations. Subcodes in the r_size field identify them. The types are:

- R_IMMED_GP_16
- R_IMMED_GP_HI32
- R_IMMED_SCN_HI32
- R_IMMED_BR_HI32
- R_IMMED_LO32

R_IMMED_GP_16 and R_IMMED_GP_HI32 entries identify address calculations performed by adding an offset to the global pointer. An R_IMMED_SCN_HI32 entry is paired with an R_IMMED_LO32 entry to identify a pair of instructions which add a 32 bit displacement to the starting address of a section. An R_IMMED_BR_HI32 entry is paired with an R_IMMED_LO32 entry to identify a pair of instructions which add a 32 bit displacement to the address of an instruction following a branch.

4.3.3.7 TLS Relocations

The types R_TLS_LITERAL, R_TLS_LOW, and R_TLS_HIGH are TLS-specific relocation types.

R_TLS_LITERAL is very similar to R_LITERAL, except it relates to a literal in the TLS data storage area, the TSD array. R_TLS_LOW and R_TLS_HIGH entries are used as a pair to identify instructions which load a TLS data address by adding a 32 bit offset to the TLS region pointer. These relocation types are identical to the R_GPRELHIGH and R_GPRELLOW relocation types except for the fact that the target instructions for the TLS relocation entries calculate addresses using the TLS region pointer instead of the GP value.

4.3.4 Relocation Entry Types

The type of a relocation entry (stored in the r_type field) describes the action the linker must perform. This section discusses the purposes of the different types and provides examples of their use.

Relocation entry fields are interpreted differently based on relocation type. There also may be constraints on fields' contents depending on the type. Some relocation entries are context sensitive and must be preceded or followed by a particular entry. Some are size specific and the computed address must fall within a specified range.

Moreover, some types are constrained to be local entries only or are associated with particular object file sections.

To describe the calculations performed by the linker, the following notation is used in the detailed descriptions for each relocation type:

`*_disp`	The displacement field of whatever instruction is indicated.
`GP`	Current GP value; begins as the contents of `AOUTHDR.gp_value` for the final object.
`new_scn_addr`	The address of the tracked section of a local relocation entry, as calculated by the linker.
`old_GP`	GP value in the input object; begins as `AOUTHDR.gp_value` for the input object.
`old_scn_addr`	The contents of `s_vaddr` in the section header of the input object file for the tracked section of a local relocation entry.
`[r_vaddr]`	The contents at the address `r_vaddr`; to be distinguished from the address itself.
`SEXT`	The constant immediately following is sign-extended.
`stack`	The relocation expression stack.
`this_new_addr`	Where `r_vaddr` will be after relocation .
`this_new_scn_addr`	Where the section containing `r_vaddr` will be after relocation, as calculated by the linker.
`this_old_scn_addr`	The contents of `s_vaddr` in the section header of the input object file for the section containing `r_vaddr`.
`tos`	Top of relocation expression stack.
`result`	The result of the relocation, which is written back into the relocated `r_vaddr` in the object file that the linker is producing.

4.3.4.1 R_ABS

Fields

`r_vaddr`	Number of relocation entries if `s_nreloc` section header field has overflowed. This number includes itself in the count. Otherwise, unused.
`r_symndx`	Unused.
`r_extern`	Unused.
`r_offset`	Unused.

r_size Unused.

Operation

N/A

Restrictions

N/A

Description

This relocation entry is used to indicate a relocation has already been performed or should not be performed. No calculation is associated with such an entry.

The first entry in a relocation section is of type R_ABS if it contains the number of relocation entries in that section (which is the case when the section header field s_nreloc overflows). This type can also be used to pad relocation data or to delete relocation entries in place. In-place deletions of relocation entries are likely to be performed during a partial link.

Example

An object file produced during a partial link has 99993 relocations associated with its .text section. A listing of the entries begins with an R_ABS because the total number overflows s_nreloc:

```
        Vaddr            Symndx Type  Off Size Extern  Name

.text:

    0x0000000000018699       0     ABS           local <null>
```

4.3.4.2 R_REFLONG

Fields

r_vaddr Points to target address.

r_symndx External symbol index if r_extern is 1; section number if r_extern is 0.

r_extern Either 0 or 1.

r_offset Unused.

r_size Unused.

Operation

```
if (r_extern == 0)
    result = (new_scn_addr - old_scn_addr) + (int)[r_vaddr]
else
    result = EXTR.asym.value + (int)[r_vaddr]
```

Restrictions

Result after relocation must not overflow 32 bits.

Description

A relocation entry of this type describes a simple address adjustment to the 32-bit value pointed to by r_vaddr. R_REFLONG entries are most likely to occur when the compilation option -xtaso_short is specified.

The relocated value may be unaligned.

Example 1

C code fragment:

```
extern int i;
void *p = (void *)(&i + 1);
```

Compile as follows:

```
$ cc -c -xtaso_short pgmname.c
```

Produces the following R_REFLONG entry:

```
                    ***RELOCATION INFORMATION***
          Vaddr     Symndx   Type  Off Size Extern  Name

.sdata:
    0x0000000000000000       0 REFLONG          extern  I
```

This relocation entry is necessary because the value of the pointer p depends on the address of the global (common storage class) symbol i, whose address is yet to be determined. At the location indicated by s_vaddr, the value 4 is stored, which will be added to the resolved address of i. The "4" represents the 4 bytes to the next integer storage location in memory after i's.

Example 2

From assembly code, the following declaration produces the same relocation entry as the previous example.

```
.long   I
```

4.3.4.3 R_REFQUAD

Fields

r_vaddr	Points to target address.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    result = (new_scn_addr - old_scn_addr) + (long)[r_vaddr]
else
    result = EXTR.asym.value + (long)[r_vaddr]
```

Restrictions

None.

Description

A relocation entry of this type describes a simple address adjustment to the 64-bit value pointed to by r_vaddr. R_REFQUAD entries are most likely to occur in data sections and almost always are used for relocation of the .lita section.

The relocated value may be unaligned.

Example 1

Small program:

```
#include <stdio.h>

main(){
    printf("printing!\n");
}
```

Relocation entries produced for its .lita section:

```
                    ***RELOCATION INFORMATION***
          Vaddr     Symndx   Type  Off Size Extern  Name

.lita:
    0x0000000000000070       1 REFQUAD          extern  printf
    0x0000000000000078       3 REFQUAD          local   .data
```

The .lita section consists of two entries, and each is relocated. One entry is external, tracking the routine name printf(), and one local, tracking the address of the string literal in the .data section.

Example 2

A R_REFQUAD entry can also be produced by an assembly language statement such as:

```
        .globl y
        .data
b:      .quad y
```

Relocation entry produced:

```
                    ***RELOCATION INFORMATION***
          Vaddr     Symndx   Type  Off Size Extern  Name

.data:
    0x0000000000000000       0 REFQUAD          extern  y
```

The variable b is allocated at s_vaddr in the .data section and will be updated by adding the address of y when the symbol y is resolved.

4.3.4.4 R_GPREL32

Fields

r_vaddr	Points to a 32-bit GP-relative value.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    result = (new_scn_addr - old_scn_addr) + old_GP - GP +
             SEXT((int)[r_vaddr]
else
    result = EXTR.asym.value - GP + SEXT((int)[r_vaddr]
```

Restrictions

Signed result after relocation must not overflow 32 bits.

Description

A relocation entry of this type indicates a 32-bit GP-relative value that must be updated. If it is a local entry, this value must be biased by the GP value for the input object file. In both cases, the current GP value is subtracted to produce a result that is an offset from the GP.

Example 1

Local R_GPREL32 entries are produced for a many-case switch statement. For example, consider the following C program:

```
main(){
    int i;

    scanf("%d",&i);
    switch(i) {
        case 0:i++; break;
        case 1:i--; break;
        case 2:i+=2; break;
        case 3:i-=2; break;
        case 4:i+=3; break;
        case 5:i-=3; break;
        case 6:i++; break;
        default: i=0;
    }
}
```

A compiler may implement a switch statement with a "jump table", that is a code sequence containing labels for each case and a jump statement selecting between them. For each case label, a relocation entry is produced:

```
          Vaddr           Symndx Type  Off Size Extern  Name

.rconst:
    0x00000000000000d0       1 GPREL32          local   .text
    0x00000000000000d4       1 GPREL32          local   .text
    0x00000000000000d8       1 GPREL32          local   .text
    0x00000000000000dc       1 GPREL32          local   .text
    0x00000000000000e0       1 GPREL32          local   .text
    0x00000000000000e4       1 GPREL32          local   .text
    0x00000000000000e8       1 GPREL32          local   .text
```

Example 2

The following assembly code sequence also produces a R_GPREL32 entry:

```
        .globl z
        .data
a:      .gprel32 z
```

Relocation entry produced:

```
                     ***RELOCATION INFORMATION***
          Vaddr      Symndx   Type  Off Size Extern  Name

gprel32.o:

.data:
    0x0000000000000000       0 GPREL32          extern  z
```

4.3.4.5 R_LITERAL

Fields

r_vaddr	Points to a load instruction in the text segment. The value to be relocated is the memory displacement from the $gp in the instruction.
r_symndx	R_SN_LITA
r_extern	Must be zero; all R_LITERAL entries are local.

r_offset Unused.

r_size Unused.

Operation

```
result = (new_scn_addr - old_scn_addr) + (SEXT((short)[r_vaddr]) +
         old_GP) - GP
```

Restrictions

The result after relocation for an R_LITERAL entry must not overflow 16 bits. .

R_LITERAL entries must be local and relative to the .lita section.

Description

A relocation entry of this type is produced when an instruction attempts to reference values in the literal-address pool (.lita section). The instruction containing the reference accesses a .lita entry using the GP value in effect and a signed 16-bit constant. The original address of the item has to be reconstructed and then adjusted for the new location of the address table. The new address then has to be reconverted into a GP displacement using the new GP value.

An R_LITERAL entry may or may not be followed by corresponding R_LITUSE entries. The R_LITERAL entry is required but the R_LITUSE entries are not.

Example

R_LITERAL entries are used when an address is loaded from the literal address pool:

```
ldq     t12, -32664(gp)
```

Relocation entry produced:

```
                    ***RELOCATION INFORMATION***
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:

    0x0000000000000038      13 LITERAL          local   .lita
```

4.3.4.6 R_LITUSE: R_LU_BASE

Fields

r_vaddr Points to memory-format instruction.

r_symndx R_LU_BASE

r_extern Must be zero; all R_LITUSE entries are local.

r_offset Unused.

r_size Unused.

Operation

Check if displacement is within 16 or 32 bits. The displacement is calculated:

```
new_lit = [relocated literal belonging to corresponding R_LITERAL]
disp = new_lit + lituse_disp - GP
```

Restrictions

A relocation entry of this type must follow either an R_LITERAL or another R_LITUSE entry with no other types intervening.

r_vaddr must be aligned on a byte boundary.

Ignored if optimization level is not at least -O1.

Cannot remove the first load instruction unless this is the only corresponding R_LITUSE entry.

Description

This relocation entry is informational and indicates that the base register of the indicated instruction holds a literal address. Note that a R_LITERAL entry, corresponding to an ldq instruction, precedes this entry.

Possible optimizations depend on the distance of the memory displacement from the GP value. If the displacement is less than 16 bits from the GP, a single instruction suffices to describe the location. The code sequence can be changed as shown:

```
ldq     rx, disp(gp)    R_LITERAL
ldq/stq ry, disp2(rx)   R_LITUSE(R_LU_BASE)
                --
ldq/stq ry, disp3(gp)
```

The linker converts the R_LITUSE entry to an R_IMMED_GP_16 for the transformed instructions.

If the displacement is within 32 bits of the GP, one memory access can be saved by replacing the first load instruction with the faster ldah instruction.

```
ldq     rx, disp(gp)    R_LITERAL
ldq/stq ry, disp2(rx)   R_LITUSE(R_LU_BASE)
                --
ldah    rx, disp3(gp)
ldq/stq ry, disp4(rx)
```

The linker will convert the R_LITERAL and the R_LITUSE, respectively, to entries of type R_IMMED_GP_HI32 and R_IMMED_GPLOW32.

This can currently only be done if exactly one R_LITUSE exists for the R_LITERAL.

Example 1

The following instructions represent a single use of an address literal:

```
0x100: ldq     a1, -32656(gp)   // R_LITERAL
0x104: lda     a1, 32(a1)       // R_LU_BASE
```

Relocation entries produced:

```
                        ***RELOCATION INFORMATION***
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:

    0x0000000000000100      13 LITERAL          local   .lita
    0x0000000000000104       1  LITUSE          local   R_LU_BASE
```

The potential optimization indicated by this R_LU_BASE is that the two instructions could possibly be replaced by a single ldq instruction of the form:

```
ldq a1, <disp>(gp)
```

Example 2

The following instructions illustrate multiple R_LITUSE entries following an R_LITERAL entry:

```
0x130:      ldq     t0, -32736(gp)      // R_LITERAL
0x134:      ldq     t1, 0(t0)           // R_LU_BASE
```

```
0x138:      zap     t1, 0x2, t1
0x13c:      insbl   v0, 0x1, v0
0x140:      bis     t1, v0, t1
0x144:      stq     t1, 0(t0)           // R_LU_BASE
```

Relocation entries produced are:

```
                    ***RELOCATION INFORMATION***
         Vaddr      Symndx   Type  Off Size Extern  Name

    0x0000000000000130      13 LITERAL          local   .lita
    0x0000000000000134       1  LITUSE          local   R_LU_BASE
    0x0000000000000144       1  LITUSE          local   R_LU_BASE
```

4.3.4.7 R_LITUSE: R_LU_JSR

Fields

r_vaddr	Points to jump instruction (in text segment).
r_symndx	R_LU_JSR
r_extern	Must be zero; all R_LITUSE entries are local.
r_offset	Unused.
r_size	Unused.

Operation

```
new_lit = [relocated literal belonging to correponding R_LITERAL]
this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
branch_disp = prologue_size + new_lit - this_new_addr + 4
result = branch_disp / 4
```

Restrictions

Must follow either an R_LITERAL or another R_LITUSE entry with no other types intervening.

Result after relocation must not overflow 21 bits (size of branch displacement field in the branch instruction format).

Description

A relocation entry of this type is informational only. It informs the linker that the indicated jump instruction is jumping to an address previously loaded out of the literal address pool. The load instruction had an associated R_LITERAL entry that precedes this relocation entry.

Under the right circumstances, the linker can optimize this sequence in several ways:

- The procedure prologue can be skipped if it is not needed to load a GP value for the procedure.
- The branch can be calculated and the instruction changed to a branch instruction.
- The preceding ldq can be removed.

The first two actions may be performed but not the last if other R_LITUSE entries correspond to the same R_LITERAL. These optimization are performed by the linker for optimization level 1 and greater. In order to preserve preemptibility of symbol references, this optimization can only be done for non-weak global symbols

in a static and dynamic executable. References to static or hidden symbols can be optimized in executables or shared libraries.

Example

The following instructions illustrate the use of a literal as the target of a jump instruction:

```
0x8:   ldq     t12, -32736(gp)  // R_LITERAL
0xc:   lda     sp,  -16(sp)
0x10:  stq     ra,  0(sp)
0x14:  jsr     ra,  (t12)       // R_LU_JSR
```

Relocation entries produced:

```
                      ***RELOCATION INFORMATION***
         Vaddr        Symndx   Type  Off Size Extern  Name

.text:

    0x0000000000000008      13 LITERAL          local   .lita
    0x0000000000000014       3  LITUSE          local   R_LU_JSR
```

The instructions identified by the R_LITERAL and R_LU_JSR entries in this example can be optimized. The ldq instruction can be replaced with a NOP instruction and the jsr can be replaced with a bsr yielding:

```
0x1200011a8:   ldq_u   zero, 0(sp)      // NOP
0x1200011ac:   lda     sp,  -16(sp)
0x120001110:   stq     ra,  0(sp)
0x120001114:   bsr     ra,  0x1200011d8
```

4.3.4.8 R_GPDISP

Fields

r_vaddr	Points to the first of a pair of instructions: lda and ldah. Either instruction may occur first.
r_symndx	Contains the unsigned byte offset from the instruction indicated in r_vaddr to the other instruction used to load the GP value.
r_extern	Must be zero; all R_GPDISP entries are local.
r_offset	Unused.
r_size	Unused.

Operation

```
result = (old_GP - GP) + (this_old_scn_addr - this_new_scn_addr)
       + (65536 * high_disp) + low_disp
```

The result after relocation is written back into the instruction pair.

```
lda_disp = result
ldah_disp = (result + 32768) / 65536
```

Restrictions

Must be a local relocation.

Must describe an lda/ldah instruction pair.

Result after relocation must not overflow 32 bits.

Description

A relocation entry of this type corresponds to two instructions in the code. The field r_vaddr points to one instruction and the address of the other is computed by adding the value of r_symndx to r_vaddr. This relocation entry occurs for each instruction sequence that loads the GP value. For instance, procedure entry points typically include instructions which load their effective GP value. They are normally the first instructions in a procedure's prologue.

Example

A simple example of an occurrence of the R_GPDISP entry is the program entry point:

```
main() {
}
```

Instructions generated:

```
0x0:    ldah    gp, 1(t12)      // R_GPDISP (r_vaddr)
0x4:    lda     gp, -32704(gp)  // R_GPDISP (r_vaddr + r_symndx)
```

Relocation entry produced:

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:
    0x0000000000000000       4  GPDISP          local
```

There are situations where a procedure is called but the R_GPDISP entry is not required. In this case, the gp_used field of the procedure's descriptor will be zero, and an R_LU_JSR optimization may cause the prologue to be skipped. See the *Calling Standard for Alpha Systems* for details on when a procedure requires calculation of a GP value.

4.3.4.9 R_BRADDR

Fields

r_vaddr	Points to a branch instruction.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result =  ((new_scn_addr - old_scn_addr) +
               (branch_displacement * 4)
               + r_vaddr + 4 - this_new_addr) / 4
else
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result = (EXTR.asym.value + (branch_displacement * 4)
              - this_new_addr) / 4
```

Restrictions

After relocation the result should be aligned on a 4-byte boundary.

The signed result must not overflow the 21-bit branch displacement field.

Description

A relocation entry of this type identifies a branch instruction in the code. The branch displacement is treated as a longword (32-bit, or one instruction) offset. The branch target's virtual address is computed:

```
va <- PC + (4 * branch_displacement)
```

The branch displacement must be relocated.

The R_BRADDR relocation can only be used for local or static references because the displacement is fixed at link time. Updating it at run time would require writing to the text segment, which is not permitted. Without the ability to update at run time, symbol preemption for shared objects will not function.

Example

A relocation of this type is used for a call of a static procedure:

```
static bar(){
    int q =1;
    printf ("the value of q is %d\n", q);
}

main (){
    bar();
}
```

Instruction generated:

```
0x4c:       bsr     ra, 0x8(zero)    // R_BRADDR
```

Relocation entry produced:

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:

    0x000000000000004c       1  BRADDR          local   .text
```

4.3.4.10 R_HINT

Fields

r_vaddr	Points to jump-format instruction.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result = ((new_scn_addr - old_scn_addr) + (jump_disp * 4) +
              r_vaddr + 4 - this_new_addr) / 4
else
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result = (EXTR.asym.value + (jump_displacement * 4) -
             this_new_addr) / 4
```

Restrictions

Result after relocation should be aligned on a 4-byte (instruction-size) boundary.

Description

Jump instructions are memory-format instructions where the 14 bits of the displacement field serve as a hint for determining the jump target. The hint is PC-relative and must be relocated to remain relevant. Note that the use of hints is for optimization purposes only and takes advantage of branch-prediction logic built into the architecture. If the hint values were not relocated, a correct executable program would still be produced but potential performance improvements would be lost.

A characteristic of R_HINT entry processing is that instead of checking for overflow of the 14-bit result after relocation, the linker truncates the result and writes it back without issuing an error or warning.

Example

Subroutine calls often cause R_HINT entries.

```
main() {
    printf("hello\n");
}
```

Instructions generated:

```
0x14:      ldq     t12, -32752(gp)     // R_LITERAL
0x18:      jsr     ra, (t12), printf   // R_HINT
```

Relocation entries produced:

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:

    0x0000000000000018       3  LITUSE          local   R_LU_JSR
    0x0000000000000018       0    HINT          extern  printf
```

Note that the same source line and corresponding instruction produce a second relocation entry of type R_LITUSE_JSR. This second entry is also informational only. It indicates that the target register of the jump instruction contains a previously loaded literal address.

4.3.4.11 R_SREL16

Fields

r_vaddr	Points to a 16-bit self-relative value.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result =  (new_scn_addr - old_scn_addr) +
              SEXT((short)[r_vaddr]) + r_vaddr - this_new_addr
else
```

```
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result = EXTR.asym.value - this_new_addr
```

Restrictions

The result after relocation must not overflow 16 bits.

Description

A relocation entry of this type is identical to an R_SREL32 entry except for the size of the value being adjusted.

Example

This type is currently not used by the compilation system.

4.3.4.12 R_SREL32

Fields

r_vaddr	Points to a 32-bit self-relative value.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result =  (new_scn_addr - old_scn_addr)
             + SEXT((int)[r_vaddr]) + r_vaddr - this_new_addr
else
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result = EXTR.asym.value - this_new_addr
```

Restrictions

The result after relocation must not overflow 32 bits.

Description

A relocation entry of this type indicates a value that describes a reference as an offset to its own location. In other words, the target address is computed by adding the contents of the relocation address ([r_vaddr]) to the address of the relocation (r_vaddr). To perform this relocation, the new location of r_vaddr must be computed and subtracted from the new target address to provide the correctly adjusted self-relative, offset which is then written back into the raw data.

Example

The code range descriptors that are generated for each object contain a 32-bit self-relative offset in the rpd_offset field. See Section 3.2.1. The rpd_offset field contains an offset to the associated run-time procedure descriptor in the .xdata section. The R_SREL32 entry identifies this value.

```
main(){
    printf("Printing\n");
}
```

Relocation entry produced:

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.pdata:

    0x0000000000000054      10  SREL32          local   .xdata
```

Note that this relationship between the .xdata and .pdata sections imposes a restriction on the distance between the text and data segments. The run-time procedures in the .xdata section must be within reach of a 32-bit signed offset from the code range descriptors in .pdata.

4.3.4.13 R_SREL64

Fields

r_vaddr	Points to a 64-bit self-relative value.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result = (new_scn_addr - old_scn_addr) + (long)[r_vaddr]
             + r_vaddr - this_new_addr
else
    this_new_addr = r_vaddr - this_old_scn_addr + this_new_scn_addr
    result = EXTR.asym.value - this_new_addr
```

Restrictions

None.

Description

A relocation entry of this type is identical to an R_SREL32 entry except for the size of the value being adjusted.

Example

This type is currently not used by the compilation system.

4.3.4.14 R_OP_PUSH

Fields

r_vaddr	0 if r_extern is 1; an unsigned offset within a section if r_extern is 0.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.

r_offset — Unused.

r_size — Unused.

Operation

```
if (r_extern == 0)
    stack[++tos] = (new_scn_addr - old_scn_addr) + r_vaddr
else
    stack[++tos] = EXTR.asym.value
```

Restrictions

This relocation entry must be followed by an R_OP_STORE entry, with one or more R_OP_PSUB or R_OP_PRSHIFT entries in between.

Stack can hold a maximum of 20 entries.

Description

A relocation entry of this type causes a value to be pushed onto the relocation stack. The value is generally the target address of the relocation, which will be adjusted using subsequent R_OP_PSUB and R_OP_PRSHIFT relocation calculations.

Example

A code range descriptor in the .pdata section contains a 32-bit field, begin_address, which is the offset of the associated code range address from the beginning of the code range descriptor table. See Section 3.2.1. This value is calculated by subtracting two addresses and storing the least significant 32 bits. A series of three stack relocation entries is used to represent this offset calculation.

```
main(){
    foo();
}
foo(){
    printf("Printing\n");
}
```

Relocation entries produced for use in calculating the begin_address in the code range descriptor for foo():

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.pdata:

    0x0000000000000030       1    PUSH          local    .text
    0x0000000000000000       3    PSUB          extern   _fpdata
    0x0000000000000078      11   STORE  0   32  local    .pdata
```

The following series of relocation entries will effectively perform the calculation:

```
(.pdata+0x78)  =  (long)(((.text+0x30)-&_fpdata) & 0xffffffff)
```

4.3.4.15 R_OP_STORE

Fields

r_vaddr — Location to store calculated bit field.

r_symndx — Section index of containing section.

r_extern — Must be 0.

r_offset — Bit offset from r_vaddr. (Bit 0 is the least significant bit in little-endian objects and the most significant bit in big-endian objects. See Section 1.7.)

r_size — Number of bits to store.

Operation

```
if (little_endian)
    rshift = r_offset
else
    rshift = 64 - (r_offset + r_size)
bitfield = ((long)[r_vaddr] >> r_offset) & ((1 << r_size) - 1)
bitfield <- stack[tos--]
```

Restrictions

Stack cannot be empty.

Description

A relocation entry of this type causes the value currently on the top of the relocation stack to be written into a bit field specified by the entry. The bit field is described using a bit position and size in bits. Note that bit numbering is reversed in a big-endian representation.

Example

An example of the R_OP_STORE entry is given in Section 4.3.4.14.

4.3.4.16 R_OP_PSUB

Fields

r_vaddr — 0 if r_extern is 1; an unsigned offset within a section if r_extern is 0.

r_symndx — External symbol index if r_extern is 1; section number if r_extern is 0.

r_extern — Either 0 or 1.

r_offset — Unused.

r_size — Unused.

Operation

```
if (r_extern == 0)
    result = (new_scn_addr - old_scn_addr) + r_vaddr
    stack[tos] = stack[tos] - result
else
    result = EXTR.asym.value
    stack[tos] = stack[tos] - result
```

Restrictions

The relocation stack cannot be empty. This entry must fall somewhere between an R_OP_PUSH entry and an R_OP_STORE entry.

Description

A relocation entry of this type causes the value at the top of the relocation expression stack to be popped, adjusted by subtracting the address described by r_extern and r_symndx, and pushed back on the stack.

Example

An example of the R_OP_STORE entry is given in Section 4.3.4.14.

4.3.4.17 R_OP_PRSHIFT

Fields

r_vaddr	0 if r_extern is 1; an unsigned offset within a section if r_extern is 0.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
if (r_extern == 0)
    result = (new_scn_addr - old_scn_addr) + r_vaddr
    stack[tos] = stack[tos] >> result
else
    result = EXTR.asym.value
    stack[tos] = stack[tos] >> result
```

Restrictions

The stack cannot be empty. So this entry must fall somewhere between an R_OP_PUSH and an R_OP_STORE.

Description

A relocation entry of this type causes the value at the top of the relocation expression stack to be popped, adjusted by right shifting the value by the number of bits described by r_extern and r_symndx, and pushed back on the stack.

Example

This relocation type can be used to convert a byte offset into an instruction offset. Right shifting a byte offset by two bits will produce an instruction offset because Alpha instructions are 4 bytes wide.

The following assembly code will result in an R_HINT entry for the 14-bit instruction offset contained in the hint field of a jsr instruction. See Section 4.3.4.10 for a description of the R_HINT entry.

```
0x3c    ldq       t12, -32752(gp)     /* &printf */
0x40    jsr       ra, (t12)
```

The R_HINT entry for the instruction at 0x40 could also be accomplished with a series of stack relocation options:

```
.text:

    0x0000000000000000       2    PUSH          extern   printf
    0x0000000000000044       1    PSUB          local    .text
    0x0000000000000002      14 PRSHIFT          local    .abs
    0x0000000000000040       1   STORE  0   14  local    .text
```

4.3.4.18 R_GPVALUE

Fields

r_vaddr	Starting virtual address for new GP value.

r_symndx	Constant that is added to the GP value in the a.out header to obtain the new GP value.
r_extern	Must be zero; all R_GPVALUE entries are local.
r_offset	Unused.
r_size	Unused.

Operation

```
new GP = AOUTHDR.gp_value + r_symndx
```

Restrictions

This type of relocation entry cannot be external.

Description

A relocation entry of this type identifies the position in the code where a new GP value takes effect. R_GPVALUE entries are inserted by the linker during partial links.

Example

A linked program that references 20,000 external symbols will have at least 3 GOT entries with 3 corresponding GP values. See Section 2.3.4. If the program has GP-relative relocation entries in both .text and .rdata sections, two R_GPVALUE entries would be reported for each of these sections.

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:
    0x0000000010084cf0   64000 GPVALUE          local
    0x00000000100cb190  111984 GPVALUE          local
.rdata:
    0x0000000001000fa00   64000 GPVALUE          local
    0x0000000001001b570  111984 GPVALUE          local
```

4.3.4.19 R_GPRELHIGH

Fields

r_vaddr	Points to a memory format instruction (ldah).
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

See R_GPRELLOW relocation type.

Restrictions

Must be followed by at least one R_GPRELLOW.

Relocated result must not overflow unsigned 32-bit range.

Description

A relocation entry of this type is invalid unless it is followed by at least one R_GPRELLOW entry. When an R_GPRELHIGH entry is encountered, no calculation is performed. The relocation calculation is deferred until the R_GPRELLOW entry is processed. See the R_GPRELLOW description for more information.

Example

See R_GPRELLOW.

4.3.4.20 R_GPRELLOW

Fields

r_vaddr	Points to memory format instruction (ld* or st*).
r_symndx	Must match R_GPRELHIGH.
r_extern	Must match R_GPRELHIGH.
r_offset	Unused.
r_size	Unused.

Operation

```
low_disp = [r_vaddr].displacement
high_disp = [R_GPRELHIGH->r_vaddr].displacement
displacement = high_disp * 65536 + low_disp

if (r_extern = 0)
    result = displacement + (new_scn_addr - old_scn_addr) +
             (old_GP - GP)
else
    result = displacement + EXTR.asym.value + (old_GP - GP)

[R_GPRELHIGH->r_vaddr].displacement = (result+32768) >> 16
[r_vaddr].displacement = result & 0xFFFF
```

Restrictions

The R_GPRELHIGH/R_GPRELLOW relocations must be used as a pair or set. At least one R_GPRELLOW entry follows each R_GPRELHIGH entry.

After relocation, the result must not overflow 32 bits.

The memory displacement for all R_GPRELLOW entries corresponding to the same R_GPRELHIGH must match.

Description

The R_GPRELHIGH/R_GPRELLOW entry pair is used to describe GP-relative memory accesses. The R_GPRELHIGH entry indicates an ldah instruction. The R_GPRELLOW entry (or entries) indicates a load or store instruction. If multiple R_GPRELLOW entries are associated with an R_GPRELHIGH, they must all describe the same memory location. A relocatable address can be formed with the following computation:

```
addr = 65536 * high_disp + SEXT (low_disp)
```

To relocate this code sequence, the memory displacement fields in each instruction must be adjusted to reflect changes in the target address they compute and in the GP value.

The reason these entries are treated as a pair is that sign extension of the low instruction's displacement field can result in an off-by-one error that must be fixed by adding one to the high instruction's displacement. This situation can only be detected if the instructions are considered together.

These relocation entries describe instructions that are primarily used for computing addresses in kernel code.. The kernel is built without a `.lita` section, and kernel performance is enhanced by code that calculates addresses directly instead of loading addresses from a `.lita` memory location. The code size, on average, is unaffected by the kernel's use of this addressing method.

Example

Use the kernel build option `-Wb,-static` to compile the following sample code.

```
static int a;
foo(){
    a++;
}
```

Code generated for loading the address of `"a"`:

```
0x0:        ldah   t0,  0(gp)
0x4:        lda    t0, 16(t0)
```

Relocation entries produced are:

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:
    0x0000000000000000       5  GPHIGH          local   .sbss
    0x0000000000000004       5   GPLOW          local   .sbss
```

4.3.4.21 R_IMMED: GP16

Fields

`r_vaddr`	Points to memory-format instruction.
`r_symndx`	External symbol index if `r_extern` is 1; section number if `r_extern` is 0.
`r_extern`	Either 0 or 1.
`r_offset`	Unused.
`r_size`	`R_IMMED_GP_16`.

Operation

N/A

Restrictions

N/A

Description

A relocation entry of this type identifies an instruction that adds a 16-bit displacement to the GP value, obtaining an address. The `r_extern` and `r_symndx`

fields specify the external symbol or section to which the calculated address is relative.

This relocation entry is created by the linker to indicate that an optimization has taken place because the displacement is within 16-bits of the GP value.

Example

N/A

4.3.4.22 R_IMMED: GP_HI32

Fields

r_vaddr	Points to memory-format instruction.
r_symndx	Unused.
r_extern	Unused.
r_offset	Unused.
r_size	R_IMMED_GP_HI32.

Operation

N/A

Restrictions

N/A

Description

A relocation entry of this type identifies an instruction that is part of a pair of instructions that add a 32-bit displacement to the GP value. This instruction adds the high portion of the 32-bit displacement. The next R_IMMED_LO32 entry identifies the instruction containing the low portion of the displacement. More than one subsequent R_IMMED_LO32 entry can share the same R_IMMED_GP_HI32 entry.

Example

N/A

4.3.4.23 R_IMMED: SCN_HI32

Fields

r_vaddr	Points to memory-format instruction.
r_symndx	Unused.
r_extern	Unused.
r_offset	Unused.
r_size	R_IMMED_SCNHI32.

Operation

N/A

Restrictions

N/A

Description

A relocation entry of this type identifies an instruction that is part of a pair of instructions that add a 32-bit displacement to the starting address of the current section. This instruction adds the high portion of the displacement. The next R_IMMED_LO32 entry identifies the instruction with the low portion.

Example

N/A

4.3.4.24 R_IMMED: BR_HI32

Fields

r_vaddr	Points to a memory-format instruction following a branch (br, bsr, jsr, or jmp) instruction.
r_symndx	Specifies a byte offset from r_vaddr to the branch instruction.
r_extern	Unused.
r_offset	Unused.
r_size	R_IMMED_BRHI32.

Operation

N/A

Restrictions

N/A

Description

A relocation entry of this type identifies an instruction that is part of a pair of instructions that add a 32-bit displacement to the address of the instruction following a branch (br, bsr, jsr, or jmp). The branch must precede this instruction. The r_symndx field specifies a byte offset from r_vaddr to the branch instruction. The instruction identified by this relocation entry adds the high portion of the displacement. The next R_IMMED_LO32 entry identifies the instruction with the low portion of the displacement.

Example

N/A

4.3.4.25 R_IMMED: LO32

Fields

r_vaddr	Points to a memory-format instruction.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	R_IMMED_LO32.

Operation

N/A

Restrictions

N/A

Description

A relocation entry of this type identifies an instruction that is part of a pair of instructions that add a 32-bit displacement to a base address. This instruction adds the low portion of the displacement. This relocation entry is combined with the previous R_IMMED_GP_HI32, R_IMMED_SCN_HI32, or R_IMMED_BR_HI32 entry. The r_extern and r_symndx fields specify the external symbol or section to which the calculated address is relative.

Example

N/A

4.3.4.26 R_TLS_LITERAL

Fields

r_vaddr	Points to an instruction that loads the TSD key for initiating a thread local storage reference – actually, not the key itself but key * 8, which gives the offset of the TLS pointer in the TSD array.
r_symndx	R_SN_LITA
r_extern	Must be zero; all R_TLS_LITERAL entries are local.
r_offset	Unused.
r_size	Unused.

Operation

```
result = (new_scn_addr - old_scn_addr) +
         (SEXT((short)[r_vaddr]) +old_GP) - GP
```

Restrictions

The result after relocation for an R_TLS_LITERAL entry must not overflow 16 bits.

R_TLS_LITERAL entries must be local and relative to the .lita section.

Description

A relocation entry of this type is very similar to an R_LITERAL entry. An R_TLS_LITERAL entry identifies an instruction that uses a GP displacement to load an the address of the symbol __tlsoffset from the .lita section.

The value of the __tlsoffset symbol is fixed at run time to be the TSD array offset of the TLS pointer. The symbol can occur anywhere in the GOT or .lita section. The linker-defined symbol __tlskey points to one of the instances of the __tlsoffset symbol.

The linker processes the R_TLS_LITERAL relocation by adjusting the GP offset in the displacement of the target instruction.

Example

Routines that reference TLS addresses will have at least one R_TLS_LITERAL entry for the load of the __tlsoffset value.

```
__declspec(thread) long foo;
main(){
    foo = 2;
}
```

Code generated will include the instruction:

```
0x14:        ldq  at, -32752(gp)
```

Relocation entry produced:

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:
    0x0000000000000014      13 TLSLITE          local   .lita
```

4.3.4.27 R_TLS_HIGH

Fields

r_vaddr	Points to memory-format instruction.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

See R_TLS_LOW description.

Restrictions

Must be followed by R_TLS_LOW entry.

Description

See R_TLS_LOW.

Example

See R_TLS_LOW.

4.3.4.28 R_TLS_LOW

Fields

r_vaddr	Points to memory-format instruction.
r_symndx	External symbol index if r_extern is 1; section number if r_extern is 0.
r_extern	Either 0 or 1.
r_offset	Unused.
r_size	Unused.

Operation

```
low_disp = [r_vaddr].displacement
high_disp = [R_TLS_HIGH->r_vaddr].displacement
displacement = high_disp * 65536 + low_disp

if (r_extern = 0)
    result = displacement + (new_scn_addr - old_scn_addr)
else
    result = displacement + EXTR.asym.value

[R_TLS_HIGH->r_vaddr].displacement = (result+32768) >> 16
[r_vaddr].displacement = result & 0xFFFF
```

Restrictions

External relocation entries of this type are limited to TLS symbols.

Local relocation entries of this type are restricted to the TLS sections .tlsdata and .tlsbss.

The relocated result must not exceed 32 bits.

Description

The linker must handle R_TLS_HIGH and R_TLS_LOW entries as a pair. The pairs of relocation entries must be in sequence starting with R_TLS_HIGH. The order and location of the instructions associated with these relocation entries are not restricted.

Example

The load of a TLS symbol's address requires an R_TLS_HIGH/R_TLS_LOW entry pair.

```
__declspec(thread) long foo;
main(){
    foo = 2;
}
```

Code generated:

```
0x0c:        call_pal  rduniq
0x10:        ldq  v0,  96(v0)
0x14:        ldq  at, -32752(gp)
0x18:        addq v0, at, v0
```

```
0x1c:        ldq  v0, 0(v0)
0x20:        ldah v0, 0(v0)
0x24:        stq  t0, 0(v0)
```

Relocation entries produced:

```
          Vaddr         Symndx   Type  Off Size Extern  Name

.text:
    0x0000000000000020       0 TLSHIGH          extern  foo
    0x0000000000000024       0  TLSLOW          extern  foo
```

4.4 Compact Relocations

Compact relocations are a highly compressed form of relocation records designed for the use of profiling tools and object restructuring tools. By default, they are generated by the linker for all fully linked executable objects and recorded in the object's `.comment` section. The linker produces this information using `libmld.a` APIs, which implement the reading and writing of compact relocations. Compact relocations are not produced for images linked with the following linker options: `-r`, `-s`. The `strip` utility will remove the comment subsection that contains compact relocations. See Chapter 7 for the format of the `.comment` section.

Compact relocations must provide crucial relocation information in much less space than the space required for actual relocation entries. This goal is accomplished by employing a heuristic function to predict relocations. For some sections, this heuristic is highly accurate. Detailing many records in the object file becomes unnecessary because the algorithm can be used instead to recreate many of the actual relocation entries.

Version Note

In releases of Tru64 UNIX prior to **V5.1**, compact relocations contained only enough relocation information to drive tools that restructure an executable's `.text`, `.init`, and `.fini` sections. From Tru64 UNIX **V5.1** onward, executables contain full compact relocation information including relocation records for text and data segment addresses in all mapped object sections.

The interfaces for compact relocations continue to evolve. These interfaces are defined and described in the header file `cmplrs/cmrlc.h`. This section describes the on-disk file format of compact relocations and the producer and consumer algorithms.

4.4.1 Overview

The procedure for creation of compact relocations is as follows:

1. Generate a list of predicted relocations using heuristics.
2. Compare the predicted relocations to the actual relocation entries (which are input data to the compact relocations producer).
3. Wherever a "miss" occurs (that is, the predicted and actual entries do not match) output a compact relocation record.

The procedure for the use of compact relocation records follows:

1. Generate the list of predicted relocations using the same heuristics as the compact relocations producer.
2. Compare the expanded compact relocations data with predicted relocations to reconstruct the actual relocation entries.

See Section 4.4.3 for more details.

4.4.2 File Format

Compact relocations are stored in a subsection of the `.comment` section. The linker and other tools do not need to be aware of the details of the internal structure of the compact relocation subsection. This knowledge is encapsulated in the `cmrlc_*` routines found in `libmld.a`.

The on-disk format of the compact relocations data consists of the following components, in order:

- Version identifier
- Compact relocations file header
- Compact relocations section headers (for each section)
- Compact relocations tables (for each section)
- Expression stack relocations tables (for each section)
- GP value tables (for each section)

Code may only assume that the version and the file header are contiguous. To access other structures, it is necessary to rely on the location information in the file header.

4.4.2.1 Compact Relocations Version

The compact relocation section begins with a version identifier, which has the following structure:

```
struct {
        unsigned int   version_major;
        unsigned int   version_minor;
};
```

SIZE - 8 bytes, ALIGNMENT - 4 bytes

The version identifier allows the format of the compact relocations to change from one release to another while providing a mechanism for tools to work on binaries with either the old or new formats. The version identifiers are separate from the header because the format of the header itself may change from release to release.

The major version identifier is incremented for changes in the format of the compact relocation data that affect the most basic access to the data. For example, changes in structure sizes or structure layout are likely to cause failures in existing code that simply reads the raw compact relocation data.

The minor version identifier is incremented whenever the compact relocation data is modified without impacting the format of the data. For example, changing the heuristic to further compact the stored relocation information would require the minor version identifier to be incremented. If the consumer routines see that an object has an old minor version number, they can call a matching version of the heuristic to correctly reconstruct the relocation information.

The major and minor version identifiers that have been used for compact relocation data are described in Table 4–5. Enumeration values for supported versions can be found in the header file `/usr/include/cmplrs/cmrlc.h`.

Table 4–5: Compact Relocation Version Identifiers

Major	Minor	OS Version	Description
0	0	V3.0	Initial version
1	0	V3.2	Fix for dynsym relocations
2	0	V4.0	Miscellaneous bug fixes
2	3	V5.1	Full compact relocations

4.4.2.2 Compact Relocations File Header

The version identifier is followed by a high-level header structure that stores the sizes and locations of the other tables with compact relocations information:

```
struct cmrlc_file_header {
        /*
         * Total number of elements in each sub-table.
         */
        unsigned long   scn_num;    /* section header table */
        unsigned long   rlc_num;    /* compact relocation table */
        unsigned long   expr_num;   /* expression relocation table */
        unsigned long   gpval_num;  /* GP value table */

        /*
         * Relative file offset from start of compact relocation data
         * to each sub-table.
         */
        unsigned long   scn_off;
        unsigned long   rlc_off;
        unsigned long   expr_off;
        unsigned long   gpval_off;
};
```

SIZE - 64 bytes, ALIGNMENT - 8 bytes

Each of the `*_num` fields indicates the number of entries in the corresponding tables. Each of the `*_off` fields contains a relative file offset from the start of the compact relocations `.comment` subsection to the start of the corresponding table. If any of the tables are not present for a particular program, the `*_num` and `*_off` fields should be set to zero.

4.4.2.3 Compact Relocations Section Header

One or more compact relocations section headers follow the compact relocations file header. Each section header has the following structure:

```
struct cmrlc_file_scnhdr {
        char            name[8];    /* section name */

        /*
         * Number of elements for this section in each sub-table.
         */
        unsigned long  rlc_snum;
        unsigned long  expr_snum;
        unsigned long  gpval_snum;

        /*
         * Index from start of table to this section's elements.
         * (This is an element index, not a byte offset.)
         */
        unsigned long  rlc_indx;
        unsigned long  expr_indx;
        unsigned long  gpval_indx;

        /*
         * Flag: True if compact relocation table is sorted by
         * increasing virtual address.
         */
         unsigned long  rlc_sorted:1;
         unsigned long  :63;
};
```

SIZE - 64 bytes, ALIGNMENT - 8 bytes

One compact relocation section header is created for each eCOFF object file section for which compact relocation data is stored. This section header is unrelated to the eCOFF section header structure except for the name field, which connects the two.

Each of the *_num fields indicates the number of entries in the corresponding table for this object file section. If the *_num field is non-zero, the corresponding *_indx field contains the index of the start of that section's entries within the table.

The rlc_sorted field indicates whether the compact relocation table entries for this section are sorted by virtual address.

If an object file section does not have entries in one of the tables for a particular program, the corresponding fields should be set to zero.

4.4.2.4 Compact Relocations Table

Compact relocation tables follow the compact relocation section headers. Each compact relocation table consists of an array of structures:

```
struct cmrlc_file_rlc {
    unsigned    v_offset;
    union {
        unsigned        word;
        struct {
            unsigned    type:5;
            unsigned    :27;
        } common;
        struct {                /* GPDISP */
            unsigned    type:5;
            unsigned    lda_offset:27;
        } gpdisp;
        struct {                /* EXPRESSION */
            unsigned    type:5;
            unsigned    index:27;
        } expr;
        struct {                /* REF*, SREL*, GPREL32 */
            unsigned    type:5;
            unsigned    rel_scn:5;
            unsigned    count:12;
            unsigned    dist:4;                         (V5.0 - )
            unsigned    :6;
        } addrtype;
        struct {                /* External REF */      (V5.1 - )
            unsigned    type:5;                         (V5.1 - )
            unsigned    r_symndx:27;                    (V5.1 - )
        } eref;                                         (V5.1 - )
        struct {                /* LITERAL */           (V5.1 - )
            unsigned    type:5;                         (V5.1 - )
            unsigned    rel_scn:5;                      (V5.1 - )
            unsigned    count:12;                       (V5.1 - )
            unsigned    dist:4;                         (V5.1 - )
            unsigned    :6;                             (V5.1 - )
        } literal;                                      (V5.1 - )
        struct {                /* LITUSE */            (V5.1 - )
            unsigned    type:5;                         (V5.1 - )
            unsigned    rel_scn:5;                      (V5.1 - )
            unsigned    lit_type:5;                     (V5.1 - )
            unsigned    litOFFSET:17;                   (V5.1 - )
        } lituse;                                       (V5.1 - )
        struct {                /* NO_RELOC, NO_LITUSE */ (V5.0 - )
            unsigned    type:5;                         (V5.0 - )
            unsigned    count:12;                       (V5.0 - )
            unsigned    dist:4;                         (V5.0 - )
            unsigned    :11;                            (V5.0 - )
        } noreloc;                                      (V5.0 - )
        struct {                /* IMMED: GP_HI32, SCN_HI32, BR_HI32 */
            unsigned    type:5;
            unsigned    subop:6;
            unsigned    br_offset:21;
        } immedhi;
        struct {                /* IMMED: all other sub-opcodes */
            unsigned    type:5;
            unsigned    subop:6;
            unsigned    rel_scn:5;
            unsigned    hi_offset:16;                   (V5.1 - )
        } immedlo;
        struct {                /* VADJUST */
```

```
            unsigned    type:5;
            signed      adjust:27;
        } vadjust;
        struct {                /* BRADDR, HINT */
            unsigned    type:5;
            unsigned    rel_scn:5;
            unsigned    :22;
        } other;
    } info;
};
```

SIZE - 8 bytes, ALIGNMENT - 4 bytes

```
/*
 * Values for 'type' field.
 */
enum cmrlc_rlctypes {
    CMRLC_REFLONG=1,
    CMRLC_REFQUAD=2,
    CMRLC_GPREL32=3,
    CMRLC_GPDISP=4,
    CMRLC_BRADDR=5,
    CMRLC_HINT=6,
    CMRLC_SREL16=7,
    CMRLC_SREL32=8,
    CMRLC_SREL64=9,
    CMRLC_EXPRESSION=10,   /* R_OP_* expression */
    CMRLC_IMMEDHI=11,      /* R_IMMED for high part */
    CMRLC_IMMEDLO=12,      /* R_IMMED for low part */
    CMRLC_NO_RELOC=13,     /* correct mispredicted relocation */
    CMRLC_VADJUST=14,      /* adjust base for succeeding 'v_offset's */
    CMRLC_LITERAL=15,                                   (V5.1 - )
    CMRLC_LITUSE=16,                                    (V5.1 - )
    CMRLC_NO_LITUSE=17,                                 (V5.1 - )
    CMRLC_REFQUAD_EXTERN=18  /* external REFQUAD */     (V5.1 - )
};

/*
 * Maximum value for 'count' field in 'addrtype' relocations.
 */
#define CMRLC_COUNT_MAX         ((1<<12) - 1)

/*
 * Maximum value for 'dist' field in 'addrtype' and 'noreloc' relocations.
 */
#define CMRLC_DIST_MAX          ((1<<4) - 1)
```

The number of elements in the array is determined by the corresponding *_num field in the section header.

The v_offset field specifies the virtual address of each relocation entry as a byte offset from a base address. Initially, the base is the starting virtual address of the current section. If relocations are required at addresses that cannot be expressed as a 32-bit offset from the section's start address, CMRLC_VADJUST relocation entries are used to extend the addressing range. However, this feature is not fully supported.

The value of the type field determines how to interpret the remainder of a compact relocation structure.

The lda_offset field specifies an instruction offset (byte offset divided by 4) from the relocation entry's virtual address to the lda instruction in an R_GPDISP entry's ldah/lda pair. This design does not support ldah/lda pairs that are separated by more than 2^29 bytes.

The rel_scn field indicates the ID of the section to which this relocation is relative. It uses the R_SN_* values from the header file reloc.h.

The count and dist fields are used to specify consecutive relocation entries that are identical. The count field can be used in this manner for R_REFLONG, R_REFQUAD, R_SREL16, R_SREL32, R_SREL64, R_GPREL32, and R_LITERAL entries. Two relocation entries are identical if they have the same type and relative section. Two relocation entries are consecutive if the difference in their virtual addresses is equal to the same multiple of the natural size for the relocation

type (16 bits for R_SREL16; 32 bits for R_REFLONG, R_SREL32, R_GPREL32; and R_LITERAL, and 64 bits for R_REFQUAD and R_SREL64). The dist field multiplied by the natural size of the relocation type gives the byte distance between repetitions of the relocation. A count value of zero is not allowed. These fields reduce the impact of mispredicting the relocations for jump tables.

4.4.2.5 Stack Relocation Table

Expression stack relocation information is stored separately. Each stack relocation table entry has the following structure:

```
struct cmrlc_file_expr {
        unsigned long  vaddr;
        unsigned       type:5;
        unsigned       rel_scn:5;
        unsigned       offset:6;  /* CMRLC_EXPR_STORE only */
        unsigned       size:6;    /* CMRLC_EXPR_STORE only */
        unsigned       last:1;    /* true for last reloc in expr */
        unsigned       :9;
        unsigned       reserved;
};
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

```
/*
 * Values for 'type' field.
 */
enum cmrlc_exprtypes {
        CMRLC_EXPR_PUSH=1,     /* R_OP_PUSH */
        CMRLC_EXPR_PSUB=2,     /* R_OP_PSUB */
        CMRLC_EXPR_PRSHIFT=3,  /* R_OP_PRSHIFT */
        CMRLC_EXPR_STORE=4     /* R_OP_STORE */
};
```

Expression stack compact relocation records are stored in a separate table because each record requires more space than other types of compact relocation records. Entries in this table are grouped into sequences of relocation entries that form a single expression. The first entry in each table starts a sequence. The last entry in each sequence has its last field set to one. A new sequence starts immediately after the end of the previous sequence.

The start of each sequence is referenced by a CMRLC_EXPRESSION entry in the section's compact relocation table. The index field of that entry points to the first entry in a stack relocation sequence. All sequences in the stack relocation table should have a corresponding CMRLC_EXPRESSION entry in the compact relocation table.

4.4.2.6 GP Value Tables

Additional tables called GP value tables are used to store GP range information. GP values are kept in tables separate from other compact relocations to reduce the processing required to map a virtual address to the corresponding active GP value.

Each GP value table consists of an array of these structures:

```
struct {
        unsigned long     vaddr
        unsigned          gp_offset
        unsigned          reserved
};
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

Each additional GP range after the first range has an entry in the table. (The first range is described by the GP value in the file's a.out header.) Therefore, a single-GOT program will have no entries in its GP value tables.

If an executable's sections have different numbers of GP ranges, gpval_num should be set to describe the section with the largest number of ranges. eCOFF sections with fewer GP ranges must still have GP value tables with gpval_num entries.

Sections with short GP value tables can duplicate their last GP value table entry until the table is the proper length.

The vaddr field contains the virtual address where the new range starts. vaddr must point within the section to which this GP value table corresponds. The new GP value is computed by adding gp_offset to the GP value in the file's a.out header.

4.4.3 Basic Algorithm for Compact Relocations Production

In order to produce compact relocations, a tool must have a set of actual relocation entries and the raw data to which those relocation entries apply. It should then apply the following algorithm to create a set of matching compact relocations:

1. Convert the external relocation entries to local relocation entries.
2. Run the prediction heuristic function to construct a set of predicted relocation entries from the raw data.
3. Compare the predicted relocation entries to the remaining actual relocation entries and create a compact relocation record for any mismatches.
4. Compress any sequences of consecutive, identical R_REF*, R_SREL*, R_GPREL32, or R_LITERAL entries.
5. Set the rlc_sorted field if the compact relocation entries are stored in a sorted order.

Any R_GPVALUE entries must be handled specially. These relocation entries must be added to their section's GP value table. They should then be removed from the list of actual relocation entries used to create compact relocations.

The first step in the algorithm is to convert actual relocation entries from external to local. The compact relocations only exist in fully linked executables with no undefined symbols. Thus, external relocation entries are not usually needed. (The compact relocation types include a type for retaining external R_REFQUAD relocations wherever symbol correspondence might be needed for post-link processing.) An external relocation entry is converted to a local relocation entry by setting its r_extern field to zero and changing its r_symndx field to the appropriate relocation section constant (see Table 4–1).

The second step is to run the prediction heuristic function over the raw data for which these actual relocation entries apply. This produces a set of predicted relocation entries.

Step three compares the predicted relocation entries to the actual relocation entries as follows:

a. If a match exists between a predicted relocation entry and an actual relocation entry at the same virtual address, do nothing.

b. If a predicted relocation entry and an actual relocation entry at the same virtual address do not match, write a compact form of the actual relocation entry to the compact relocation data file.

c. If only a predicted relocation entry exists for a particular virtual address, write a compact CMRLC_NO_RELOC record to the data file at this virtual address.

d. If only an actual relocation entry exists for a particular virtual address, write a compact form of the actual relocation entry to the compact relocation data file.

Creating a compact relocation entry from an actual relocation entry is fairly straightforward except in the case of an expression stack relocation sequence. First, create entries in the stack relocation table for each relocation entry in the sequence. Normally, this sequence starts with an R_OP_PUSH entry and ends with

an R_OP_STORE entry. The last entry should have the last field set to one. Then create a CMRLC_EXPRESSION compact relocation entry whose index field points to the first entry in the stack relocation table for this expression. (This can only be done for a sequence that describes a complete expression.)

The fourth step is to compress any sequences of R_REF*, R_SREL*, R_GPREL32, or R_LITERAL entries that are consecutive and identical . Such a sequence exists if all relocation entries in the sequence have the same relocation type, are relative to the same rel_scn value (R_SN_* constant), and have v_offset fields that increase by a multiple of the natural size of the relocation type (for example, 8 bytes for R_REFQUAD, 2 bytes for R_SREL16). Such sequences can be replaced with a single compact relocation entry that has the sequence's type and rel_scn value. The v_offset field should be that of the first relocation entry in the sequence. The dist field should be set to the distance between repeated relocations in natural size increments, and the count field should be set to the number of relocation entries in the sequence.

The final step is to set the rlc_sorted field in the compact relocation section header. If the compact relocations are stored in order of increasing v_offset values, this field should be set to one. Otherwise, it should be set to zero.

4.4.4 Basic Algorithm for Compact Relocations Consumption

A consumer tool can read back the compact relocation entries if it has the compact relocation information and the raw data that they describe. The consumer tool can use this information to regenerate the actual relocation entries by following this algorithm:

1. Expand any R_REF*, R_SREL*, R_GPREL32, or R_LITERAL compact relocation entries whose count field is greater than one.
2. Run the prediction heuristic function to construct a set of predicted relocation entries from the raw data.
3. Compare the predicted relocation entries to the compact relocation entries and reconstruct the actual relocation entries.

The first step in this algorithm just undoes the compression step (step four) in the production algorithm.

The second step runs the same prediction heuristic that was used in the production algorithm. To guarantee that the generated predicted relocation entries are the same as when the compact relocation entries were produced, it is critical that the heuristic function is the same. It is also critical that the raw data is the same as when the compact relocation entries were produced.

The final step compares the predicted relocation entries with the stored compact relocation entries as follows:

1. If only a predicted relocation entry exists for a particular virtual address, report the predicted relocation entry.
2. If a CMRLC_NO_RELOC entry exists at the same virtual address as a predicted relocation entry, do not report a relocation entry at this virtual address.
3. If a compact relocation entry other than CMRLC_NO_RELOC exists at the same virtual address as a predicted relocation entry, report the compact relocation entry.
4. If only a compact relocation entry exists for a particular virtual address, report the compact relocation entry.

4.5 Linkerdef Relocations

Version Note

Linkerdef relocations are supported in Tru64 UNIX **V5.1** and greater for symbol table format **V3.13** and greater.

Linkerdef relocations are generated by the linker for all fully linked executable objects and shared libraries. They are not produced for images linked with the following linker options: `-r`, `-s`. The `strip` utility will remove the comment subsection that contains linkerdef relocations. See Chapter 7 for the format of the `.comment` section.

The linkerdef relocations supplement compact relocation information. They provide relocation information for all uses of linker-defined symbol values within the section data of an object. This information is not currently accessible in compact relocation information. Compact relocations are generally stored as local relocations with no symbolic information. Linkerdef relocations are also unique because they contain relocations for absolute symbols with literal values such as `_DYNAMIC_LINK` and `_procedure_table_size`.

Tools that modify linked objects, such as `om` and `spike`, can use linkerdef relocations to update references to linker-defined symbol values that are necessarily changed as a result of other changes made to the linked object.

4.6 Language-Specific Relocations Features

Relocation entries may be generated for language-specific compiler-generated external symbols. For example, they are often generated in Fortran programs for the procedure `for_set_reentrancy()` and in C++ programs for exception-handling labels.

5

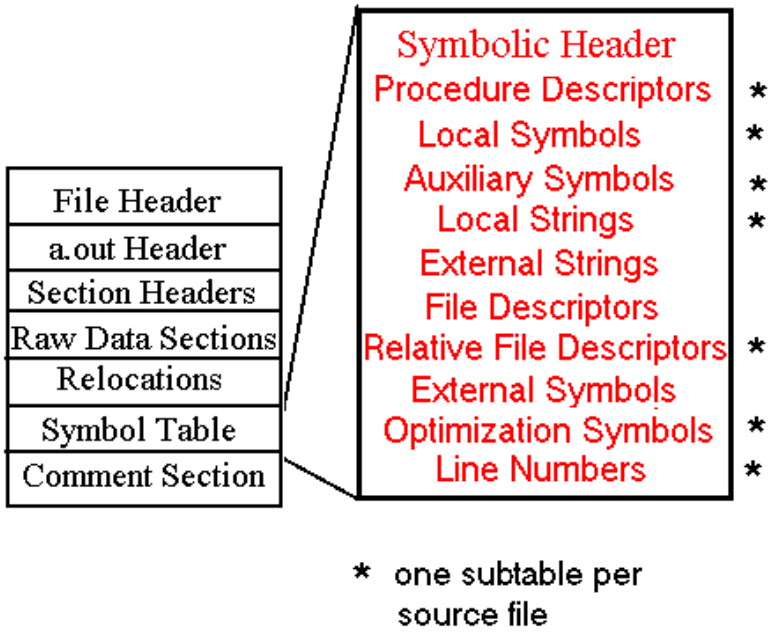
Symbol Table

One of the chief tasks of the compilation process is the production of a symbol table, which is a collection of data structures whose purpose is to store type, scope, and address information about program data. Compilers and assemblers create the symbol table. It is read and may be modified by linkers, profiling tools, and assorted object manipulation tools. It also contains information required for debugging.

For large applications, a single compilation can involve many program components, including source files, header files, and libraries. Data from all of these files must be described in the symbol table.

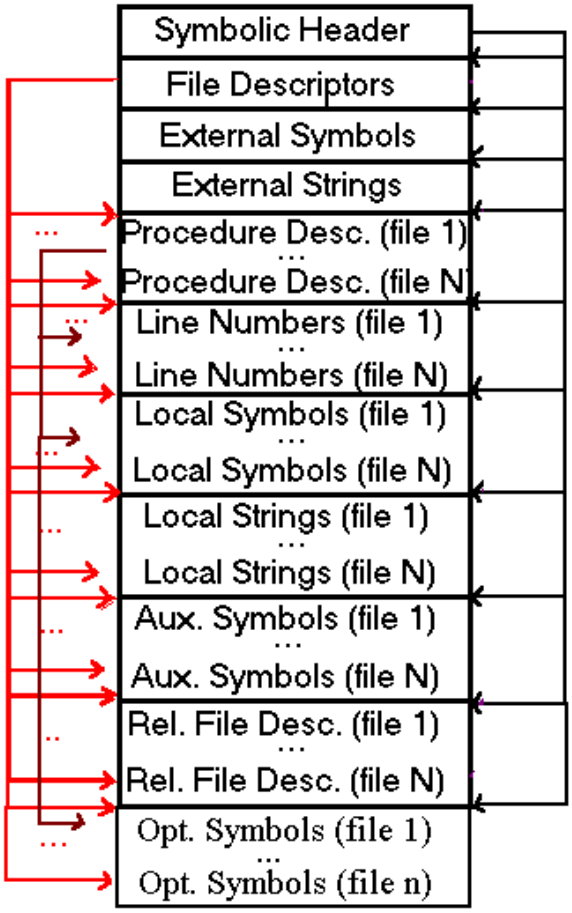
The Tru64 UNIX eCOFF symbol table, when present, comprises a large portion of the physical object file and is often considered a stand-alone entity. It is divided into numerous sections, including a header section that is used for navigation. The contents of the symbol table are shown in Figure 5–1.

Figure 5–1: Symbol Table Sections



The symbol table has a hierarchical design. The sections storing local symbols, local strings, relative file descriptors, procedure descriptors, line numbers, auxiliary symbols, and optimization symbols are divided into subtables and organized by file. Local symbols, local strings, and optimization symbols are further broken down by procedure. Figure 5–2 depicts this hierarchy.

Figure 5–2: Symbol Table Hierarchy



A particular symbol table may not contain all sections, for one of the following reasons:

- Relative file descriptors are present in linked objects only.
- The line number, auxiliary symbol and optimization symbol tables are produced only when debugging information is requested.
- Symbol table information may be partially or entirely removed by post-link object tools.
- Optimization symbols are not present in symbol table formats less than **V3.13**.

The function of each symbol table section is summarized below:

- The symbolic header stores the sizes and locations of all other symbol table sections.
- The line number table enables debuggers to map machine instructions to source code lines.
- The procedure descriptor table contains call-frame information as well as pointers to a procedure's local symbols, line numbers and optimization entries.
- The local symbol table describes procedures, static and local data, and user-defined types.
- The external symbol table stores information about global symbols.
- The relative file descriptor table contains a post-link file descriptor table index mapping for each file in the compilation.

- The local and external string tables store local and external symbol names, respectively.
- The file descriptor table stores the sizes and locations of each subtable produced for contributing source and include files. It also contains miscellaneous information about each file, such as the source language and the level of symbolic information.
- The auxiliary symbol table contains data type information for local and external symbols.
- The optimization symbols section stores procedure relative information, including extended source location information and optimized debugging information.

Several tools are available to view the contents of the symbol table. See the `stdump`(1), `odump`(1), and `nm`(1) man pages.

This chapter covers symbol table organization and usage, concentrating on debugging issues in particular. The current version of the symbol table is **V3.13**. The dynamic symbol table built by the linker is discussed separately in Section 6.3.3.

5.1 New or Changed Symbol Table Features

Tru64 UNIX V5.1 includes the following new or changed features:

- Alignment for common storage class symbols (see Section 5.2.6 and Section 2.3.5)
- Tail call flag used in procedure call optimization (see Section 5.2.3)
- A new ESLI command to describe gaps in address ranges (see Section 5.3.2.2)
- A new basic type for 32-byte complex (see Table 5–5).
- A new representation for empty classes or structures (see Section 5.3.8.6.1) to distinguish them from opaque classes and structures (see Section 5.3.8.6.2).

Version 3.13 of the symbol table includes the following new or changed features:

- 64-bit auxiliary support (see Section 5.3.7.3)
- Parameters with static storage and unallocated parameters (see Section 5.2.11)
- New optimization symbols section (see Section 5.3.3)
- Extended Source Location Information (see Section 5.3.2.2)
- New representation for procedures with no text (see Section 5.3.6.1)
- Modified variant record representation (see Section 5.3.8.11)
- New function pointer representation (see Section 5.3.8.5)
- Block symbol added for alternate entry prologue size (see Section 5.3.6.7)
- Address of locally stripped `FDR`s set to `addressNil` (see Section 5.3.1.2)
- Uplevel links for referencing local symbols in an outer scope (see Section 5.3.4.4)
- New profile feedback information (see Section 5.3.5)
- New representation for C++ namespaces (see Section 5.3.6.4)
- Unnamed union or structure representation (see Section 5.3.8.3)

5.2 Structures, Fields and Values for Symbol Tables

Unless otherwise specified, all structures described in this section are declared in the header file `sym.h`, and all constants are defined in the header file `symconst.h`.

5.2.1 Symbolic Header (HDRR)

```
typedef struct {
        coff_ushort     magic;
        coff_ushort     vstamp;
        coff_int        ilineMax;
        coff_int        idnMax;
        coff_int        ipdMax;
        coff_int        isymMax;
        coff_int        ioptMax;
        coff_int        iauxMax;
        coff_int        issMax;
        coff_int        issExtMax;
        coff_int        ifdMax;
        coff_int        crfd;
        coff_int        iextMax;
        coff_long       cbLine;
        coff_off        cbLineOffset;
        coff_off        cbDnOffset;
        coff_off        cbPdOffset;
        coff_off        cbSymOffset;
        coff_off        cbOptOffset;
        coff_off        cbAuxOffset;
        coff_off        cbSsOffset;
        coff_off        cbSsExtOffset;
        coff_off        cbFdOffset;
        coff_off        cbRfdOffset;
        coff_off        cbExtOffset;
} HDRR, *pHDRR;
```

SIZE - 144 bytes, ALIGNMENT - 8 bytes

Symbolic Header Fields

`magic` — To verify validity of the symbol table, this field must contain the constant `magicSym`, defined as `0x1992`.

`vstamp` — Symbol table version stamp. This value consists of a major version number and a minor version number, as defined in the `stamp.h` header file:

Symbol	Value	Description
`MAJ_OBJ_STAMP`	3	Current major object format version
`MIN_OBJ_STAMP`	13	Current minor object format version

See Section 1.4.5 for a description of object and symbol table versioning.

`ilineMax` — Number of line number entries (if expanded).

`idnMax` — Obsolete.

`ipdMax` — Number of procedure descriptors.

`isymMax` — Number of local symbols.

`ioptMax` — Byte size of optimization symbol table.

`iauxMax` — Number of auxiliary symbols.

`issMax` — Byte size of local string table.

issExtMax	Byte size of external string table.
ifdMax	Number of file descriptors.
crfd	Number of relative file descriptors.
iextMax	Number of external symbols.
cbLine	Byte size of (packed) line number entries.
cbLineOffset	Byte offset to start of (packed) line numbers.
cbDnOffset	Obsolete.
cbPdOffset	Byte offset to start of procedure descriptors.
cbSymOffset	Byte offset to start of local symbols.
cbOptOffset	Byte offset to start of optimization entries.
cbAuxOffset	Byte offset to start of auxiliary symbols.
cbSsOffset	Byte offset to start of local strings.
cbSsExtOffset	Byte offset to start of external strings.
cbFdOffset	Byte offset to start of file descriptors.
cbRfdOffset	Byte offset to start of relative file descriptors.
cbExtOffset	Byte offset to start of external symbols.

General Notes:

The size and offset fields describing symbol table sections must be set to zero if the section described is not present.

The cb*Offset fields are byte offsets from the beginning of the object file.

The i*Max fields contain the number of entries for a symbol table section. Legal index values for a symbol table section will range from 0 to the value of the associated i*Max field minus one.

For an explanation of packed and expanded line number entries, see the discussion in Section 5.3.2.2.

5.2.2 File Descriptor Entry (FDR)

```
typedef struct fdr {
        coff_addr       adr;
        coff_long       cbLineOffset;
        coff_long       cbLine;
        coff_long       cbSs;
        coff_int        rss;
        coff_int        issBase;
        coff_int        isymBase;
        coff_int        csym;
        coff_int        ilineBase;
        coff_int        cline;
```

```
        coff_int        ioptBase;
        coff_int        copt;
        coff_int        ipdFirst;
        coff_int        cpd;
        coff_int        iauxBase;
        coff_int        caux;
        coff_int        rfdBase;
        coff_int        crfd;
        coff_uint       lang : 5;
        coff_uint       fMerge : 1;
        coff_uint       fReadin : 1;
        coff_uint       fBigendian : 1;
        coff_uint       glevel : 2;
        coff_uint       fTrim : 1;
#ifndef TANDEMSYM
        coff_uint       reserved : 5;
#else
        coff_uint       platform : 3;   (not supported)
        coff_uint       reserved : 2;
#endif
        coff_ushort     vstamp;         (SV3.13 - )
        coff_uint       reserved2;
} FDR, *pFDR;
```

SIZE - 96 bytes, ALIGNMENT - 8 bytes

See Section 5.3.2.1 for related information.

File Descriptor Table Entry Fields

`adr` — Address of first instruction generated from this source file, which should be the same value as found in the `PDR.adr` field of the first procedure descriptor for this file. If no instructions are associated with this source file, this field should be set to `0`. File descriptors that have been merged by source language in locally-stripped objects will have this field set to `addressNil (-1)`.

Version Note

This use of `addressNil` is supported in symbol table format **V3.13** and greater.

`cbLineOffset` — Byte offset from start of packed line numbers to start of entries for this file.

`cbLine` — Byte size of packed line numbers for this file.

`cbSs` — Byte size of local string table entries for this file.

`rss` — Byte offset from start of file's local string table entries to source file name; set to `issNil (-1)` to indicate the source file name is unknown.

`issBase` — Start of local strings for this file.

`isymBase` — Starting index of local symbol entries for this file.

`csym` — Count of local symbol entries for this file.

`ilineBase` — Debuggers and other tools expand the packed line numbers, producing an array of line numbers with an entry for each machine instruction in the program. This field is an index

for this file's first line number entry in the expanded line number array.

`cline` — See the preceding description of `ilineBase`. This field is a count of this file's entries in the expanded line number array.

`ioptBase` — Byte offset from start of optimization symbol table to optimization symbol entries for this file.

`copt` — Byte size of optimization symbol entries for this file.

`ipdFirst` — Starting index of procedure descriptors for this file.

`cpd` — Count of procedure descriptors for this file.

`iauxBase` — Starting index of auxiliary symbol entries for this file.

`caux` — Count of auxiliary symbol entries for this file.

`rfdBase` — Starting index of relative file descriptors for this file.

`crfd` — Count of relative file descriptors for this file.

`lang` — Source language for this file (see Table 5–1).

`fMerge` — Informs linker whether this file can be merged.

`fReadin` — True if file was read in (as opposed to just created).

`fBigendian` — Unused.

`glevel` — Symbolic information level with which this file was compiled. This value is not the same as the user's idea of debugging levels. The value mapping from the user level `-g` option to the symbol table value is:

Debug switch	glevel contents
`-g0`	2
`-g1`	1
`-g2`	0
`-g3`	3

`fTrim` — Unused.

`platform` — Identifies the platform associated with the file descriptor. Set to `platUndef`, `platGuard`, `platOss`, or `platPc`.

Version Note

The `platform` field is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX

`vstamp` Symbol table version stamp (`HDRR.vstamp`) value from the original object module (.o file) that is recorded by the linker. The linker may combine objects that were compiled at different times and potentially contain different versions of the symbol table. In post-link objects, this value may or may not match the version stamp in the symbolic header. For pre-link objects, the value in this field will either be zero or the same as the symbolic header stamp.

Version Note

The `vstamp` field is supported on Tru64 UNIX **V5.0** and greater for symbol table version **V3.13** and greater.

`reserved` Must be zero.

`reserved2` Must be zero.

General Notes:

The `i*Base` fields provide the starting indices of this file's subtables within the symbol table sections. If the associated count fields are set to 0, the base fields will also be set to zero.

For an explanation of packed and expanded line number entries, see the discussion in Section 5.3.2.2.

Table 5–1: Source Language (`lang`) Constants

Name	Value	Commant
`langC`	0	
`langPascal`	1	
`langFortran`	2	
`langAssembler`	3	
`langMachine`	4	
`langNil`	5	
`langAda`	6	
`langPl1`	7	
`langCobol`	8	
`langStdc`	9	
`langMIPSCxx`	10	Unused.
`langDECCxx`	11	
`langCxx`	12	

Table 5–1: Source Language (`lang`) Constants (cont.)

Name	Value	Commant
langFortran90	13	Not used by all compilers - langFortran might be used instead for both f77 and f90
langBliss	14	
langPTAL	15	**(not supported)**
langCplusplusV1	16	**(not supported)**
langCplusplusV2	17	**(not supported)**
langMax	31	Number of language codes available

Version Note

The language constants langPTAL, langCplusplusV1, and langCplusplusV2 are reserved for use on Tandem big-endian systems. They are not supported on Tru64 UNIX.

5.2.3 Procedure Descriptor Entry (PDR)

```
#ifndef TANDEMSYM
struct pdr {
#else
struct pdrv4 {
#endif
        coff_addr       adr;
        coff_long       cbLineOffset;
        coff_int        isym;
        coff_int        iline;
        coff_uint       regmask;
        coff_int        regoffset;
        coff_int        iopt;
        coff_uint       fregmask;
        coff_int        fregoffset;
        coff_int        frameoffset;
        coff_int        lnLow;
        coff_int        lnHigh;
        coff_uint       gp_prologue : 8;
        coff_uint       gp_used : 1;
        coff_uint       reg_frame : 1;
        coff_uint       prof : 1;
        coff_uint       gp_tailcall : 1;   (V5.1 - )
#ifndef TANDEMSYM
        coff_uint       reserved : 12;
#else
        coff_uint       optlevel : 4;      (not supported)
        coff_uint       reserved : 8;
#endif
        coff_uint       localoff : 8;
        coff_ushort     framereg;
        coff_ushort     pcreg;
#ifdef TANDEMSYM
        coff_uint       proctype : 16;     (not supported)
        coff_uint       reserved2 : 48;
} PDRV4, *pPDRV4;
#else
} PDR, *pPDR;
#endif
```

SIZE - 64 bytes (72 bytes for Tandem), ALIGNMENT - 8 bytes

See Section 5.3.4 for related information.

Procedure Descriptor Table Entry Fields

adr — The start address of this procedure. Set to addressNil (-1) for procedures with no text.

Version Note

Prior to symbol table format **V3.13** this field may not be updated by the linker. To determine the procedure start address for symbol table formats **V3.10 - V3.12**, use the algorithm described in Section 5.3.4.1.

`cbLineOffset` Byte offset to the start of this procedure's packed line numbers from the start of the file descriptor entry (`FDR.cbLineOffset`).

`isym` Start of local symbols for this procedure. This symbol is the symbol for the procedure (symbol type `stProc`). The name of the procedure can be obtained from the `iss` field of the symbol table entry.

If the object is stripped of local symbol information, this field contains an external symbol table index for the procedure symbol's entry.

If this procedure has no symbols associated with it, this field should be set to `isymNil (-1)`. This situation occurs for a static procedure in an object stripped of local symbol information.

`iline` Start of line number entries (if expanded) for this procedure. Set to `ilineNil (-1)` to indicate that this procedure does not have line numbers.

`regmask` Saved general register mask.

`regoffset` Offset from the virtual frame pointer to the general register save area in the stack frame.

`iopt` Start of procedure's optimization symbol entries. Set to `ioptNil (-1)` to indicate that this procedure does not have optimization symbol entries.

`fregmask` Saved floating-point register mask.

`fregoffset` Offset from the virtual frame pointer to the floating-point register save area in the stack frame.

`frameoffset` Size of the fixed part of the stack frame. The actual frame size can exceed this value. A routine can extend its own frame size for frame sizes larger than 2 GB or for dynamic stack allocation requests.

`lnLow` Lowest source line number within this file for the procedure. This is typically the line number of the first instruction in the procedure, but not always. Code optimizations can rearrange or remove instructions making the first instruction map to a different line number.

`lnHigh` — Highest source line number within this file for the procedure. This field contains a value of `-1` for alternate entry points, which is how an alternate entry point is identified.

`gp_prologue` — Byte size of GP prologue.

`gp_used` — Flag set if the procedure uses GP.

`reg_frame` — True if the procedure is a light-weight or null-weight procedure. See the General Notes section following these definitions for more details on procedure weights.

`prof` — True if the procedure has been compiled with `-pg` for `gprof` profiling.

`gp_tailcall` — Indicates that a call to this procedure may result in a tail call return from a different GP domain. This bit is used exclusively for tail call optimizations.

Version Note

The `gp_tailcall` field is supported in Tru64 UNIX **V5.1** and greater.

`optlevel` — Optimization level. Set to `0` for unknown or `1` through `6` for optimization levels `0` through `5` respectively.

Version Note

The `optlevel` field is used on Tandem big-endian systems. It is not supported on Tru64 UNIX.

`reserved` — Must be zero.

`localoff` — Bias value for accessing local symbols on the stack at run time.

`framereg` — Frame pointer register number.

`pcreg` — PC (Program Counter) register number.

`proctype` — Procedure attribute flags. See Table 5–2 for flag descriptions.

Version Note

The `proctype` field and the associated flag values in Table 5–2 are reserved for use on Tandem big-endian systems. They are not supported on Tru64 UNIX.

Table 5–2: Procedure Attribute Flags

Flag	Value	Description
TNDM_MAIN	0x0001	Main entry point
TNDM_RESIDENT	0x0002	Resident routine
TNDM_PRIVILEGED	0x0004	Privileged routine
TNDM_CALLABLE	0x0008	Callable routine
TNDM_ENTRY	0x0010	Alternate entry, procedure, or subprocedure
TNDM_SUBPROC	0x0020	Subprocedure
TNDM_INTERRUPT	0x0040	Interrupt routine
TNDM_SHELL	0x0080	Shell routine
TNDM_COMPILER_GENERATED	0x0200	Procedure can have multiple copies
TNDM_EXTENSIBLE	0x0800	Extensible procedure
TNDM_EDITLINE	0x8000	Edit line numbers

General Notes:

For more information on call frames, see Section 5.3.4.2.

If the value of gp_prologue is zero and gp_used is 1, a gp prologue is present but was scheduled into the procedure prologue. Otherwise, the gp_prologue field gives the number of bytes occupied by the GP prologue instructions at the procedure's start address.

If there is a chain of tail call procedures, some of which are in the same GP domain, and some that are in a different GP domain, then gp_tailcall must be set for all procedures in the chain. For example, suppose there is a tail call from A to B, and a tail call from B to C. A and B are in the same GP domain, but C is in a different GP domain. In this case gp_tailcall must be set in both A's and B's PDR, because callers can't rely on the standard definition of GP after calling A. See the *Alpha Architecture Reference Manual* for additional details.

For an explanation of packed and expanded line number entries, see the discussion in Section 5.3.2.2.

A procedure may be heavy-, light-, or null-weight. The weight of a procedure can be determined from its descriptor by using the following guidelines:

Weight	Indications
Heavy	reg_frame is 0 and bit 26 of the register mask (regmask) is on
Light	reg_frame is 1 and regoffset is ra_save
Null	reg_frame is 1 and regoffset is 26

See the *Calling Standard for Alpha Systems* for details on the calling conventions for different weight procedures. Note that a calling routine does not need to know the weight of the routine being called.

5.2.4 Line Number Entry (LINER)

Line numbers are represented using two formats: packed and expanded. The packed format is a byte stream that can be interpreted as described in Section 5.3.2.2 to build an expanded table that maps instructions to source line numbers. The LINER type is used to refer to a single entry in the expanded table. It is declared as:

```
typedef int LINER, *pLINER;
```

A second, newer form of line number information is located in the optimization symbols section. See Section 5.2.10 and Section 5.3.2.2.

5.2.5 Local Symbol Entry (SYMR)

```
typedef struct {
        coff_long       value;
        coff_int        iss;
        coff_uint       st : 6;
        coff_uint       sc  : 5;
        coff_uint       reserved : 1;
        coff_uint       index : 20;
} SYMR, *pSYMR;
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

See Section 5.2.11, Section 5.3.4, and Section 5.3.8 for related information.

Local Symbol Table Entry Fields

`value` — A field that can contain an address, size, offset, or index. Its interpretation is determined by the symbol type and storage class combination, as explained in Section 5.2.11.

`iss` — Byte offset from the `issBase` field of a file descriptor table entry to the name of the symbol. If the symbol does not have a name, this field is set to `issNil (-1)`. Generally, all user-defined symbols have names. A symbol without a name is one that has been created by the compilation system for its own use.

`st` — Symbol type (see Table 5–3).

`sc` — Storage class (see Table 5–4).

`reserved` — Must be zero.

`index` — An index into either the local symbol table or auxiliary symbol table, depending on the symbol type and class. The index is used as an offset from the `isymBase` field in the file descriptor entry for an entry in the local symbol table or an offset from the `iauxBase` field for an entry in the auxiliary symbol table.

The index field may have a value of `indexNil`, which is defined as (long)`0xfffff`. This value is used to indicate that the index is not a valid reference.

The next two tables contain all defined values for the `st` and `sc` constants, along with short descriptions. However, these fields must be considered as pairs that have a limited number of possible pairings as explained in Section 5.2.11.

Table 5–3: Symbol Type (`st`) Constants

Constant	Value	Description
`stNil`	0	Dummy entry
`stGlobal`	1	Global variable
`stStatic`	2	Static variable
`stParam`	3	Procedure argument

Table 5–3: Symbol Type (st) Constants (cont.)

Constant	Value	Description
stLocal	4	Local variable
stLabel	5	Label
stProc	6	Global procedure
stBlock	7	Start of block
stEnd	8	End of block, file, or procedure
stMember	9	Member of class, structure, union, or enumeration
stTypedef	10	User-defined type definition
stFile	11	Source file name
stStaticProc	14	Static procedure
stConstant	15	Constant data
stBase	17	Base class (for example, C++)
stVirtBase	18	Virtual base class (for example, C++)
stTag	19	Data structure tag value (for example, C++ class or struct)
stInter	20	Interlude (for example, C++)
stModule	22	**(not yet implemented)** Fortran90 module definition.
stNamespace	22	**(V5.0 -)** Namespace definition (for example, C++)
stModview	23	**(not yet implemented)** Modifiers for current view of given module.
stUsing	23	**(V5.0 -)** Namespace use (for example, C++ "using").
stAlias	24	**(V5.0 -)** Defines an alias for another symbols. Currently, only used for namespace aliases.
stDefine	25	**(not supported)** Macro definition
stObjinfo	26	**(not supported)** Name/data object info
stToolinfo	27	**(not supported)** Compiler info
stSrcinfo	28	**(not supported)** Source data info
stEquivRel	29	**(not supported)** Equivalence variable
stMax	64	Maximum number of symbol types

General Notes:

Symbol type codes with more than one interpretation are identified by the lang field in the associated file descriptor. This applies to the stModule/stNamespace and stModview/stUsing symbol types.

Version Note

The symbol types: stDefine, stObjinfo, stToolinfo, stSrcinfo, and stEquivRel are reserved for use on Tandem big-endian systems. They are not supported on Tru64 UNIX.

Table 5–4: Storage Class (sc) Constants

Constant	Value	Description
scNil	0	Dummy entry
scText	1	Symbol allocated in the .text section
scData	2	Symbol allocated in the .data section
scBss	3	Symbol allocated in the .bss section
scRegister	4	Symbol allocated in a register
scAbs	5	Symbol value is absolute
scUndefined	6	Symbol referenced but not defined in the current module
scUnallocated	7	Storage not allocated for this symbol
scResText	8	**(not supported)** Resident text
scTlsUndefined	9	TLS symbol referenced but not defined in the current module
scInfo	11	Symbol contains debugger information
scSData	13	Symbol allocated in the .sdata section
scSBss	14	Symbol allocated in the .sbss section
scRData	15	Symbol allocated in the .rdata section
scVar	16	Parameter passed by reference (for example, Fortran or Pascal)
scCommon	17	Common symbol
scSCommon	18	Small common symbol
scVarRegister	19	Parameter passed by reference in a register
scVariant	20	Variant record (for example, Pascal or Ada)
scFileDesc	20	File descriptor (for example, COBOL)
scSUndefined	21	Small undefined symbol
scInit	22	Symbol allocated in the .init section
scReportDesc	23	Report descriptor (for example, COBOL)
scXData	24	Symbol allocated in the .xdata section
scPData	25	Symbol allocated in the .pdata section
scFini	26	Symbol allocated in the .fini section
scRConst	27	Symbol allocated in the .rconst section
scTlsCommon	29	TLS common symbol
scTlsData	30	Symbol allocated in the .tlsdata section
scTlsBss	31	Symbol allocated in the .tlsbss section
scMax	32	Maximum number of storage classes

Version Note

The scResText storage class is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX.

5.2.6 External Symbol Entry (EXTR)

```
typedef struct {
        SYMR            asym;
        coff_uint       jmptbl : 1;
        coff_uint       cobol_main : 1;
        coff_uint       weakext : 1;
```

```
        coff_uint     alignment : 4;   (V5.1 - )
#ifdef TANDEMSYM
        coff_uint     xport : 1;       (not supported)
        coff_uint     multiext : 1;    (not supported)
        coff_uint     reserved : 23;
#else
        coff_uint     reserved:25;
#endif
        coff_int      ifd;
} EXTR, *pEXTR;
```

SIZE - 24 bytes, ALIGNMENT - 8 bytes

External Symbol Table Entry Fields

`asym` — External symbol table entry. This structure has the same format as a local symbol entry. The field interpretations differ as described in the following entries.

`asym.value` — Contains the symbol address for most defined symbols. See Section 5.2.11 for details.

`asym.iss` — Byte offset in external string table to symbol name. Set to `issNil (-1)` if there is no name for this symbol.

`asym.st` — Symbol type. See Table 5–3 for possible values.

`asym.sc` — Storage class. See Table 5–4 for possible values.

`asym.reserved` — Must be zero.

`asym.index` — Contains either an index into the auxiliary symbol table for a type description or an index into the local symbol table pointing to a related symbol.

The index field may have a value of `indexNil`, which is defined as `(long)0xfffff`. This value is used to indicate that the index is not a valid reference.

`jmptbl` — Unused.

`cobol_main` — Flag set to indicate that the symbol is a COBOL main procedure.

`weakext` — Flag set to identify the symbol as a weak external. See Section 6.3.4.2 for more details on weak symbols.

`alignment` — Power of two byte alignment for common storage class symbols biased by 2^3 (8). Supported values range from 0 through 13 yielding a minimum alignment of 8 bytes and a maximum alignment of 64K bytes. For symbols with storage classes other than `scCommon` and `scSCommon` this field should be ignored.

Version Note

The `alignment` field is supported on Tru64 UNIX **V5.1** and greater.

`xport` Flag set to indicate the symbol is to be exported from a shared library.

Version Note

The `xport` field is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX.

`multiext` Flag set to indicate that multiple definitions of the symbol are allowed.

Version Note

The `multiext` field is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX.

`reserved` Must be zero.

`ifd` Index of the file descriptor where the symbol is defined. Set to `ifdNil (-1)` for undefined symbols and for some compiler system symbols.

5.2.7 Relative File Descriptor Entry (RFDT)

The relative file descriptor table provides a post-link mapping of file descriptor indices. The purpose of this table is to minimize work for the linker, which does not update symbol table references to local symbols. This information is used to obtain the file offset used to bias local symbol indices. Because this table is also known as the File Indirect Table, two declarations are included in the `sym.h` header file, as shown here.

```
typedef int RFDT, *pRFDT;
typedef int FIT, *pFIT;
```

SIZE - 4 bytes, ALIGNMENT - 4 bytes

See Section 5.3.2.1 for related information.

5.2.8 Auxiliary Symbol Table Entry (AUXU)

The auxiliary symbol table entry is a 32-bit union. It is either interpreted as a `TIR` or `RNDXR` structure or as an integer value. See Section 5.3.7.3 for detailed instructions on reading the auxiliary symbols.

```
typedef union {
        TIR             ti;
        RNDXR           rndx;
        coff_int        dnLow;
        coff_int        dnHigh;
        coff_int        isym;
        coff_int        iss;
        coff_int        width;
        coff_int        count;
        coff_int        slice;    (V5.0a)
} AUXU, *pAUXU;
```

SIZE - 4 bytes, ALIGNMENT - 4 bytes

See Section 5.3.7.3 for related information.

Auxiliary Symbol Table Entry Fields

ti	Type information record (TIR), as defined in Section 5.2.8.1.
rndx	Relative index into local or auxiliary symbols (rndx), as defined in Section 5.2.8.2.
dnLow	Lower bound of range or array dimension. For large structures, two of these fields can be used together to form one 64-bit number.
dnHigh	Upper bound of range or array dimension. For large structures, two of these fields can be used together to form one 64-bit number.
isym	For procedures (stProc or stStaticProc symbols), this field is an index into the local symbols. It is also used as an index into the relative file descriptors.
iss	Unused.
width	Width of a bit field or array stride in bits. Fortran compilers set the array stride to the array element size in bits. Two of these fields can be used together to form one 64-bit number.
count	Count of ranges for variant arm. This field name is only used within the type description of a variant block (stBlock, scVariant).
slice	Reserved.

General Notes:

The fields dnLow, dnHigh, or width must all use either the 32-bit or 64-bit representation when used together. For example, an array dimension cannot be specified with a 32-bit dnLow and a 64-bit dnHigh.

5.2.8.1 Type Information Record (TIR)

```
typedef struct {
        coff_uint       fBitfield : 1;
        coff_uint       continued : 1;
        coff_uint       bt  : 6;
        coff_uint       tq4 : 4;
        coff_uint       tq5 : 4;
        coff_uint       tq0 : 4;
        coff_uint       tq1 : 4;
        coff_uint       tq2 : 4;
        coff_uint       tq3 : 4;
} TIR, *pTIR;
```

SIZE - 4 bytes, ALIGNMENT - 4 bytes

Type Information Record Entry Fields

fBitfield	Flag set if bit width is specified.
continued	Flag set to indicate that the type description is continued in another TIR record. This will happen if the type is represented with more than six type qualifiers.

bt	Basic type (see Table 5–5 and Section 5.3.7.1).
tq0, tq1, tq2, tq3, tq4, tq5	Type qualifiers (see Table 5–6 and Section 5.3.7.2). The lower-numbered tq fields must be used first, and all unneeded fields must be set to tqNil (0).

Table 5–5: Basic Type (bt) Constants

Constant	Value	Description
btNil	0	Undefined or void
btAdr32	1	Address (32 bits)
btChar	2	Character
btUChar	3	Unsigned character
btShort	4	Short (16 bits)
btUShort	5	Unsigned short (16 bits)
btInt	6	Integer (32 bits)
btUInt	7	Unsigned integer (32 bits)
btLong32	8	Long (32 bits)
btULong32	9	Unsigned long (32 bits)
btFloat	10	Floating point
btDouble	11	Double-precision floating point
btStruct	12	Structure or record
btUnion	13	Union
btEnum	14	Enumeration
btTypedef	15	Defined by means of a user-defined type definition
btRange	16	Range of values (for example, Pascal subrange)
btSet	17	Sets (for example, Pascal)
btComplex	18	Single complex (for example, Fortran COMPLEX*8)
btDComplex	19	Double complex (for example, Fortran COMPLEX*16)
btIndirect	20	Indirect definition; following rndx points to an entry in the auxiliary symbol table that contains a TIR (type information record)
btFixedBin	21	Fixed binary (for example, COBOL)
btDecimal	22	Packed or unpacked decimal (for example, COBOL)
btPicture	25	Picture (for example, COBOL)
btVoid	26	Void
btPtrMem	27	Currently unused
btScaledBin	27	Scaled binary (for example, COBOL)
btVptr	28	Virtual function table (for example, C++)
btArrayDesc	28	Array descriptor (for example, Fortran, Pascal)
btClass	29	Class (for example, C++)
btLong64	30	Address (64 bits)
btLong	30	Long (64 bits)
btULong64	31	Unsigned long (64 bits)
btULong	31	Unsigned long (64 bits)

Table 5–5: Basic Type (bt) Constants (cont.)

Constant	Value	Description
btLongLong	32	Long long (64 bits)
btULongLong	33	Unsigned long long (64 bits)
btAdr64	34	Address (64 bits)
btAdr	34	Address (64 bits)
btInt64	35	Integer (64 bits)
btUInt64	36	Unsigned integer (64 bits)
btLDouble	37	Long double floating point (128 bits)
btInt8	38	Integer (64 bits)
btUInt8	39	Unsigned integer (64 bits)
btRange_64	41	**(V5.0 -)** 64-bit range
btProc	42	**(V5.0 -)** Procedure or function
btCobolIndex	43	**(not supported)** COBOL index variables
btReal32	44	**(not supported)** Tandem float
btReal64	45	**(not supported)** Tandem double
btQComplex	46	**(V5.1 -)** Quad complex (for example Fortran COMPLEX*32)
btChecksum	63	Symbol table checksum value stored in auxiliary record
btMax	64	Number of basic type codes

Table Notes:

1. btInt and btLong32 are synonymous.
2. btUInt and btULong32 are synonymous.
3. btLong, btLong64, btLongLong, btInt64, and btInt8 are synonymous.
4. btULong64, btULongLong, btUInt64, and btUInt8 are synonymous.

Version Note

The basic type constants: btCobolIndex, btReal32, and btReal64 are reserved for use on Tandem big-endian systems. They are not supported on Tru64 UNIX.

Table 5–6: Type Qualifier (tq) Constants

Constant	Value	Description
tqNil	0	No qualifier (placeholder)
tqPtr	1	Pointer
tqProc	2	**(obsolete)** Procedure or function
tqArray	3	Array
tqFar	4	32-bit pointer; used with the -xtaso emulation
tqVol	5	Volatile
tqConst	6	Constant
tqRef	7	Reference

Table 5–6: Type Qualifier (tq) Constants (cont.)

Constant	Value	Description
tqArray_64	8	**(V5.0 -)** Large array
tqHasLen	9	**(not supported)** Length for buffer parameters
tqShar	10	**(V5.0a -)** Reserved
tqSharArr_64	11	**(V5.0a -)** Reserved
tqMax	16	Number of type qualifier codes

Version Note

The tqHasLen type qualifier is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX.

5.2.8.2 Relative Symbol Record (RNDXR)

```
typedef struct {
        coff_uint       rfd : 12;
        coff_uint       index : 20;
} RNDXR, *pRNDXR;
```

SIZE - 4, ALIGNMENT - 4

Relative Symbol Record Fields

rfd — Index into relative file descriptor table if it exists; otherwise, index into file descriptor table.

This field may have a value of ST_RFDESCAPE, defined as 0xfff in the header file cmplrs/stsupport.h. This value is used to indicate that the next auxiliary entry, interpreted as an isym, contains the actual rfd index.

index — Symbol index. Used as an offset from either FDR.isymbase or FDR.iauxbase, depending on context.

5.2.9 String Table

Objects can contain two string tables: the local string table (corresponding to local symbols) and the external string table (corresponding to external symbols). The local string table is present only for objects created with full debugging information; it is removed if an object is locally stripped.

The storage format for the string tables is a list of null-terminated character strings. It is correctly considered as one long character array, not an array of strings. Fields in the symbolic header and file headers represent string table sizes and offsets in bytes.

5.2.10 Optimization Symbol Entry (PPODHDR)

The optimization symbol table contains information for optimized debugging, basic block profiling, and other miscellaneous procedure-specific data. Each procedure's associated optimization symbol table data begins with an array of PPODHDR structures. See Section 5.3.3 for a description of the optimization symbol table.

Version Note

The following structure definition is for Tru64 UNIX **V5.0** and greater. It is used for symbol table format **V3.13** and greater.

```
typedef struct {
        coff_uint       ppode_tag;
        coff_uint       ppode_len;
        coff_ulong      ppode_val;
} PPODHDR, *pPPODHDR;
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

Optimization Symbol Entry Fields

`ppode_tag` Identifies the kind of data described by this entry.

`ppode_len` Indicates the size in bytes of the data that is found in the raw data area for this entry. When this field is zero, the only data is stored in the `ppode_val` field.

`ppode_val` This field is either a pointer to the entry's data or is itself the data. If `ppode_len` is nonzero, this field is a relative file offset from the beginning of the current PPOD (Per-Procedure Optimization Descriptor) to the applicable data area. If `ppode_len` is zero, this field contains the data for the entry.

A PPOD contains multiple `PPODHDR`s. A `PPODHDR` and its associated data are collectively referred to as a PPODE (Per-Procedure Optimization Descriptor Entry.) Figure 5–10 in Section 5.3.3 shows several PPODs with multiple `PPODHDR`s and their data.

Table 5–7: Optimization Tag Values

Name	Value	Description
PPODE_STAMP	1	Version number of the PPOD stored in ppode_val. The current PPOD_VERSION value is 1.
PPODE_END	2	End of entries for this PPOD.
PPODE_EXT_SRC	3	Extended source line information.
PPODE_SEM_EVENT	4	Semantic event information. (Reserved for future use.)
PPODE_SPLIT	5	Split lifetime information. (Reserved for future use.)
PPODE_DISCONTIG_SCOPE	6	Discontiguous scope information. (Reserved for future use.)
PPODE_INLINED_CALL	7	Inlined procedure call information. (Reserved for future use.)
PPODE_PROFILE_INFO	8	Profile feedback information.

5.2.11 Symbol Type and Class (st/sc) Combinations

Entries in the symbol table are primarily identified by the combination of their symbol type (`st`) and storage class (`sc`) values. Not all combinations are valid. Figure 5–3 indicates which combinations are currently in use.

Figure 5–3: st/sc Combination Matrix

st \ sc	B a s e dB	C o m	F i l e DDFII	R e l R Rp eo gr it sDSmD teBma essot	g C oS	S U n d eS fy imTsm nreBm eexso	T U T l s TC lo	lU sn U UaUs nlne Tdldr leoeS sfcft Diair antnu teeec	V a r R eV ga irX siD Vtaa aent
								batso	
								srssn	
								aciot	
								latae	
								rcsna	
								dftsn	
								adddt	rrta
Alias		X							
Base		X							
Block	X	X X		X		X			X
Constant	X X	X		X	X X				
End	X	X X		X		X			X
Expr									
File						X			
Forward									
Global	X XX	X		XX	XXX	X XX		XX X	
Inter		X							
Label	X X	X X X	XXX	X X		XX		X X	X
Local	X X	X X X	XXX	X X X		XX		X X	XX X
Member		X X		X					
Module									
Modview									
Namespace		X							
Nil									
Number									
Param	X X	X X	XX	X X X			X		XX
Proc		X	X			X		X	
RegReloc									
Split									
StaParam									
Static	X XX	X X	XX	X X		X		X	
StaticProc		X X				X			
Str									
Tag		X							
Type									
Typedef		X							
Using		X							
VirtBase		X							

A symbol's type and class taken together determines interpretation of other fields in the symbol table entry. The same combination can be used for different purposes in different contexts. As a result, to understand the symbol entry, it also may be necessary to access type information in the auxiliary table or the source language information in the file descriptor.

The contents of the `value` and `index` fields for each combination, with a brief explanation of the symbol's use, are described in the following list of combinations. For many combinations, greater detail can be found in Section 5.3.7 and Section 5.3.8.

stGlobal/scAbs

- The `value` field contains an absolute value.
- The `index` field is an auxiliary table index or `indexNil` if there is no type information.
- This symbol is a global absolute value.

stGlobal/scSData,
stGlobal/scData,
stGlobal/scSBss,
stGlobal/scBss,
stGlobal/scRData,
stGlobal/scRConst

- The `value` field is the symbol's address.

- The index field is an auxiliary table index or indexNil if there is no type information.
- This symbol is a defined global variable.

stGlobal/scTlsData,
stGlobal/scTlsBss

- The value field is the offset from the base of the object's TLS region.
- The index field is an auxiliary table index or indexNil if there is no type information.
- This symbol is a defined global TLS variable.

stGlobal/scSCommon,
stGlobal/scCommon,
stGlobal/scTlsCommon

- The value field is the symbol's size in bytes.
- The index field is an auxiliary table index or indexNil if there is no type information.
- This symbol is a common.

stGlobal/scSUndefined,
stGlobal/scUndefined,
stGlobal/scTlsUndefined

- The value field is zero in linked objects. In relocatable objects, the value field is ignored. (Some compilers store the size in bytes of the global variable in the value field.)
- The index field is an auxiliary table index or indexNil if there is no type information.
- This symbol is an undefined global variable.

stStatic/scAbs

- The value field is an absolute value.
- The index field is an auxiliary table index or indexNil if there is no type information.
- This symbol is an absolute value with static scope.

stStatic/scSData,
stStatic/scData,
stStatic/scSBss,
stStatic/scBss,
stStatic/scRData,
stStatic/scRConst

- The value field is the symbol's address.
- The index field is an auxiliary table index or indexNil if there is no type information.
- This symbol is a defined static variable.

stStatic/scTlsData,
stStatic/scTlsBss

- The value field is an offset from the base of the object's TLS region.
- The index field is an auxiliary table index or indexNil if there is no type information.

- This symbol is a defined static TLS variable.

stStatic/scCommon

- The `value` field is zero.
- The `index` field is an auxiliary table index or `indexNil` if there is no type information.
- This symbol is a Fortran common block.

stStatic/scInfo

- The `value` field is zero.
- The `index` field is an auxiliary table index.
- This symbol is a C++ static data member.

stParam/scAbs

- The `value` field is an offset from the virtual frame pointer.
- The `index` field is an auxiliary table index.
- This symbol is a parameter stored on the stack.

stParam/scRegister

- The `value` field is the number of the register containing the parameter.
- The `index` field is an auxiliary table index.
- This symbol is a parameter stored in a register.

stParam/scVar

- The `value` field is an offset from the virtual frame pointer to the parameter's address.
- The `index` field is an auxiliary table index.
- This symbol is a parameter stored on the stack. One level of indirection is required to access the parameter's value.

stParam/scVarRegister

- The `value` field is the register number containing the address of the parameter.
- The `index` field is an auxiliary table index.
- This symbol is a parameter stored on the stack. One level of indirection is required to access the parameter's value.

stParam/scInfo

- The `value` field is zero.
- The `index` field is an auxiliary table index.
- This symbol is a parameter of a C++ member function, function pointer definition, or procedure with no code.

stParam/scSData,
stParam/scData,
stParam/scSBss,
stParam/scBss,
stParam/scRData,
stParam/scRConst

- The `value` field is the address of the parameter.

- The index field is an auxiliary table index.
- This symbol is a static parameter.

Version Note

Static parameters are supported in symbol table format **V3.13** and greater.

stParam/scUnallocated

- The value field is zero.
- The index field is an auxiliary table index.
- This is an unallocated parameter.

stLocal/scAbs

- The value field is an offset from the virtual frame pointer.
- The index field is an auxiliary table index.
- This is a local variable stored on the stack.

stLocal/scRegister

- The value field is the number of the register containing the variable.
- The index field is an auxiliary table index.
- This symbol is a local variable stored in a register.

stLocal/scVar

- The value field is an offset from the virtual frame pointer to the symbol's address.
- The index field is an auxiliary table index.
- This symbol is a local variable stored on the stack. One level of indirection is required to access its value.

stLocal/scVarRegister

- The value field is the register number containing the address of this variable.
- The index field is an auxiliary table index.
- This symbol is a local variable stored on the stack. One level of indirection is required to access its value.

stLocal/scUnallocated

- The value field is zero.
- The index field is an auxiliary table index.
- This is an unallocated local variable.

Version Note

The use of scUnallocated is supported in symbol table format **V3.13** and greater.

stLocal/scText,
stLocal/scInit,
stLocal/scFini,

stLocal/scSData,
stLocal/scData,
stLocal/scSBss,
stLocal/scBss,
stLocal/scRData,
stLocal/scRConst,
stLocal/scTlsData,
stLocal/scTlsBss

- The `value` field is the address of the section indicated by the storage class.
- The `index` field is `indexNil`.
- These are special symbols inserted by the linker for shared objects. They are found in the external symbol table and their names are the section names (for example, `.text` or `.init`).

stLabel/scAbs

- The `value` field is the symbol's value. This may be either a numeric constant or absolute address.
- The `index` field is `indexNil`.
- This symbol is a linker defined absolute symbol.

stLabel/scText,
stLabel/scInit,
stLabel/scFini,
stLabel/scSData,
stLabel/scData,
stLabel/scXData,
stLabel/scPData,
stLabel/scSBss,
stLabel/scBss,
stLabel/scRData,
stLabel/scRConst,
stLabel/scTlsData,
stLabel/scTlsBss

- The `value` field is the label's value (an address).
- The `index` field is `indexNil`.
- This symbol is an allocated label. It can be associated with any raw data section of the object file.

stLabel/scUnallocated

- The `value` field is zero.
- The `index` field is `indexNil`.
- This symbol is an unallocated label.

stProc/scNil

- The `value` field is zero.
- The `index` field is `indexNil`.
- This symbol can be ignored. Compilers may produce this type/class combination for procedures that have been optimized away and that don't require debug information. The linker removes these symbols from the external symbol table in linked objects.

stProc/scText

- The value field is the procedure's address.
- This symbol can occur in the external or local symbol table:
  - In the local symbol table, the index field is an auxiliary table index.
  - In the external symbol table, it is the local symbol index of the corresponding procedure symbol in the local symbol table, unless the file is stripped of local symbol information. If the file is locally stripped, the index field is indexNil.
- This symbol is a defined procedure.

stProc/scUndefined

- The value field is zero.
- The index field is indexNil.
- This symbol is an undefined procedure.

stProc/scInfo

- The value field contains a value of:
  - -1 (a procedure with no code)
  - -2 (a function prototype or function pointer definition)
  - A non-negative index into the virtual function table for this function, for a C++ virtual member function.

Version Note

The use of -1 and -2 in the value field is supported in symbol table format **V3.13** and greater.

- The index field is an auxiliary table index.
- This symbol represents a procedure without code, a function prototype, or a function pointer. The value field is used to distinguish among these possibilities.

stBlock/scText

- The value field depends on context:
  - If this is the first stBlock/scText symbol following an stProc/scText symbol, the value is the byte offset from the procedure's address to the address of the first instruction beyond the end of the procedure's prologue.
  - Otherwise, it is the byte offset from the procedure's address to the starting instruction address of the block.
- The index field is the local symbol index of the symbol following the matching stEnd. If this is the first stBlock/scText following an stProc/scText for an alternate entry point, the index field will be set to indexNil because the symbol will not have a matching stEnd symbol.

Version Note

The use of stBlock/scText for alternate entry points is supported in symbol table format **V3.13** and greater.

- This symbol indicates the start of a block scope.

stBlock/scInfo

- The value field depends on context:
  - Size in bytes for a class, structure, or union.
  - Size of the underlying data type for an enumerated type.
  - Auxiliary table index for a variant record.
  - Zero for the block scope of a procedure with no code.
- The index field is the local symbol index of the symbol following the matching stEnd.
- This symbol indicates the start of a structure, union, or enumeration definition (in C; the C++ representation differs). It describes a variant arm if it is inside an stBlock/scVariant scope. This symbol is also used to define the block scope of a procedure with no code.

stBlock/scCommon

- The value field is the size of the common block in bytes.
- The index field is the local symbol index of the symbol following the matching stEnd.
- This symbol is a scoping symbol for a Fortran common block. It occurs in the context of the synthesized file used to define a common block.

stBlock/scVariant

- The value field is the local symbol index of the structure member whose value determines which variant range is used.
- The index field is a the local symbol index of the symbol following the matching stEnd.
- This symbol occurs in the context of Pascal and Ada variant records. It indicates the start of the symbols for one variant.

stBlock/scFileDesc,
stBlock/scReportDesc

- The value field is zero.
- The index field is a the local symbol index of the symbol following the matching stEnd.
- This symbol occurs in COBOL only. It indicates the start of the file or report descriptor scope.

stEnd/scText

- The value field depends on the type of scope it is ending. It is:
  - The size in bytes of the procedure's text (for a procedure).
  - Byte offset from a procedure's address to the start of the epilogue (for the outermost text block in a procedure).
  - Byte offset from a procedure's address to the first instruction address beyond the end of the block (for a text block).
  - Zero (for a file).
- The index field is the local symbol index of the matching stBlock, stProc, or stFile.
- This symbol ends a file, procedure, or text block scope.

stEnd/scInfo

- The value field is zero.
- The index field is a the local symbol index of the matching stBlock or stNamespace.
- If the matching symbol is an stBlock, this symbol ends a structure, union, enumeration, C++ member function definition, procedure with no code, or the block scope contained by a procedure with no code. If the matching symbol is an stNamespace, this symbol ends a namespace definition.

stEnd/scCommon

- The value field is zero.
- The index field is the local symbol index of the matching stBlock.
- This symbol ends a Fortran common definition.

stEnd/scVariant

- The value field is the same as that of the matching stBlock.
- The index field is the local symbol index of the matching stBlock.
- This symbol ends a variant record block.

stEnd/scFileDesc,
stEnd/scReportDesc

- The value field is zero.
- The index field is the local symbol index of the matching stBlock.
- This symbol ends a file or report descriptor block.

stMember/scInfo

- The value field depends on the symbol's data type:
  - The ordinal value (for an element of an enumerated type).
  - Zero (for a namespace or union member).
  - Bit offset from the beginning of the structure (for a C structure or C++ class member).
- The index field is an auxiliary table index.
- This symbol describes a data structure field or the member of a namespace. It is found inside a block defining a data structure (for example, class or struct) or a namespace definition block.

stMember/scFileDesc,
stMember/scReportDesc

- The value field is zero or one, depending on whether the symbol is local or external, respectively.
- The index field is an auxiliary table index.
- This symbol occurs in COBOL only. It is found inside a file descriptor or report descriptor block.

stTypedef/scInfo

- The value field depends on the purpose of this symbol:
  - Zero (for a user-defined type definition).

  - The auxiliary table index of the next auxiliary entry after the start of the class definition (for a compiler-inserted symbol). In effect, the value is the contents of the `index` field plus one.

- The `index` field is an auxiliary table index.
- This symbol is a user-chosen name for a data type. It also appears as a compiler-inserted symbol following the `stTag`/`scInfo` symbol for a C++ opaque class or structure.

stFile/scText

- The `value` field is zero.
- The `index` field is the local symbol index of the symbol following the matching `stEnd`.
- This symbol denotes the scoping block for a source file.

stStaticProc/scText

- The `value` field is the procedure's address.
- The `index` field is an auxiliary table index.
- This symbol is a defined static procedure.

stStaticProc/scInit,
stStaticProc/scFini

- The `value` field is the procedure address.
- The `index` field is an auxiliary table index.
- These combinations are used for the special symbols `__istart` and `__fstart`, which are inserted by the linker.

stConstant/scAbs

- The `value` field is the value of the constant.
- The `index` field is an auxiliary table index.
- This symbol represents a named value (for example, Fortran `PARAMETER`).

stConstant/scSData,
stConstant/scData,
stConstant/scSBss,
stConstant/scBss,
stConstant/scRData,
stConstant/scRConst

- The `value` field is the symbol's address.
- The `index` field is an auxiliary table index.
- This symbol represents allocated constant data.

stBase/scInfo

- The `value` field is the offset of the base class relative to a derived class.
- The `index` field is an auxiliary table index.
- This symbol is a C++ base class. It is found inside a block defining a data structure (for example, class or struct).

stVirtBase/scInfo

- The `value` field is an index (starting at 1) of the base class run-time description in the virtual base class table. See Section 5.3.8.6.3.
- The `index` field is an auxiliary table index.
- This symbol is a C++ virtual base class. It is found inside a block defining a data structure (for example, class or struct).

stTag/scInfo

- The `value` field is zero.
- The `index` field is an auxiliary table index.
- This symbol is a C++ class, structure, or union. See Section 5.3.8.6. Note that the representation for C structures and unions (Section 5.3.8.3) is different.

stInter/scInfo

- The `value` field is zero.
- The `index` field is an auxiliary table index.
- This symbol is used in C++ to connect the definition of a member function with its prototype in the class definition context.

stNamespace/scInfo

- The `value` field is zero.
- The `index` field is the local symbol index of the symbol following the matching `stEnd`.
- This symbol indicates the start of the symbols in a namespace definition.

Version Note

Namespace symbols are supported in symbol table format **V3.13** and greater.

stUsing/scInfo

- The `value` field is zero.
- The `index` field is an auxiliary table index.
- This symbol specifies a C++ namespace (or portion thereof) that is being imported into another scope.

Version Note

Namespace USING directives are supported in symbol table format **V3.13** and greater.

stAlias/scInfo

- The `value` field is zero.
- The `index` field is an auxiliary table index.
- This symbol defines an alias for a C++ namespace.

Version Note

Namespace aliases are supported in symbol table format **V3.13** and greater.

Combinations may be valid in the local symbol table, the external symbol table, or both. Table 5–8 shows which combinations are valid in which table, based on the symbol type value and also the storage class value where necessary. Only combinations previously specified as valid apply where the storage class value is shown as a wildcard value with the character '*'.

Table 5–8: Valid Placement for `st/sc` Combinations

`st/sc` Combination	External Symbol Table	Local Symbol Table
stNil, sc*	X	X
stGlobal, sc*	X	
stStatic, sc*		X
stParam, sc*		X
stLocal, *scSCN* [1]	X	
stLocal, not *scSCN* [1]		X
stLabel, sc*	X	X
stProc, scInfo		X
stProc, scText	X	X
stProc, scUndefined	X	
stBlock, sc*		X
stEnd, sc*		X
stMember, sc*		X
stTypedef, sc*		X
stFile, sc*		X
stStaticProc, scText		X
stStaticProc, scInit/scFini	X	
stConstant, sc*	X	X
stBase, sc*		X
stVirtBase, sc*		X
stTag, *		X
stInter, sc*		X
stNamespace, sc*		X
stUsing, sc*		X
stAlias, *		X

Table Notes:

1. *scSCN* is a section storage class: scData, scSData, scBss, scSBss, scRConst, scRData, scInit, scFini, scText, scXData, scPData, scTlsData, scTlsBss

5.3 Symbol Table Usage

5.3.1 Levels of Symbolic Information

Different levels of symbolic information can be stored with an object file. Compilers often provide options that allow the user to choose the desired level of symbolic information for their program. This choice may be influenced by size considerations and debugging needs. A trade-off exists between the benefit of saving space in the object file and the amount of information available to tools that consume symbolic information.

It is also possible to change the amount of symbolic information present in a program that has already been compiled and linked. Information can be added or deleted. Two of the most common and useful operations are locally stripping and fully stripping the symbol tables in executable files. Tools that modify linked executables, such as instrumentation tools and code optimizers, may rewrite parts of the symbol table to reflect changes that they made.

5.3.1.1 Compilation Levels

The representation of symbolic information supported by compilers can be broken down into four levels:

1. Minimal– Only information required for linking
2. Limited– Source file and line number information for profiling and limited debugging (stack-tracing)
3. Full– Complete debugging information for non-optimized code
4. Optimized– Debugging information for optimized code

These levels correspond to the system compiler switches `-g0` (minimal), `-g1` (limited), `-g2` (full), and `-g3` (optimized). Table 5–9 shows the symbol table sections that are produced by system compilers at each compilation level.

Table 5–9: Symbol Table Sections Produced at Various Compilation Levels

	Compilation Level			
Symbol Table Section	**Minimal**	**Limited**	**Full**	**Optimized**
Symbolic header	Yes	Yes	Yes	Yes
File Descriptors	Yes	Yes	Yes	Yes
External Symbols	Yes	Yes	Yes	Yes
External Strings	Yes	Yes	Yes	Yes
Procedure Descriptors	Yes	Yes	Yes	Yes
Line Numbers	No	Yes	Yes	Yes
Relative File Descriptors	No	No	Yes	Yes
Optimization Symbols	No	Partial	Yes	Yes
Local Symbols	No	Partial	Yes	Yes
Local Strings	No	Partial	Yes	Yes
Auxiliary Symbols	No	Partial	Yes	Yes

The minimal level of symbolic information that may be produced during compilation includes only the symbol information required for the linker to function properly. This includes external symbol information that is needed to perform symbol resolution and relocation.

If the limited level of symbolic information is requested, line number entries are generated, as well as external symbol information and procedure descriptors. In addition, local symbols for procedures (and the corresponding auxiliary symbols, optimization symbols, and local strings) are present. Limited symbolic information is sufficient to meet the needs of profiling tools. The information present at this level is a subset of that required for full debugger support.

If full symbolic information is included, all symbol table sections are produced in full. This level enables full debugging support with complete type descriptions for local and external symbols. Optimization is disabled.

Optimized symbolic information is designed to balance the aims of performance and debugging capabilities. This level supplies the same information as the full debugging option, but it also allows all compiler optimizations. As a result, some of the correlation is lost between the source code and the executable program.

On Tru64 UNIX systems, users can choose to compile their programs with any one of the four levels of symbolic information. The options `-g0`, `-g1`, and `-g2` specify increasing levels of symbolic information. The system compiler's default is to produce the minimal level (`-g0`). Currently, debugging of optimized code (`-g3`) is not fully supported. See `cc`(1) for more details.

5.3.1.2 Locally Stripped Images

Objects can be produced with only global symbolic information stored in the symbol table. Selection of the `-x` option causes the linker to create a locally-stripped object. Reasons for stripping local symbolic information include reducing file size and limiting the amount of symbolic information available to end users of an application.

A locally-stripped object is very similar to an object produced with minimal symbolic information (see Section 5.3.1.1). The difference is the consolidation of file descriptors, which the linker does only for locally-stripped objects.

In a locally-stripped image, the file descriptors are included solely for the purpose of identifying source file languages. One file descriptor is present for each source language involved in the compilation. These file descriptors will have their `adr` field set to `addressNil` indicating the file descriptors cannot be used to identify text addresses.

Version Note

The preceding use of `addressNil` is supported in symbol table format **V3.13** and greater. In symbol table formats less than **V3.13**, the file descriptor `adr` value should be ignored.

The procedure descriptor table is present in full but is rearranged to group procedures by source language. All procedure descriptors for procedures written in a particular source language are thus contiguous, and they reflect the file descriptor's information.

External symbols are also present in a locally-stripped image. The file indices (`ifd` field) of the external symbols are updated to identify the generic file descriptor for the appropriate source language. The index fields are set to zero to indicate that no type information is available. External symbols with the storage class `scNil` are removed. These are debugging symbols that are not normally produced for minimal symbol tables.

Limited debugging is possible with locally-stripped objects. Because the procedure descriptors are retained, stack traces are possible. External symbol information

can also be viewed, and language-dependent handling of symbols (for example, C++ name demangling) is preserved.

A linked executable file can be locally stripped at any time after its creation using the command `ostrip -x`. The output is the same as described above. This operation may also alter the raw data of the `.comment` section. See Chapter 7 for details.

5.3.1.3 (Fully) Stripped Images

Executable files may be fully stripped at any time after creation using either the `strip` command or the command `ostrip -s`. Stripping an executable will result in complete removal of the symbol table, including the symbolic header. The file header fields `f_symptr` and `f_nsyms` are set to zero to indicate that the file has been stripped.

This operation may also alter the raw data of the `.comment` section. See Chapter 7 for details.

5.3.2 Source Information

The final executable image for a program bears little resemblance to the source code files from which it was created. One of the principal functions of the symbol table is to track the relationship between the two so that the debugger is able to describe the resulting program in a way that the programmer can recognize.

5.3.2.1 Source Files

Much of the complication of source information stems from the "include" system. When a compilation involves several source files, there may be duplication of the header files included in each source file, or of the source files themselves. To avoid repetition of header file information in the linked object, the linker merges the input objects' included files wherever possible. Compilers mark file descriptors as mergeable or unmergeable. The linker then examines the input file descriptors and performs the merge whenever possible.

The linker considers two file descriptors to be mergeable if all of the following criteria are met:

1. The file descriptor `fMerge` bit is set in both (marked as mergeable by compiler).
2. Files have the same name.
3. Files are written in the same language.
4. Files contain the same number of local and auxiliary symbols.
5. Checksums match.

   The checksums match if either:

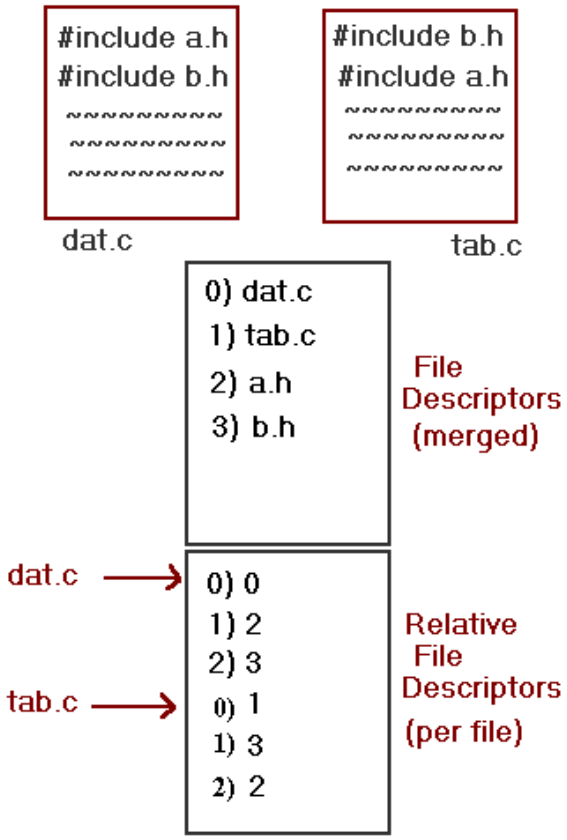
   a. Neither file's first auxiliary record is a `btChecksum`.

   b. Both files' first auxiliary record is a `btChecksum` and they are identical.

The role of the relative file descriptor (RFD) tables is to track file-relative information after merging. A relative file descriptor table entry maps the index of each file at compile time to its index after linking. After linking, local or auxiliary symbols must be accessed through the RFD table to obtain the updated file descriptor index. This mechanism is necessary because the indices in the local symbol table are not updated when files are merged.

Figure 5–4 is an example of the use of the relative file descriptor table.

Figure 5–4: Relative File Descriptor Table Example



For a symbol reference composed of a file index and symbol index (offset within file), the relative file descriptor table is used as follows:

1. To look up given file index in the RFD table to get the updated file index.
2. To look up new file index in the (merged) file descriptor table to get the base of symbols for that file.
3. To add symbol index to file's base to access the symbol entry.

See Section 5.3.7.3 for the representation of relative indices in the auxiliary symbol table.

5.3.2.2 Line Number Information

For a debugger to be effective, a connection must be made between high-level-language statements in source files and the executable machine instructions in object files. Line number entries map executable instructions to source lines. This mapping allows a debugger to present to a programmer the line of source code that corresponds to the code being executed. The line number information is produced by the compiler and should be rewritten if an application such as an instrumentation tool or an optimizer modifies code.

Line number information is emitted in two forms, one found in the line number table and one in the optimization symbol table (see Section 5.3.3).

The line number information found in the optimization symbol table is referred to as ESLI (extended source location information). This is a new form of line number that augments the information in the line number table. ESLI will only be present for procedures that cannot be described accurately by entries in the line number table.

Version Note

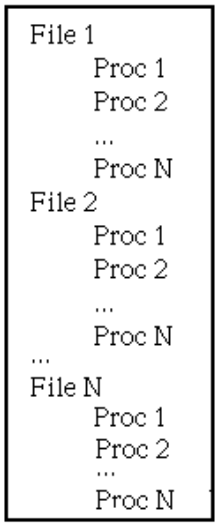
In symbol table formats less than **V3.13** line number information is found exclusively in the line number table.

5.3.2.2.1 The Line Number Table

Line number information is generated for each source file that contributes executable code to a program. Within each source file, line numbers are organized by procedure, in the order of appearance in the file. The line number symbol table section is produced only when a program is compiled with limited or greater symbolic information (see Section 5.3.2.2).

Figure 5–5 illustrates the organization of the line number table.

Figure 5–5: Line Number Table



The order outlined in Figure 5–5 is not guaranteed to match the ordering of file descriptors or procedure descriptors in those tables. The starting offset for a procedure's line table entries can be computed by adding the procedure descriptor's `cbLineOffset` to the containing file descriptor's `cbLineOffset`. The count of line number entries for a specific procedure can only be determined by finding the starting offset of the next procedure's entries in the line number table. This calculation is illustrated by the `proc_pline_count()` function in the packed line number programming example in Section 10.1.

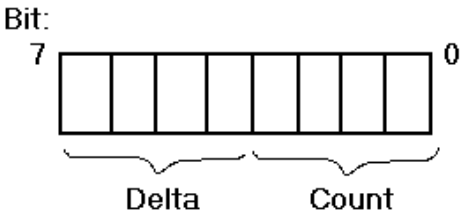
Alternate entry points have a starting line number, but they have no specific ending line number. Procedure descriptors for a procedure and each of its associated alternate entry points share a common end offset in the line number table. See Section 5.3.6.7 for more information on alternate entry points.

The line number table has two forms. The "packed" form is used in the object file. The "expanded" form is a more useful representation to programmers and can be derived algorithmically (or by API) from the packed form.

The packed line numbers are stored as bytes. Each packed entry within the single byte value consists of two parts: count and delta. The count is the number of instructions generated from a source line. The delta is the number of source lines between the current source line and the previous one that generated executable instructions.

Figure 5–6 shows how these two values are represented.

Figure 5–6: Line Number Byte Format

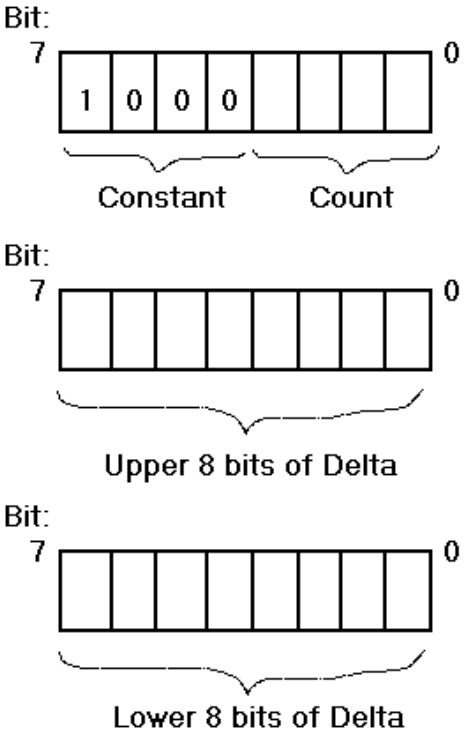


The four-bit count is interpreted as an unsigned value between 1 and 16 (0 means 1, 1 means 2, and so forth). A zero value would be wasted when no instructions are generated for a source line and, as a result, no line number entry will exist for that line.

The four-bit delta is interpreted as a signed value in the range -7 to +7. Code generators may produce instructions that are not in the same order as the corresponding source lines. Therefore, the offset to the "next" source line may be a forwards or backward jump.

Either of these quantities may fall outside the representable range. For a delta outside the range, an extended format exists (as shown in Figure 5–7). This extended format can represent delta values in the range -32768 to 32767. Delta values outside of this range are not representable. This is a permanent restriction of the packed line number format.

Figure 5–7: Line Number 3-Byte Extended Format



For a count outside the range, one or more additional entries follow, with the delta set to zero.

If both fields are out of range, the delta is handled first. An extended-format delta representation is followed by an entry with the delta bits set to zero and the remainder of the count contained in the count value.

The packed line number format can be expanded to produce the instruction-to-source-line mapping that is needed for debugging. A sample program is provided in Section 10.1 to illustrate interpretation of packed line numbers.

The following source listing of a file named `lines.c` provides an example that shows how the compiler assigns line numbers:

```
1   #include <stdio.h>
2   main()
3   {
4       char c;
5
6       printf("this program just prints input\n");
7       for (;;) {
8          if ((c =fgetc(stdin)) != EOF) break;
9       /*   this is a greater than 7-line comment
10           * 1
11           * 2
12           * 3
13           * 4
14           * 5
15           * 6
16           * 7
17           */
18           printf("%c", c);
19      } /* end for */
20  } /* end main */
```

The compiler generates line numbers only for the lines 2, 6, 8, 18, and 20; the other lines are either blank or contain only comments.

Table 5–10 shows the packed entries' interpretation for each source line.

Table 5–10: Line Number Example

Source Line	LINER contents	Interpretation
2	03	Delta 0, count 4
6	44	Delta 4, count 5
8	29	Delta 2, count 10
18 [1]	88 00 0a	Delta 10, count 9
19	10	Delta 1, count 1
20	14	Delta 1, count 5

Table Note:

1. Extended format (delta is greater than 7 lines).

The compiler generates the following instructions for the example program:

```
[lines.c:   2] 0x0:     ldah     gp, 1(t12)
[lines.c:   2] 0x4:     lda      gp, -32592(gp)
[lines.c:   2] 0x8:     lda      sp, -16(sp)
[lines.c:   2] 0xc:     stq      ra, 0(sp)
[lines.c:   6] 0x10:    ldq      a0, -32720(gp)
[lines.c:   6] 0x14:    ldq      t12, -32728(gp)
[lines.c:   6] 0x18:    jsr      ra, (t12), printf
[lines.c:   6] 0x1c:    ldah     gp, 1(ra)
[lines.c:   6] 0x20:    lda      gp, -32620(gp)
[lines.c:   8] 0x24:    ldq      a0, -32736(gp)
[lines.c:   8] 0x28:    ldq      t12, -32744(gp)
[lines.c:   8] 0x2c:    jsr      ra, (t12), fgetc
[lines.c:   8] 0x30:    ldah     gp, 1(ra)
[lines.c:   8] 0x34:    lda      gp, -32640(gp)
[lines.c:   8] 0x38:    and      v0, 0xff, t0
[lines.c:   8] 0x3c:    stq      v0, 8(sp)
[lines.c:   8] 0x40:    xor      t0, 0xff, t0
```

```
[lines.c:   8] 0x44:    bne      t0, 0x6c
[lines.c:  18] 0x48:    ldq      t2, 8(sp)
[lines.c:  18] 0x4c:    sll      t2, 0x38, t2
[lines.c:  18] 0x50:    sra      t2, 0x38, a1
[lines.c:  18] 0x54:    ldq      a0, -32752(gp)
[lines.c:  18] 0x58:    ldq      t12, -32728(gp)
[lines.c:  18] 0x5c:    jsr      ra, (t12), printf
[lines.c:  18] 0x60:    ldah     gp, 1(ra)
[lines.c:  18] 0x64:    lda      gp, -32688(gp)
[lines.c:  19] 0x68:    br       zero, 0x24
[lines.c:  20] 0x6c:    bis      zero, zero, v0
[lines.c:  20] 0x70:    ldq      ra, 0(sp)
[lines.c:  20] 0x74:    lda      sp, 16(sp)
[lines.c:  20] 0x78:    ret      zero, (ra), 1
[lines.c:  20] 0x7c:    call_pal halt
```

After expanding packed line numbers, the following instruction-to-source mapping (formatted `instruction number.source line number`) is produced by `odump` for the `-l` option:

```
 0.    2        1.    2        2.    2
 3.    2        4.    6        5.    6
 6.    6        7.    6        8.    6
 9.    8       10.    8       11.    8
12.    8       13.    8       14.    8
15.    8       16.    8       17.    8
18.   18       19.   18       20.   18
21.   18       22.   18       23.   18
24.   18       25.   18       26.   19
27.   20       28.   20       29.   20
30.   20       31.   20
```

Header files included in an object have no associated line numbers recorded in the symbol table. Line number information for included files containing source code is not supported by the packed line number format. The following section describes a more comprehensive line number representation that includes line number information for header files.

5.3.2.2.2 Extended Source Location Information (ESLI)

Version Note

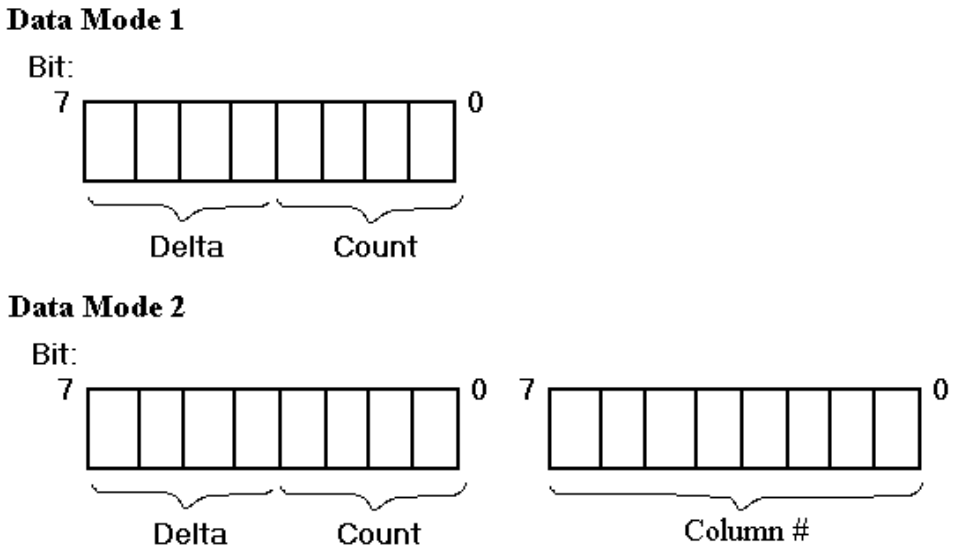
ESLI is supported for symbol table format **V3.13** and greater.

The line number table does not correctly describe optimized code or programs with untraditional source files, resulting in images that are difficult to debug. Extended Source Location Information (ESLI) is intended to provide more information to enable debugging of optimized programs, including PC and line number changes, file transitions, and line and column ranges. ESLI is essentially a superset of the older line number table.

ESLI is stored in the optimization symbols section. This information is accessible on a per-procedure basis from the procedure descriptors. See Section 5.3.3 for more detail on accessing information in the optimization symbols section.

ESLI is a byte stream that can be interpreted in two modes: data mode or command mode. Currently, two formats are defined for data mode. These are designated as "Data Mode 1" and "Data Mode 2". Additional data modes may be defined as needed.

Figure 5–8: ESLI Data Mode Bytes

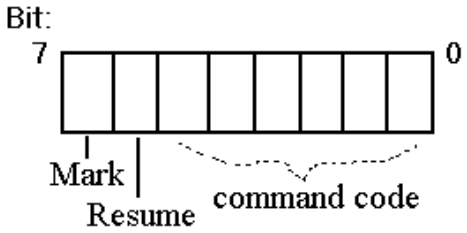


Data Mode 1 is the initial mode for a procedure's ESLI. Data Mode 1 is identical to the packed line number format with the exception of the interpretation of the delta PC escape value `0x80` (which indicates a switch to command mode).

In Data Mode 2, each entry consists of two bytes. The first byte is identical to the encoding and interpretation of Data Mode 1. The second byte is an absolute column number (from 0 to 255), where column number 0 indicates that column information is missing or not meaningful for this entry. The escape from Data Mode 2 to command mode consists of a delta PC escape value set to `0x80` and column number set to 0.

In command mode, each byte is either a command or a command parameter. For a command byte, the low-order six bits are a command code, and the two high bits are used as flags, as shown in Figure 5–9. The "mark" flag, if set, announces that a new state has been established. Several commands may be required to fully describe a new state. The "resume" flag, if set, indicates the end of command mode. The next byte following a command with "resume" set will be a data mode byte. The effective data mode can be changed by `SET_DATA_MODE` commands in command mode, otherwise the data mode that was in effect prior to the escape to command mode will be resumed. See Table 5–11 for a complete list of commands.

Figure 5–9: ESLI Command Byte



Command parameters are stored in LEB (Little Endian Byte) 128 format. See Section 1.4.6 for a description of this data representation. PC deltas are always expressed as machine instruction offsets and must be scaled by the size of a machine instruction before adding to the current PC. No other deltas need to be scaled.

Table 5–11 shows how to interpret the bytes in command mode. These definitions can be found in the system header file `linenum.h`.

Table 5–11: ESLI Commands

Name	Value	Parameters by Type
ADD_PC	1	SLEB
ADD_LINE	2	SLEB
SET_COL	3	LEB
SET_FILE	4	LEB
SET_DATA_MODE	5	LEB
ADD_LINE_PC	6	SLEB, SLEB
ADD_LINE_PC_COL	7	SLEB, SLEB, LEB
SET_LINE	8	LEB
SET_LINE_COL	9	LEB, LEB
SEQUENCE_BREAK	10	SLEB
SET_EXP	11	LEB

ADD_PC
: Parameter is a signed value to add to the current PC value.

ADD_LINE
: Parameter is a signed value to add to the current line number.

SET_COL
: Parameter is an unsigned value that represents a new column number. The column number is used to associate the PC with a particular location within a source line. Column number parameters use a zero-based representation that must be adjusted by adding 1.

SET_FILE
: Parameter is an unsigned value used to switch file context. This command is typically followed by a `set_line` command.

SET_DATA_MODE
: Parameter is an unsigned value used to set the data mode that will be in effect when data mode is resumed. The only parameter values that are currently accepted are `1` and `2`. Additional data modes may be defined in future releases.

ADD_LINE_PC
: Both parameters are signed values. The first is added to the line number and the second is added to the PC.

ADD_LINE_PC_COL
: The first two parameters are signed values and the third is an unsigned value. The first two are added to the line number and PC respectively. The third is used to set the column number.

SET_LINE
: Parameter is an unsigned value that sets the current line number.

SET_LINE_COL
: Both parameters are unsigned values. The first represents the line number and the second represents the column number.

SEQUENCE_BREAK — Indicates the end of a contiguous sequence of address descriptions. The value of the parameter is added to the current address, and the resulting address becomes the starting address of the next sequence of address descriptions. The current file and line number continue to apply as the current values for the new sequence as well. (These can, however, be changed using the appropriate commands.)

Version Note

The SEQUENCE_BREAK command is supported in Tru64 UNIX **V5.1** and greater for symbol table format **V3.13** and greater.

SET_EXP — Set exponent for Tandem edit line numbers. The value of the parameter is an unsigned integer from 0 through 7 representing a power of 10 from -3 through 4.

Version Note

The SET_EXP command is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX.

A tool reading the ESLI must maintain the current PC value, file number, line number, and column. Taken together, these four values represent the current "state". Consumers must also keep track of the mode in effect to interpret the data properly. A sample program is provided in Section 10.2 to illustrate consumption of ESLI.

Data encoded in ESLI can be represented in tabular format. The PC value and file, line, and column numbers can be stored as a state table. The following example shows how to build this state table.

In this example ESLI will record line numbers for a routine that includes text from a header file.

Source listing for line1.c:

```
1   /* ESLI example using included source lines */
2
3   main() {
4      char *msg;
5
6      msg = (char *)0;
7
8   #include "line2.h"
9
10     printf("%s", msg);
11  }
```

Source listing for line2.h

```
1   msg = (char *)malloc(20);
2   /*
3    *
4    *
5    *
6    *
7    *
8    *
9    *
10   */
11  strcpy(msg, "Hello\n");
```

The compiler generates the following instructions for the example program:

```
      main:
[line1.c:   3] 0x1200011d0:     ldah    gp, 8192(t12)
[line1.c:   3] 0x1200011d4:     lda     gp, 28336(gp)
[line1.c:   3] 0x1200011d8:     lda     sp, -16(sp)
[line1.c:   3] 0x1200011dc:     stq     ra, 0(sp)
[line1.c:   3] 0x1200011e0:     stq     s0, 8(sp)
[line1.c:   6] 0x1200011e4:     bis     zero, zero, s0
[line2.h:   1] 0x1200011e8:     bis     zero, 0x14, a0
[line2.h:   1] 0x1200011ec:     ldq     t12, -32560(gp)
[line2.h:   1] 0x1200011f0:     jsr     ra, (t12)
[line2.h:   1] 0x1200011f4:     ldah    gp, 8192(ra)
[line2.h:   1] 0x1200011f8:     lda     gp, 28300(gp)
[line2.h:   1] 0x1200011fc:     bis     zero, v0, s0
[line2.h:  11] 0x120001200:     bis     zero, s0, a0
[line2.h:  11] 0x120001204:     lda     a1, -32768(gp)
[line2.h:  11] 0x120001208:     ldq     t12, -32600(gp)
[line2.h:  11] 0x12000120c:     jsr     ra, (t12)
[line2.h:  11] 0x120001210:     ldah    gp, 8192(ra)
[line2.h:  11] 0x120001214:     lda     gp, 28272(gp)
[line1.c:  10] 0x120001218:     ldq_u   zero, 0(sp)
[line1.c:  10] 0x12000121c:     lda     a0, -32760(gp)
[line1.c:  10] 0x120001220:     bis     zero, s0, a1
[line1.c:  10] 0x120001224:     ldq     t12, -32552(gp)
[line1.c:  10] 0x120001228:     jsr     ra, (t12)
[line1.c:  10] 0x12000122c:     ldah    gp, 8192(gp)
[line1.c:  10] 0x120001230:     lda     gp, 28244(gp)
[line1.c:  11] 0x120001234:     bis     zero, zero, v0
[line1.c:  11] 0x120001238:     ldq     ra, 0(sp)
[line1.c:  11] 0x12000123c:     ldq     s0, 8(sp)
[line1.c:  11] 0x120001240:     lda     sp, 16(sp)
[line1.c:  11] 0x120001244:     ret     zero, (ra)
```

The ESLI and its interpretation for the generated code is shown in the following table.

Table 5–12: ESLI Example

		Command			State			
			(M)ark	(R)esume		(F)ile	(L)ine	(C)olumn
ESLI bytes (hex)	**Mode**	**Code**	**M**	**R**	**PC (hex)**	**F**	**L**	**C**
Initial State (from PDR)	Data1				1200011d0	0	3	0
04	Data1				1200011e4	0	3	0
30	Data1				1200011e8	0	6	0
80	Data1	Escape						
04 01	Cmd	set_file(1)				1		
48 01	Cmd	set_line(1)		R			1	
05	Data1				120001200	1	1	0
80	Data1	Escape						
86 0a 06	Cmd	add_line_pc(10,6)	M		120001218	1	11	0
04 00	Cmd	set_file(0)				0		
48 0a	Cmd	set_line(10)		R			10	
06	Data1				120001234	0	10	0
16	Data1				120001250	0	11	0

The handling of alternate entry points differs from the handling of main entry points. Procedure descriptors for alternate entry points are identified by a PDR.lnHigh value of -1. If the PC for an instruction maps to an alternate entry point, the following steps should be taken:

- Find procedure descriptor for the corresponding main entry. This is accomplished by searching back in the procedure descriptors until a `PDR` is found that is not an alternate entry (`PDR.lnHigh` is not `-1`).
- Access the ESLI for the procedure.
- Read the ESLI until the PC value matches the `PDR.adr` field of the alternate entry's procedure descriptor.

5.3.3 Optimization Symbols

Version Note

Optimization symbols are supported for symbol table format **V3.13**. and greater.

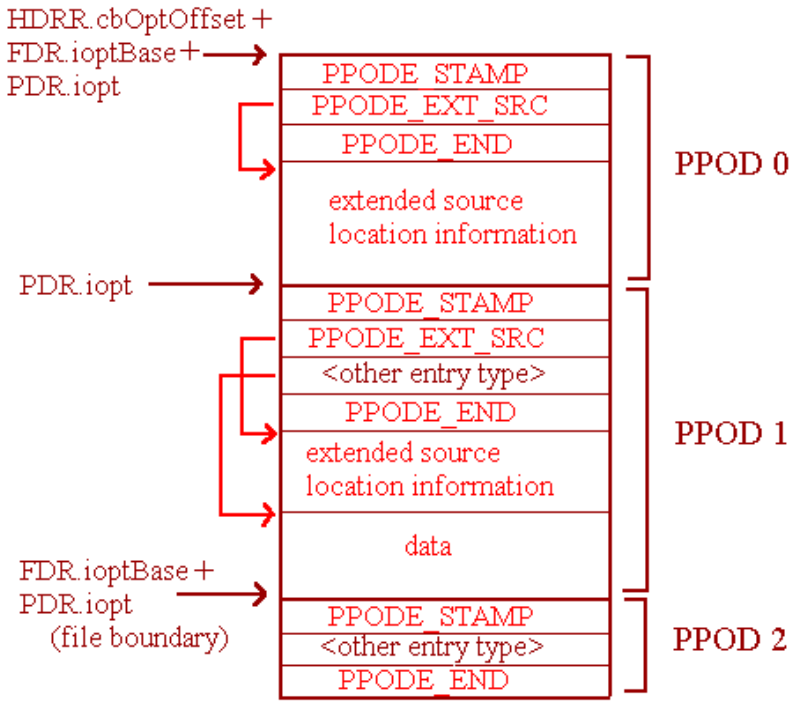
The optimization symbols section gives individual producers and consumers the ability to communicate information about any aspect of the object file, in any form they choose. New information can be generated at any time with minimal coordination between all producers and consumers.

The optimization section is organized on a per-procedure basis. Each procedure descriptor has a pointer to the optimization symbols in the field `PDR.iopt`. If no optimization symbols are associated with the procedure, the field contains `ioptNil`. Otherwise, it contains the index of the first optimization symbol entry for this procedure. Consumers should access the optimization symbols through the procedure descriptors. The optimization section is not present in a locally-stripped object.

This section consists of a sequence of zero or more Per-Procedure Optimization Descriptions (PPODs), as shown in Figure 5–10. Each PPOD's internal structure consists of two parts:

1. A leading sequence of structured entries using a Tag-Length-Value model to describe subsequent raw data. The structure of the PPOD entry can be found in Section 5.2.10.
2. The raw data area.

Figure 5–10: Optimization Symbols Section



This section has the following alignment requirements:

- Octaword (16-byte) alignment of the beginning of the section.
- Octaword (16-byte) alignment of the beginning of the raw data area.
- Octaword (16-byte) alignment of each PPOD.

Object file producers must produce either an empty optimization symbols section or a valid one. An empty one has the symbolic header fields `cbOptOffset` and `ioptMax` set to zero. If an optimization section is present, but a particular file does not contribute to it, the file descriptor field `copt` is set to zero. In this case, all procedure descriptors belonging to the file must have their `iopt` fields set to `ioptNil`.

Tools that both read and write object files must consume a valid optimization symbols section (if present in the input file) and produce an equivalent and valid section in its output file. If a tool does not know how to process the section contents, the section must be omitted from the output file. If a tool does know how to process portions of the optimization symbols, those portions may be modified and the rest should be removed. The linker concatenates input optimization symbols sections into one output section without reading or modifying any of the entries.

The format and flexible nature of this section are similar by design to the `.comment` section. The structures are the same size and contain the same fields (with different names), and the rules of navigation are the same. The primary difference is that the optimization section contains procedure-specific information; whereas, the comment section contains object-specific information.

5.3.4 Run-Time Information

The symbol table contains information that debuggers must interpret to find symbols at run time. This section describes the information that the static symbol table structures provides. Algorithms for determining run-time symbol addresses are included.

5.3.4.1 Procedure Addresses

The following pseudocode describes an algorithm for determining the procedure start address:

```
if (HDRR.vstamp >= 0x30D || PDR.isym == isymNil)
    return(PDR.adr)
else
    foreach FDR in HDRR
        foreach PDR in FDR
            if PDR matches
                if (FDR.csym == 0)  /* Use external symbol */
                    return (EXTR[PDR.isym].asym.value)
                else                /* Use local symbol */
                    return (SYMR[FDR.isymbase + PDR.isym].value)
```

If local symbol information is present for the given `PDR`, the `isym` field identifies the local symbol table entry that contains the start address of the procedure. If no local symbol information is present, the `isym` field identifies the external symbol table entry containing the start address of the procedure. If no symbol information is present for the `PDR`, the `isym` field is set to `isymNil` and the `adr` field will contain a reliable start address.

Version Note

The `PDR.adr` field is reliably updated by the linker for symbol table format **V3.13**. The preceding algorithm is recommended for determining procedure addresses in symbol table formats less than **V3.13**.

5.3.4.2 Stack Frames

A stack frame is a run-time memory structure that is created whenever a procedure is called. The *Calling Standard for Alpha Systems* specifies the stack frame format and related code requirements. This section explains how to interpret procedure descriptor fields related to the stack frame.

Two types of stack frames are supported: fixed-size frames and variable-size frames. The variable frame format is used for procedures that dynamically allocate memory and for those with very large frames. Figure 5–11 shows a fixed-size frame and Figure 5–12 shows a variable-sized frame.

From the procedure descriptor, you can determine which type of stack frame the procedure has. The field `PDR.framereg` stores the frame pointer register number. If this field has a value of 30 ($sp), the stack frame is a fixed-size frame. If it has a value of 15 ($fp), the stack frame is a variable-size frame.

Figure 5–11: Fixed-Size Stack Frame

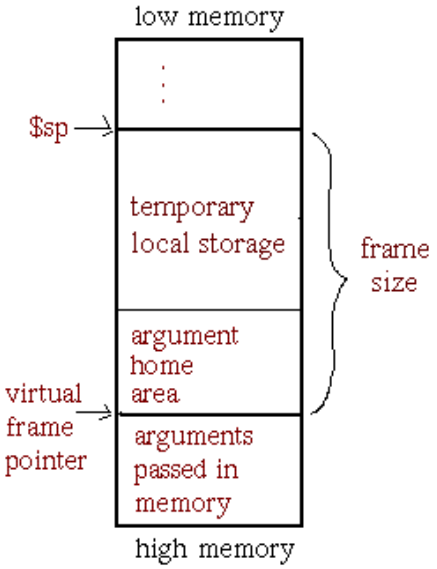
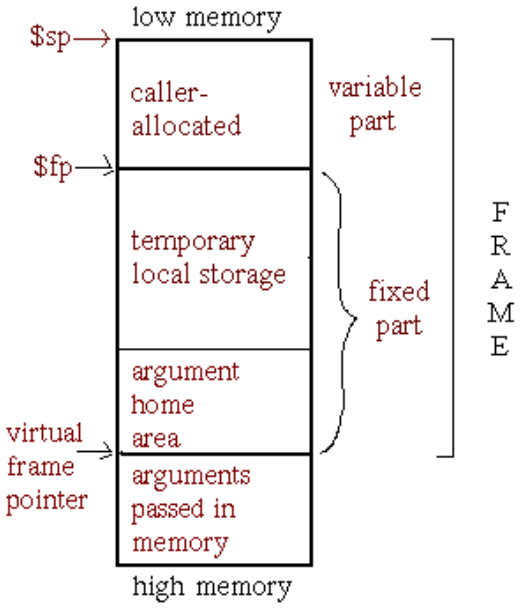


Figure 5–12: Variable-Size Stack Frame



For both types of stack frames, the value of `PDR.frameoffset` is the size of the fixed part of the stack frame. In the case of a fixed-size frame, it is the entire frame size. For a variable-sized frame, the entire frame size cannot be determined from the symbol table. The code may dynamically increase and decrease the size of the frame multiple times during procedure execution.

The virtual frame pointer represents the contents of the frame pointer register at procedure entry, prior to prologue execution. The (real) frame pointer is the contents of the frame pointer register after prologue execution. The difference

between the virtual and real frame pointer values is the fixed frame size, which is subtracted from the $sp contents during the procedure prologue. Note that stack offsets recorded in the symbol table are relative to the virtual frame pointer, not the real value used at run time.

The contents of the frame pointer register at are used at run time as the base address for accessing data, such as parameters and local variables, on the stack. See Section 5.3.4.3 for details.

5.3.4.3 Local Symbol Addresses

Local variables and parameters may be stored in registers or on the stack. Those stored in registers (identified by a storage class of `scRegister`) do not have addresses. For local variables and parameters with addresses, this section explains how to calculate their run-time locations from the symbol table information.

To calculate the run-time address for a local variable (`stLocal`) based on its symbol table value:

```
Frame pointer - PDR.localoff + SYMR.value
```

To calculate the run-time address for a parameter (`stParam`) based on its symbol table value:

```
Frame pointer - argument_home_area_size + SYMR.value
```

The argument home area is a portion of the stack frame designated for parameter storage. See Figure 5–11 for an illustration. For historical reasons, the size of this area is always 48 bytes.

The calculations above must be performed at run time when the actual frame pointer value is known. Note that the value becomes valid only after the procedure prologue has executed.

To calculate the locations based on static information, convert the symbol's value to an offset from the real frame pointer:

Local:

```
PDR.frameoffset - PDR.localoff + SYMR.value
```

Parameter:

```
PDR.frameoffset - 48 + SYMR.value
```

The resulting offsets are always positive values because the frame pointer contains the address of the lowest memory in the fixed part of the stack frame at run time.

5.3.4.4 Uplevel Links

Version Note

Uplevel links are supported in symbol table format **V3.13** and greater.

An uplevel link is the real frame pointer of an ancestor of a nested routine. The routine nesting may be a feature of the language (such as Pascal), or the nesting may occur in optimized code which has been decomposed for parallel execution into smaller routines. Uplevel links provide debuggers a method of finding all local symbols associated with the ancestor routine.

When a procedure is passed a static link, that static link will be represented within the scope of the procedure definition as a local automatic symbol with a special

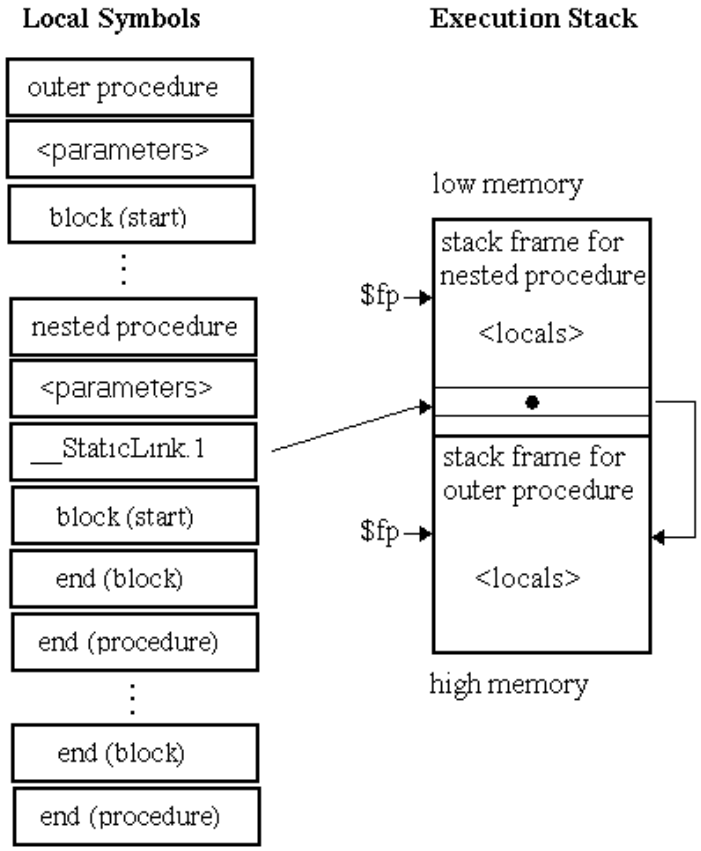
name beginning with "__StaticLink.". The lifetime of this symbol begins after the procedure prologue has been executed.

The static link symbol will occur between the procedure's parameter definitions and the first stBlock symbol.

The full name of the symbol will be "__StaticLink." followed by a positive decimal integer with no leading zeros. This integer value identifies the number of levels up the ancestor tree the static link points to.

For example, if the name is "__StaticLink.3" it will contain the static link of the procedure in which it is defined, and that procedure's static link points to a stack frame that is three levels up in the procedure's ancestor tree, the great-grandfather of the procedure.

Figure 5–13: Representation of Uplevel Reference



Debuggers of Tru64 UNIX object files need to use the uplevel link information to determine which symbols are visible at a location in the program and to compute the addresses of local symbols in ancestor routines. When the debugger needs the current value or address of a name that might be defined as an uplevel reference, two separate actions may be required: finding the procedure that defines the currently visible instance of that name, and finding the address of the currently visible instance of that name. If only type information is required, finding the procedure that defines the name may be sufficient.

Finding the defining procedure is accomplished by repeatedly looking up the name in the local symbol table of a chain of procedures that extends from the

current procedure through its chain of ancestors until either the name is found in a procedure or the end of the chain of ancestors is reached without finding the name. If this search terminates without finding the name, the debugger should conclude that the name is not visible by uplevel reference at the current location in the program.

When searching for the desired procedure, the debugger should count how many levels in the ancestor chain were traversed before finding the name. If zero levels were traversed, the name is defined within the current procedure and is not an uplevel reference. The number of levels traversed is assumed to be in the variable `LevelsToGo` in the algorithm below.

Finding the address for the name involves locating static link values and dereferencing them with appropriate offsets. Basically, while the number of levels to be traversed is greater than zero, find the static link symbol for the current level and obtain its value. Finally, add the desired symbol's offset from the real frame pointer to the final static link value.

The recommended algorithm for finding the address is as follows:

```
LevelsToGo = <from name lookup above>
NewProc = CurrentProcedure
NewFrame = FramePointerValue(CurrentProcedure)
Failed = false
while (LevelsToGo > 0 && !Failed)
    StaticLink = FindStaticLinkSym(NewProc)
    if (StaticLink == NULL)
        Failed = true
    else
        NewFrame = *(NewFrame + StaticLink->symbol.offset)
        Levels = StaticLinkLevels(StaticLink)
        LevelsToGo = LevelsToGo - Levels
        for (; Levels > 0; Levels--)
            NewProc = NewProc->proc.parent
```

if `Failed` is true after executing this algorithm, required information about static links is missing in the symbol table, and an error has occurred. If `LevelsToGo` ends up less than zero, the optimizer's static link optimization has eliminated a static link level that would be needed to compute the address of the name. It is recommended that debuggers inform the user that optimization prevents the debugger from computing the address of the name.

If `Failed` is false and `LevelsToGo` is equal to zero, the address for the currently visible instance of the name is `NewFrame` plus the offset of the name with respect to the real frame pointer for `NewProc`.

The function `StaticLinkLevels` returns the integer at the end of the name for the indicated static link symbol.

5.3.4.5 Finding Thread Local Storage (TLS) Symbols

This section explains how to interpret symbolic information for TLS symbols (identified by a storage class of `scTlsData` or `scTlsBss`). See Section 3.3.9 or the *Programmer's Guide* for general information on TLS.

A TLS symbol's value contains its offset from the start of the TLS region for that object. This offset can be used at process execution time to determine the address of the TLS symbol for a particular thread.

A debugger can calculate TLS symbol addresses by looking up the address of the TLS region using run-time structures and adding the offset of the TLS symbol to that address. The following formula can be used to calculate TLS symbol addresses.

```
TLS sym address = *(TEB.TSD + __tlskey) + SYMR.value
```

A detailed description of this formula follows:

1. Get the address of the Thread Environment Block (TEB).
2. Get the address of the Thread Specific Data (TSD) array from the TEB structure.
3. Get the offset of the TLS pointer in the TSD array.

   This offset is normally stored in a `.lita` or `.got` entry. This value should be accessed using the symbol `__tlskey` . In spite of the fact that `__tlskey` is a label symbol, no ampersand is used in this context because the value that the label points to is being retrieved. The address of `__tlskey` will need to be adjusted by the address mapping displacement in the same manner that the debugger adjusts addresses of text and data symbols.

   For static executables, the `.lita` entry contains the constant offset (2048). This offset identifies the first and only TSD slot (256) that will be allocated for the TLS pointer.

   For shared objects, the `.got` entry labeled by `__tlskey` is initially 0, indicating that the TSD slot has not been allocated yet. After the object's initialization routines have run, a TSD key will be allocated and the `.got` entry will contain its offset.
4. Get the TLS pointer value. The TLS pointer is a 64-bit address set to the start of the TLS Region.
5. Calculate the address of the TLS symbol by adding the offset of the TLS symbol to the TLS pointer value.

TLS common symbols (`scTlsCommon`) should not occur in linked objects, so debuggers should not need to support them. Executables and shared libraries can only reference TLS symbols that they define, so successfully linked objects should have not TLS undefined or TLS common symbols.

5.3.5 Profile Feedback Data

Version Note

Profile feedback data is supported in symbol table format **V3.13** and greater.

Profile feedback data is stored in entries in the optimization symbols table with tag type `PPODE_PROFILE_INFO`. The data contained in this section is intended for Compaq internal use only. It contains execution profiling feedback used by compilers and the `om` utility.

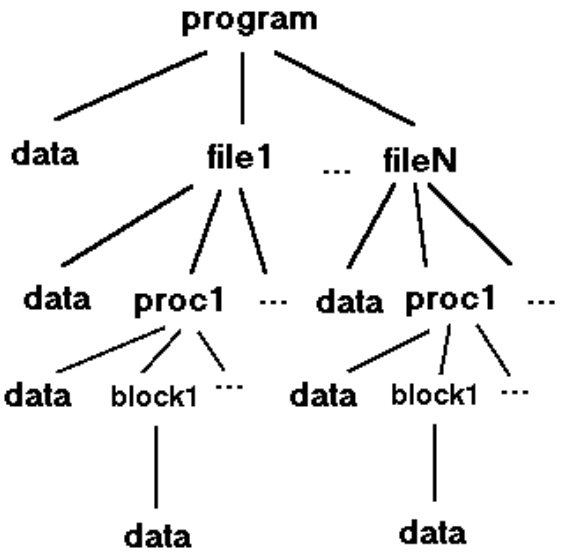
Profile feedback data contains relative file descriptor and local symbol table indexes. If an object tool removes, adds, or rearranges relative file descriptors or local symbol table entries it must also remove all optimization symbol table entries including the profile feedback data.

5.3.6 Scopes

From a user-program's point of view, an identifer's scope determines its visibility in different parts of the program. Programming languages provide facilities for declaring and defining names of procedures, variables and other program components inside various scoping levels. This section briefly discusses the concept of scope and then explains how it is represented in the symbol table. References are made to structures in the auxiliary symbol table; see Section 5.3.7.3 for details.

Generally speaking, the four main scoping levels in a program are block scope, procedure scope, file scope, and program scope. Most programming languages have constructs to implement at least these scoping levels. Figure 5–14 shows the hierarchy of these scopes.

Figure 5–14: Basic Scopes



Names with block scope can only be referenced inside the declaring block. Blocks are delimited by begin and end markers, the syntax of which varies among languages.

Names with procedure scope are only recognized inside their enclosing subroutines. For instance, the names of formal parameters and local variables declared inside a procedure are accessible only to that procedure's executable statements.

Names with file scope can be referenced by any instruction within the file where they are declared. A file can be composed of procedures and data external to any procedure. Both external data names and procedure names can have file scope or program scope. Note that in a compilation involving only a single file or in a compilation for a programming language with no separate-compilation facilities, file scope and program scope are equivalent.

Names with program scope are visible everywhere in the program, even when the executable program is built from many source and header files. The linker must resolve these names or pass them to the dynamic loader to resolve. See Section 5.3.10 for more information about symbol resolution.

In the symbol table, procedure scope, file scope, and program scope correspond to local, static, and global symbols, respectively. Block scope names are also local symbols. Local and static symbols appear in the local symbol table, and global symbols are in the external symbol table.

5.3.6.1 Procedure Scope

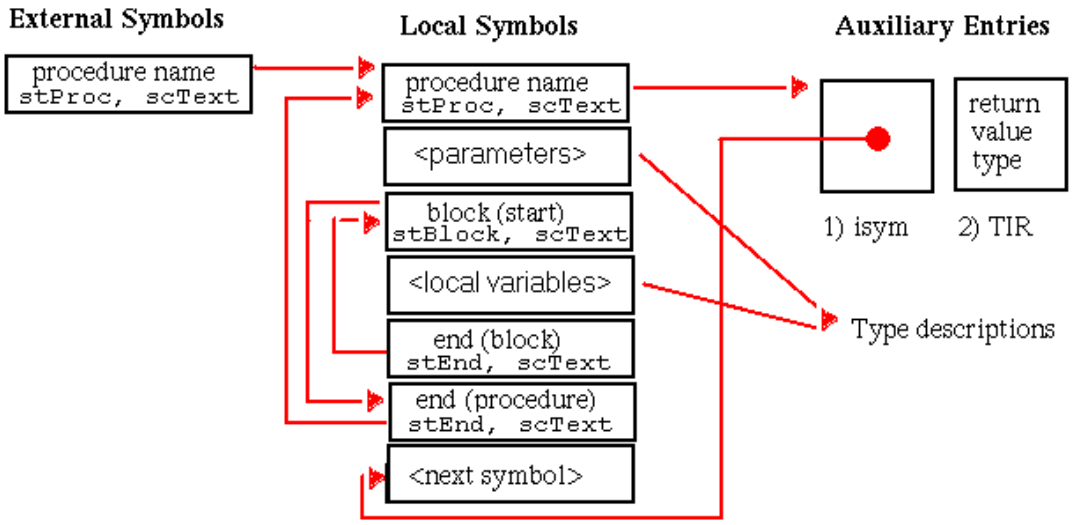
Although procedure symbols can only be global or static (with symbol types `stProc` and `stStaticProc`, respectively), procedure entries appear in the local symbol table to identify the containing scope of their local data. The set of symbols appearing in the local symbol table to describe a procedure scope and their associated auxiliary entries is shown in Figure 5–15. Global procedures also have

entries in the external symbol table. As illustrated, the indices of these external entries point to the scoping entries in the local symbol table.

Note

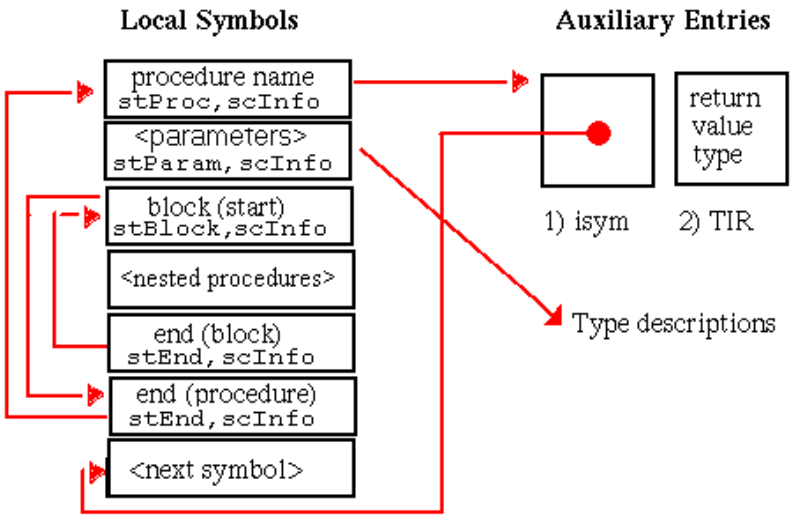
In this chapter, all diagrams of symbol table representations use arrows to show that one entry contains an index to another entry. For external and local symbol table entries, the index used is contained in the `index` field. For auxiliary symbols, the `isym` or `RNDXR` field is the index used. Any exceptions to this general rule are noted in the diagrams.

Figure 5–15: Procedure Representation



A special instance of a procedure definition occurs for a procedure with no text. This type of procedure occurs only in the local symbol table and is very similar to the representation of other procedures. It is generally used for procedures that have been optimized away that still need to be represented for debugging or profiling information.

Figure 5–16: Procedure with No Text



A procedure with no code can contain only nested procedures that also have no code associated with them. If a procedure with no code does not contain any nested procedures, the `stBlock`/`stEnd` symbol pair can be omitted from the representation.

The `stProc` symbol included in this representation is distinguished from similar `stProc` symbols by its value field that is set to `addressNil (-1)`.

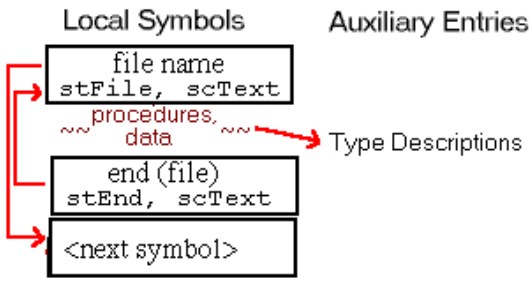
Version Note

Procedures with no code are supported in symbol table format **V3.13** and greater.

5.3.6.2 File Scope

As in the case of procedures, file name entries appear in the local symbol table to define the file's scope. This representation is shown in Figure 5–17. Note that file symbols appear in the local symbol table only.

Figure 5–17: File Representation

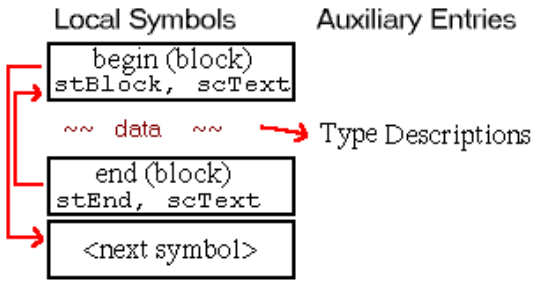


5.3.6.3 Block Scope

In general, the local symbol table denotes scoping levels with `stBlock` and `stEnd` pairs, as shown in Figure 5–18.

All symbols contained between these two entries belong to the scope they describe. Nested blocks are possible, and `stEnd` symbols match the most recent occurrences of `stBlock` (or other opening symbol entries such as `stProc` or `stTag`).

Figure 5–18: Block Representation



Block scopes occur in many languages. In C, they take the form of lexical blocks. In C++, declarations can occur anywhere in the code. In Pascal and Ada, nested procedures are possible, with local variables at any or all levels.

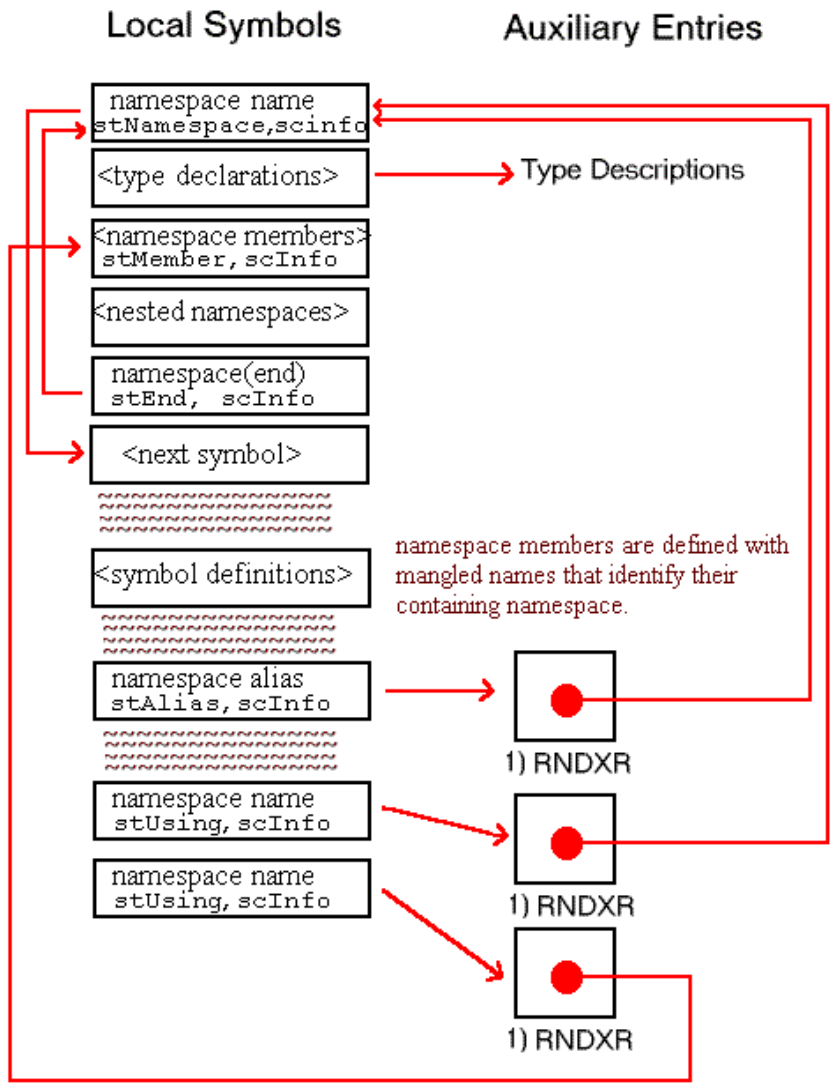
5.3.6.4 Namespaces (C++)

Version Note

Namespaces are supported in symbol table format **V3.13** and greater.

A C++ namespace is a mechanism that allows the partitioning of the program global name space. This partitioning is intended to reduce name clashing and provide greater program manageability to C++ developers.

Figure 5–19: C++ Namespace Representation



A namespace definition may exist only at the global scope or within another namespace. The namespace representation in Figure 5–19 shows a single contribution to a namespace. This representation may be replicated many times in the symbol table for a single namespace. A namespace definition may be continued within the same file or over multiple source files.

A single namespace contribution that spans multiple source files is represented as if it were contained entirely within the source file in which it began.

Namespaces may be aliased, allowing a single namespace to be referred to by multiple names. Namespace components may also be referenced without their namespace qualification if they are included within a scope by a using directive or using declaration. The representations of namespace aliases, using directives, and using declarations are shown in Figure 5–19. Namespace definitions, namespace component declarations, namespace aliases, using directives, and using declarations occur only in the local symbol table. Namespace component definitions may occur in the local or external symbol table.

5.3.6.4.1 Namespace Components

The components of a namespace are represented in two parts: declarations and definitions. Namespace components that do not require definition must be declared in the namespace definition. Namespace components that are referenced by a using declaration must be declared in the namespace definition. All other namespace component declarations may be omitted from the namespace definition.

Namespace component names are mangled only as needed. Function and data definitions have mangled name definitions in the local or external symbol table. These entries are mangled for type-safe linkage and as a method of matching components with the namespaces to which they belong. Names of component declarations within a namespace definition may or may not be mangled. They are not required to include the namespace name in their mangled form.

Empty namespace contributions can be omitted, but at least one instance of a namespace definition must occur somewhere in the local symbol table. This definition is required because name mangling rules do not distinguish namespace component definitions from class member definitions.

5.3.6.4.2 Namespace Aliases

Namespace aliases can occur in namespace, file, procedure, or block scope in the local symbol table. The index value for the `stAlias` entry is an auxiliary table index. The auxiliary entry is a `RNDXR` record containing the local symbol table index of the `stNamespace` symbol in the first instance of a namespace definition within a compilation unit. For an alias of an alias, the `RNDXR` record can also contain the index of another `stAlias` symbol in the local symbol table. Section 9.2.5 provides an example of a namespace alias.

The `stAlias` symbol type may be used in future versions of the symbol table format as a general purpose symbol alias representation. The semantic interpretation of the `stAlias` symbol depends on the type of the symbol it aliases.

5.3.6.4.3 Unnamed Namespace

An unnamed namespace can be declared at the global scope or within another namespace. An unnamed namespace is unique within a compilation unit. Multiple contributions to a unique unnamed namespace are not allowed. Unnamed namespace contributions are included in the non-mergeable portion of a C++ header file.

Unnamed namespace components are subject to the same rules as named namespaces for declarations and definitions.

The `stNamespace` symbol for an unnamed namespace has a compiler generated name starting with `__N1`. This same name is used to identify the unnamed namespace in the mangled names of components of that namespace. (See the unnamed namespace example in Section 9.2.4.)

5.3.6.4.4 Usage of Namespaces

A C++ using directive or a using declaration is represented by a symbol of type `stUsing`. It may occur in any scope in the local symbol table. The index value for the `stUsing` entry is an auxiliary table index. If the `stUsing` entry represents a using declaration for a single namespace component, the auxiliary entry is a `RNDXR` record containing the local symbol table index of a namespace component declaration. If the `stUsing` entry represents a using directive, its `RNDXR` auxiliary contains the local symbol table index of the `stNamespace` symbol in the first definition of that namespace in the compilation unit.

A using directive for a namespace alias is represented with a RNDXR auxiliary that directly references the aliased namespace. This representation contains no record of the alias referenced by the using directive.

Names are not required for stUsing entries, but they can be set to match the namespace or namespace component to which they refer.

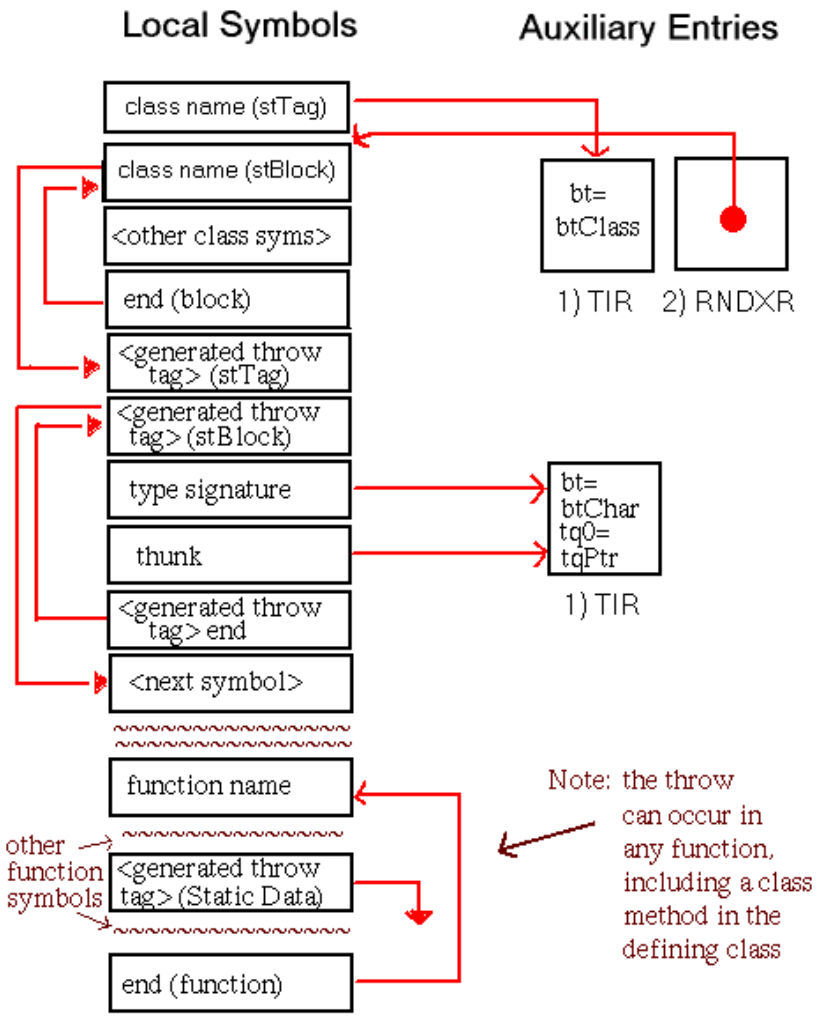
Namespace components that are referenced by an stUsing symbol must be declared in the namespace definition.

Section 9.2.3 provides an example of namespace definitions and uses.

5.3.6.5 Exception Handling Blocks (C++)

In C++, a special scoping mechanism is introduced to expand user-defined exception-handling capabilities. Exception handlers are defined to "catch" exceptions that are "thrown" by other functions. The symbol table must contain sufficient information to recognize the scope of a handler. The compiler generates special symbols to identify where exception handlers are valid.

Figure 5–20: C++ Exception Handler Representation



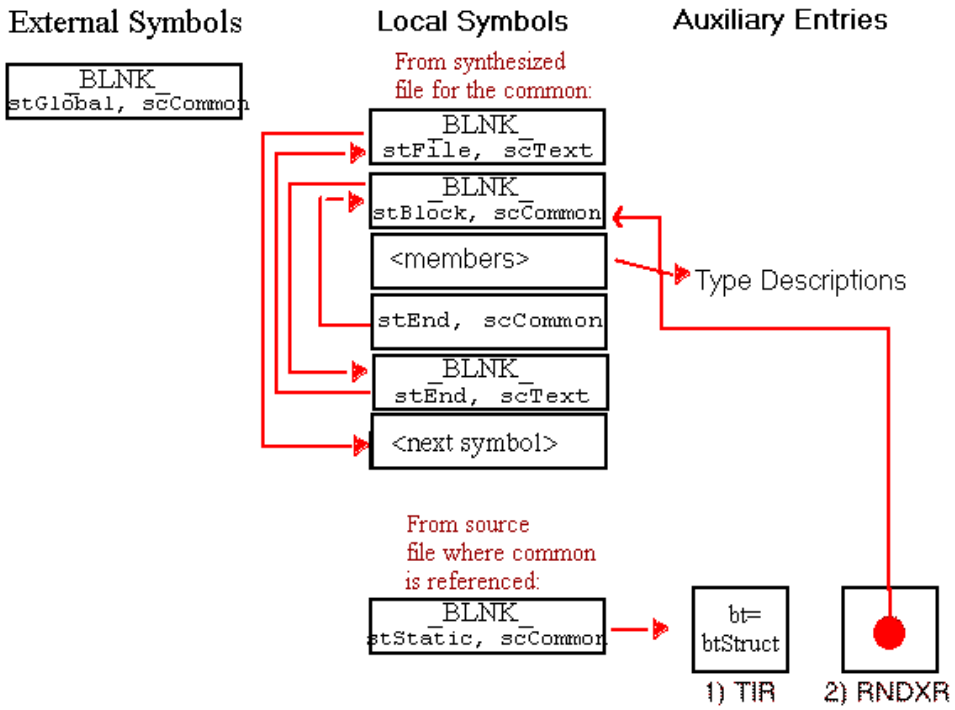
5.3.6.6 Fortran Common Blocks

Fortran common blocks constitute another scoping level. Fortran uses common blocks as a way of specifying data that is global or shared between program units.

A common block is global storage that can be named, allocated, accessed, and used by various subroutines. The block can be named or unnamed; unnamed blocks are known as "blank commons". Internal to the symbol table, blank commons are named `_BLNK__`.

Figure 5–21 shows the symbolic representation of Fortran common blocks.

Figure 5–21: Fortran Common Block Representation



Because a Fortran common is represented as a synthesized file, it also has an entry in the file descriptor table. Furthermore, a global symbol with the same name is also present in the external symbol table.

An example of a Fortran common block can be found in Section 9.3.1.

5.3.6.7 Alternate Entry Points

Fortran also has a facility for creating alternate entry points in procedures. An alternate entry point is represented using an `stProc`/`scText` symbol. In the procedure descriptor table, an alternate entry point is identified by a `lnHigh` field with a value of -1. Procedure descriptors for alternate entry points follow the procedure descriptor for the primary entry point. In the local symbol table, an alternate entry point has an entry inside the scope of the procedure's primary entry.

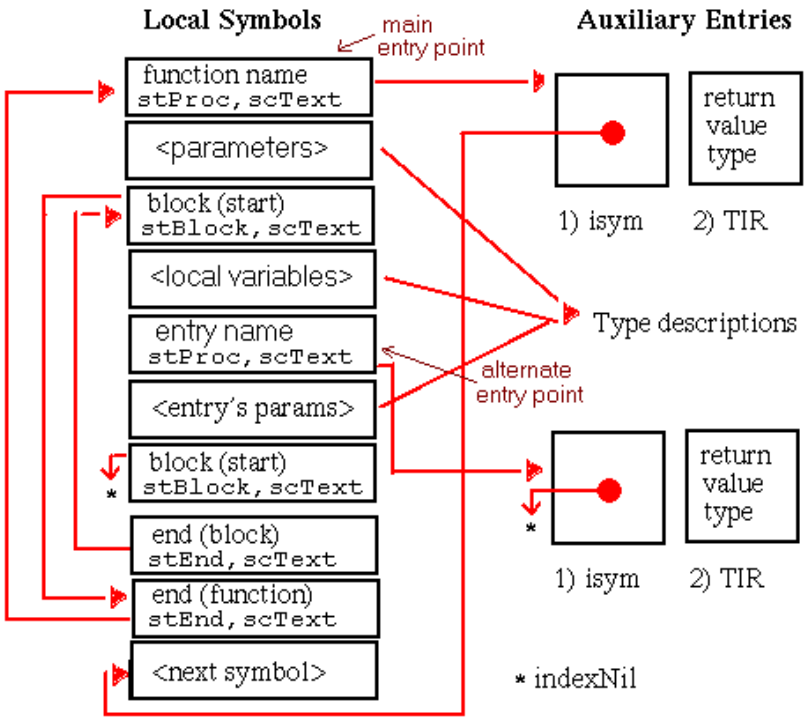
The representation of a procedure with an alternate entry point is shown in Figure 5–22

Version Note

The `stBlock` symbol that follows the alternate entry's `stProc` symbol in Figure 5–22 is supported in symbol table format **V3.13** and greater.

In symbol table formats less than **V3.13** alternate entries do not have a start block symbol, and their prologue size is unknown.

Figure 5–22: Alternate Entry Point Representation



An example of Fortran alternate entries can be found in Section 9.3.2.

5.3.7 Data Types in the Symbol Table

A data element's type dictates its size and interpretation in a programming environment. One of the symbol table's most important tasks is to represent data types in a compact and complete manner.

Type information is stored in the local and auxiliary symbol tables. This section provides guidelines for understanding the type information plus specific examples for depicting a range of types.

5.3.7.1 Basic Types

All programming languages have a set of simple types that are built into the language and from which other data types can be derived. Examples of simple types are integer, character, and floating point. Languages also provide constructs for creating user-defined types based on the simple types. For example, a C++ class can be built using any simple type or previously defined user-defined type and the language facility for declaring classes.

Similarly, a basic type in the symbol table is a building block from which each language constructs its type information. Basic type (`bt`) values directly represent many of the simple types for supported languages; for instance, the value `btChar` indicates a character. Other `bt` values represent language constructs for building

aggregate types; a value of btStruct may be used, for example, to represent a C structure or Pascal record.

The symbol table uses approximately forty basic type values. The interpretation of some of these values is language dependent. See Table 5–5 for a list of all values.

5.3.7.2 Type Qualifiers

Type qualifiers can be applied to basic types to create other data types. Examples are "pointer to", "array of", and "function returning". Generally the number and order of type qualifiers is unrestricted.

See Table 5–6 for a list of type qualifiers and their meanings.

5.3.7.3 Interpreting Type Descriptions in the Auxiliary Table

This section explains in detail the encoding of type descriptions in the symbol table. To fully describe the type of a symbol, the auxiliary symbol table must be created and referenced. Compilation with full symbolic information (-g option on system compilers) results in the creation of this table.

To correctly decode the type information, proceed sequentially, beginning with the symbol table entry. Several fields may be required from other symbol table structures:

- symbol type (st)
- storage class (sc)
- index (SYMR.index)
- value (SYMR.value)
- source language (FDR.lang)

The first step is to determine whether the symbol contains an index of an auxiliary table description.

Table 5–13: Symbols with Auxiliary Type Descriptions

Symbol Type	Storage Class	Conditions	AUXU Index Field
stGlobal	Any	None	index
stStatic	Any	None	index
stParam	Any	None	index
stLocal	Any	Local symbol table	index
stProc	Any	Local symbol table	index
stBlock	scInfo	Inside an scVariant block	value
stMember	scInfo	None	index
stTypedef	scInfo	None	index
stStaticProc	Any	Local symbol table	index
stConstant	Any	None	index
stBase	scInfo	None	index
stVirtBase	scInfo	None	index
stTag	scInfo	None	index
stInter	scInfo	None	index
stNamespace	scInfo	None	index

Table 5–13: Symbols with Auxiliary Type Descriptions (cont.)

Symbol Type	Storage Class	Conditions	AUXU Index Field
stUsing	scInfo	None	index
stAlias	scInfo	None	index

If the index does represent a record in the auxiliary symbol table, the interpretation of the first auxiliary entry (AUXU) depends on the type of the symbol:

- If the symbol's type is stProc or stStaticProc and the symbol is a local symbol, the indexed AUXU is an isym (set to indexNil for alternate entry points) and the second AUXU is a TIR. External procedure symbols do not have descriptions in the auxiliary table.
- If the symbol's type is stInter, stAlias, or stUsing, the indexed AUXU is an RNDXR and the type description does not contain a TIR.
- If the symbol is an stBlock symbol inside an scVariant block, the symbol entry's value field is an index into the auxiliary table. This special case is the only one where the value is used as an auxiliary symbol pointer. In all other cases, it is the index field that potentially indexes the auxiliary table type description.
- Otherwise, the indexed AUXU is a TIR.

The next task is to examine the contents of the TIR. The TIR contains constants representing the basic type of the symbol and up to six type qualifiers, labeled tq0-tq5. If a type has more than one qualifier, they are ordered from lowest to highest. Lower qualifiers are applied to the basic type before higher qualifiers. All unused tq fields are set to tqNil, and no tqNil fields are present before or between other type qualifiers.

In addition to the basic type and type qualifiers, the TIR contains two flags: an fBitfield flag to mark whether the size of the type is explicitly recorded, and a continued flag to indicate that the type description is continued in another TIR. If fBitfield is set, the TIR is immediately followed by a width entry. If more than six type qualifiers are required for the current definition, the description is continued, and the continued flag is set. If exactly six type qualifiers are needed, all six fields are used and the continued flag is cleared.

To illustrate, consider the type "array of pointers to integers". The basic type is "integer" and has two qualifiers, "array of" and "pointer to". Each element of the array is a "pointer to integer". Therefore, the qualifier "pointer to" must be applied first to the basic type "integer". In this example, the qualifier "pointer to" is lower than the qualifier "array of". The contents of the TIR are as follows:

```
bt: btInt
tq0: tqPtr
tq1: tqArray
tq2: tqNil
tq3: tqNil
tq4: tqNil
tq5: tqNil
continued: 0
fBitfield: 0
```

The contents of the TIR dictate how to interpret any subsequent records. The records appear in a prescribed order:

- If the fBitfield flag is set, a width record follows the TIR.
- If the basic type is btPicture, the next four records contain integer values: the string table index of the picture string, the length, precision and scale.
- If the basic type is btScaledBin, the next three records contain integer values: a basic type, the precision and scale.

- If the basic type field is btStruct, btUnion, btEnum, btClass, btIndirect, btSet, btTypedef, btRange, btRange_64, btDecimal, btFixedBin, or btProc, the next record is an RNDXR.
- If the rfd field of the RNDXR contains the value ST_RFDESCAPE, the next record is an isym.
- If the basic type is btRange, the next two records are dnLow and dnHigh.
- If the basic type is btRange_64, the next two records are dnLow records and the two after that are dnHigh records.
- If the basic type is btDecimal or btFixedBin, the next two records contain integer values: the precision and scale.
- For each array type qualifier in the TIR, the following symbols occur:
  - An RNDXR, again possibly followed by an isym
  - Either one or two dnLow records (depending on whether the array is tqArray or tqArray_64)
  - Either one or two dnHigh records (depending on whether the array is tqArray or tqArray_64)
  - Either one or two width records (depending on whether the array is tqArray or tqArray_64)
- If the continued flag is set, the next record is another TIR

For a type description containing more than one TIR, the fields of all TIR records are interpreted in the same way. When a TIR is reached with the flag cleared and any records associated with that TIR have been decoded, the type description is complete.

As an example, consider an array of structures with the fBitfield flag set. A total of seven auxiliary records can be used to describe the type:

1. The TIR with a basic type of btStruct and with tq0 set to tqArray.
2. A width record. The size of the basic type.
3. A RNDXR record. A pointer to the structure definition in the local symbol table.
4. A RNDXR record. A pointer to the array index type description elsewhere in the auxiliary table.
5. A dnlow record. The lower bound of the array's range.
6. A dnhigh record. The upper bound of the array's range.
7. A width record. The distance in bits between each element in the array.

If the continued flag of the TIR is cleared, the width record corresponding to the array qualifier is the final AUXU for this type description.

For another view of this process, see Figure 5–23. Each box represents one auxiliary entry belonging to the symbol's type description. Using the flowchart, an ordered list of entries can be assembled.

Figure 5–23: Auxiliary Table Interpretation

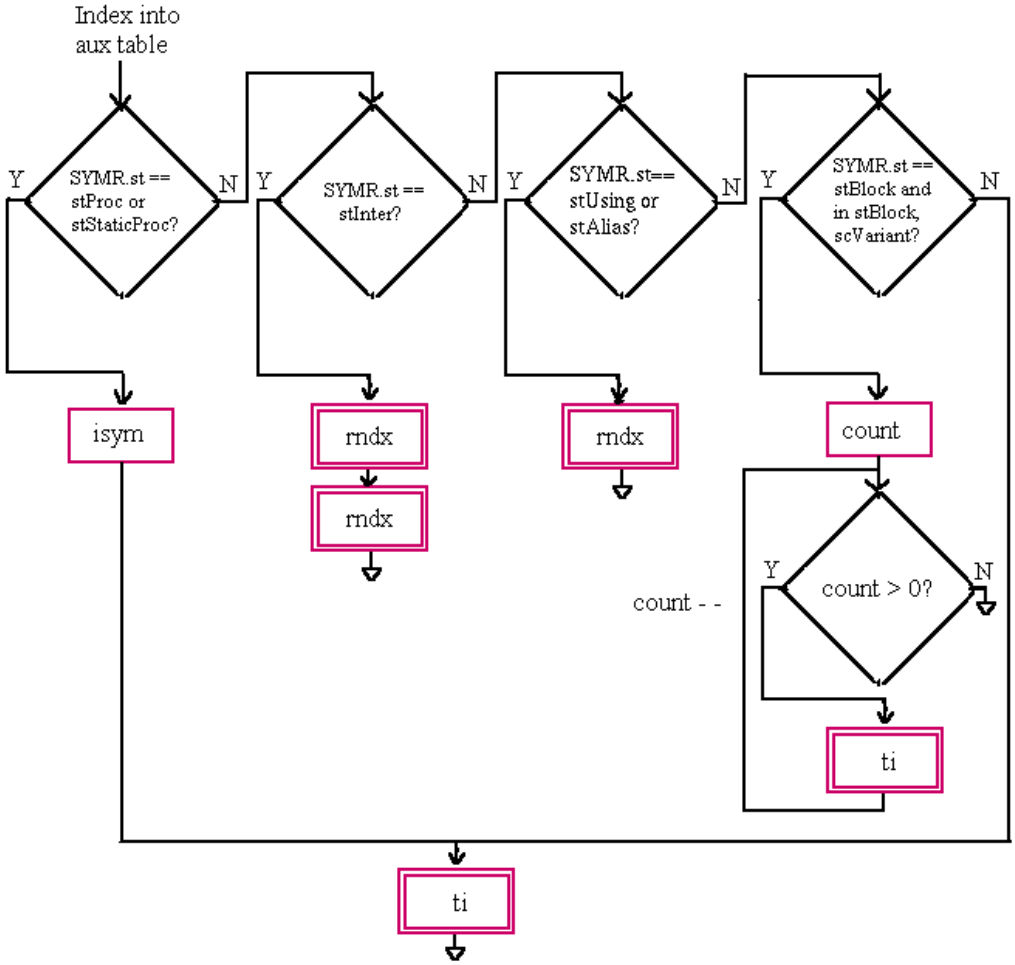


Figure 5–24: Auxiliary Table "ti" Interpretation

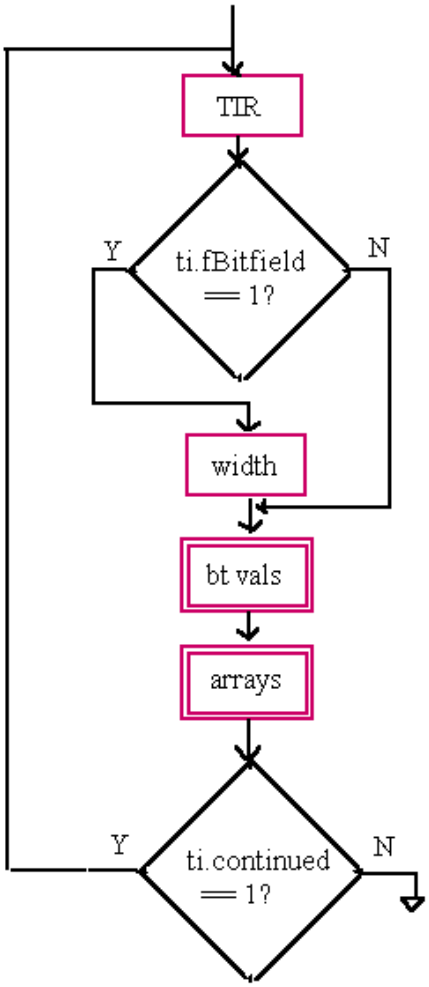


Figure 5–25: Auxiliary Table "bt vals" Interpretation

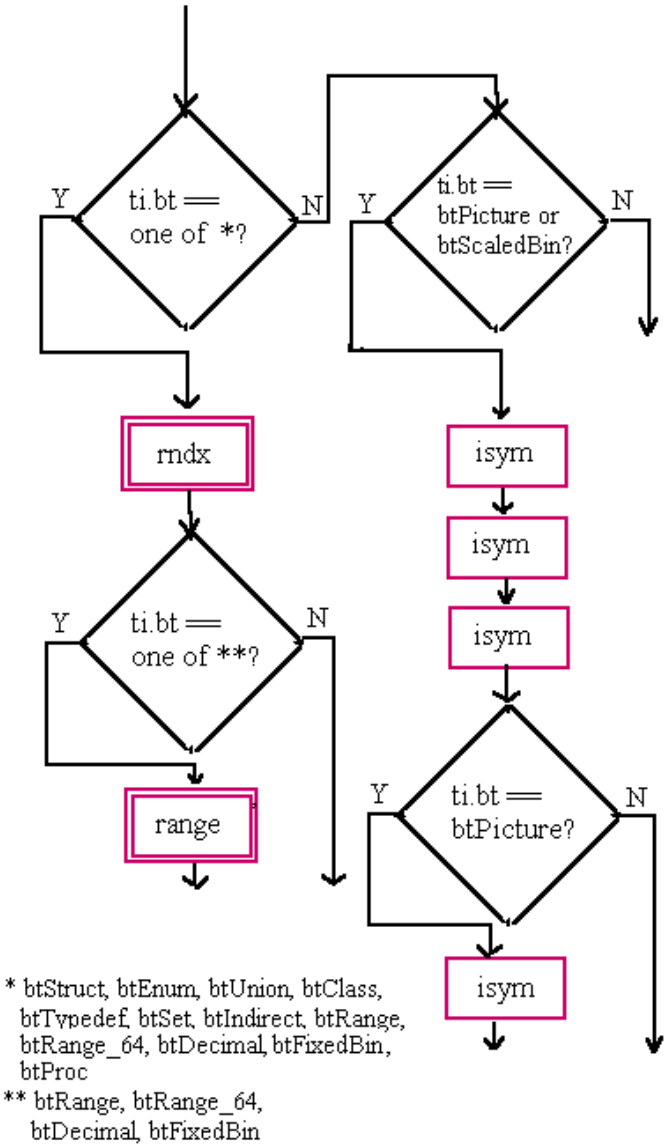


Figure 5–26: Auxiliary Table "arrays" Interpretation

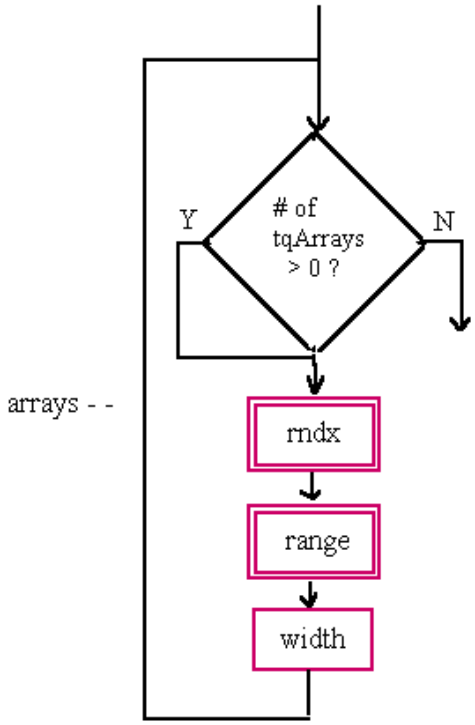


Figure 5–27: Auxiliary Table Range Interpretation

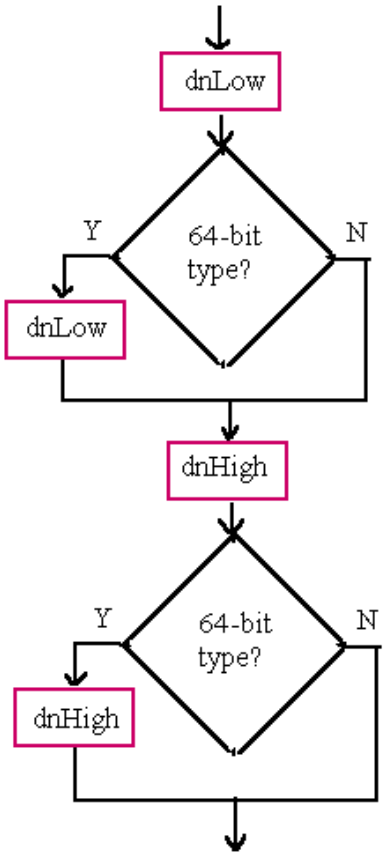
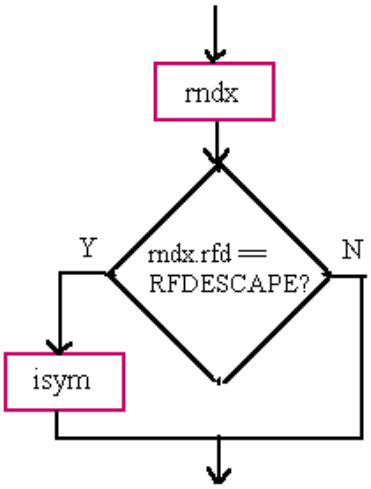


Figure 5–28: Auxiliary Table `RNDXR` Interpretation



The final step is to decode the `RNDXR` records. The basic types that are followed by `RNDXR` records require reference to another local or auxiliary symbol to complete the type description. Interpret the `RNDXR` records as follows:

- If the basic type is `btStruct`,`btUnion`, `btEnum`, `btClass`, `btProc`, or `btTypedef`, the `index` field of the `RNDXR` points into the local symbol table. The specified local symbol is the start of the definition of the structure, union, enumeration, class, or user-defined type. For `btProc`, the referenced local symbol is the start of the set of symbols defining the procedure's signature.

- If the basic type is btSet, the RNDXR points into the auxiliary symbol table. The specified record is the start of the description of the type of each element in the set.
- If the basic type is btIndirect, the RNDXR points into the auxiliary symbol table. The specified auxiliary record is the start of the description of the referenced type.
- If the basic type is btRange, the RNDXR points into the auxiliary symbol table. The specified auxiliary record is the start of the description of the type being subranged.
- If the basic type is btFixedBin, the rfd field of the RNDXR contains a Boolean value. If rfd is true, the base is decimal; if rfd is false, the base is binary. The index field represents a type code.
- If the basic type is btDecimal, the rfd field of the RNDXR contains the value 1 for 4-bit digits (packed decimal) or 2 for 8-bit digits (zoned decimal). The index field represents a type code.

Additionally, the index of every RNDXR used as a pointer must be mapped through the relative file descriptor table (see Section 5.3.2.1), if the table exists. The rfd field of the record controls this mapping. The following algorithm can be used to locate the symbol referenced by the relative index record:

```
if (RNDXR.rfd == ST_RFDESCAPE)
    RFD = (++AUXU).isym
else
    RFD = RNDXR.rfd
if (HDRR.crfd) /* RFD table exists */
    IFD = (current FDR's RFD table)[RFD]
else
    IFD = RFD

if (SYMR needed)
    SYMBASE = FDR[IFD].isymBase
    SYMR = SYMBASE[RNDXR.index]
else if (AUXU needed)
    AUXBASE = FDR[IFD].iauxBase
    AUXU = AUXBASE[RNDXR.index]
```

5.3.8 Individual Type Representations

This section provides sketches of type representations in the local and auxiliary symbol tables. The connections between the two tables is depicted for each type. This form of representation is only possible when full symbolic information is present.

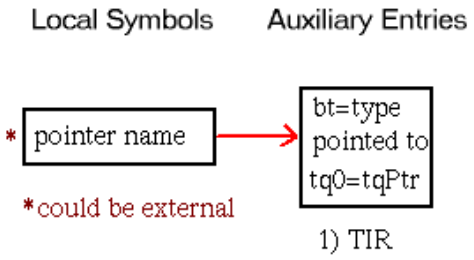
Note that external symbols as well as local symbols reference the auxiliary table, although the examples in this chapter use local symbols only.

5.3.8.1 Pointer Type

A pointer is a variable containing the address of another variable. A pointer is represented by a tqPtr type qualifier modifying another type. A pointer is represented by a single symbol with an entry in the auxiliary table, as shown in Figure 5–29.

Note that if the pointer referenced a user-defined type, such as a class or structure, the TIR would be followed by an RNDXR (and possibly an isym).

Figure 5–29: Pointer Representation



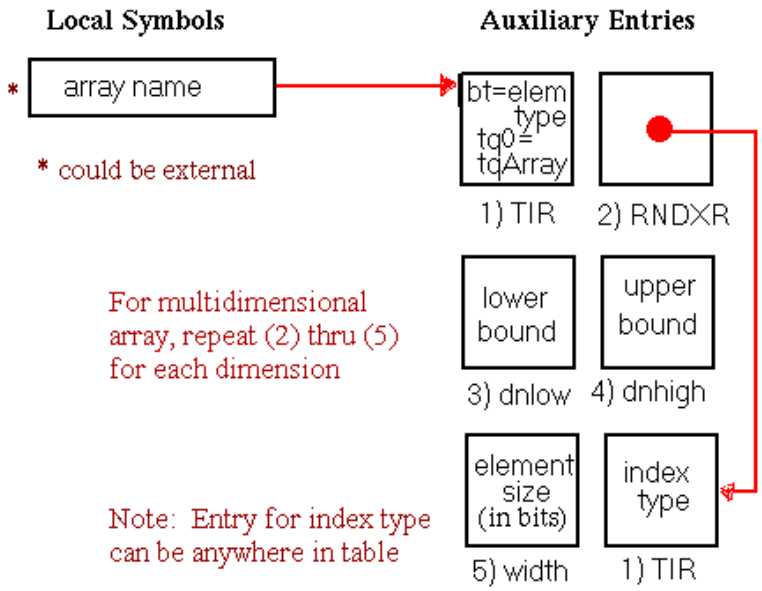
The combination of type qualifiers `tqFar` and `tqPtr` are used to represent a short (32-bit) pointer. This pointer type is used with the XTASO emulation.

5.3.8.2 Array Type

An array is a list of elements that all have the same type. Arrays may be fixed size and allocated at compile time or dynamically sized and allocated at run time. This section describes the fixed-size array symbol table representation. For information on Fortran dynamic arrays, see Section 5.3.8.9. For conformant arrays in Pascal and Ada, see Section 5.3.8.10.

An array is represented by a `tqArray` or `tqArray_64` type qualifier applied to another type. This second type describes the type of all elements in the array. In the local or external symbol table, a single entry represents an array. Figure 5–30 shows the symbol table description for an array.

Figure 5–30: Array Representation



Note that for an array of elements of a user-defined type, such as a class or structure, another `RNDXR` (and possibly an `isym`) would be inserted between the `TIR` and the `RNDXR` describing the subscript type.

If an array has multiple dimensions, the symbols describing the dimension appear in the order of innermost to outermost. For example, the following declaration produces a `TIR` with the `tqArray` qualifier followed by the `RNDXR` and range description for 0-1 followed by the entries for the dimension 0-99:

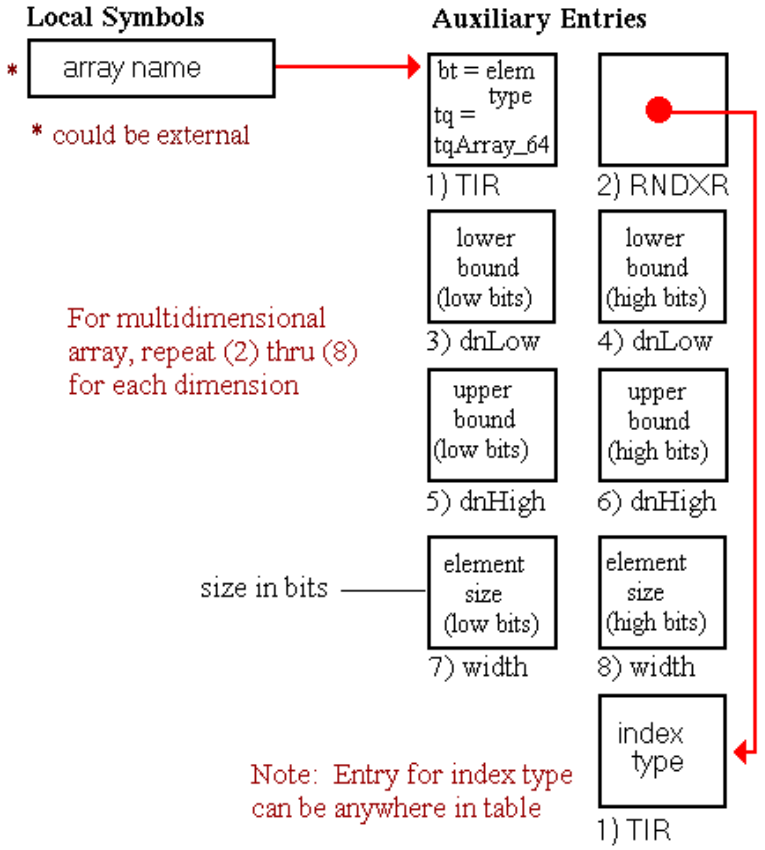
```
float floattable[100][2]
```

Some arrays may have dimensions too large to represent in the 32-bit format shown in Figure 5–30. Such arrays are represented using a 64-bit format in which two auxiliary entries are used for the dimension bounds and size. Figure 5–31 illustrates the 64-bit representation.

Version Note

The 64-bit representation of arrays is supported in symbol table format **V3.13** and greater.

Figure 5–31: 64-Bit Array Representation



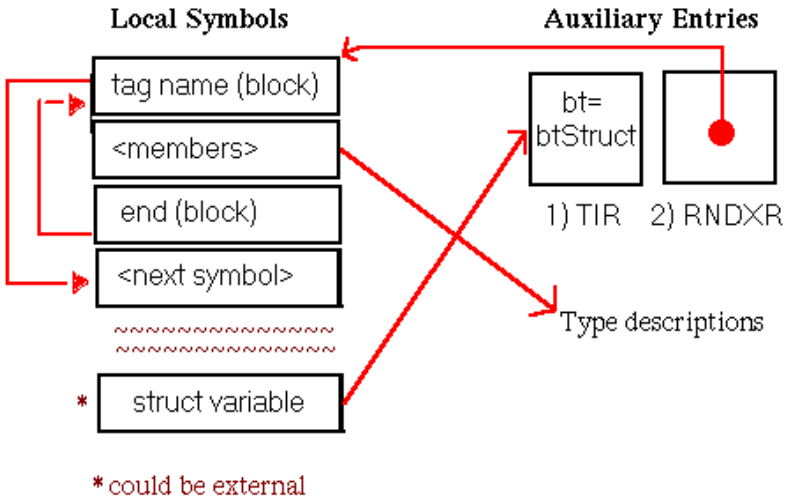
5.3.8.3 Structure, Union, and Enumerated Types

This section applies to data structures in languages other than C++. For the C++ structure, union, or enumerated type representation, see Section 5.3.8.6.

Structures, unions, and enumerated types have a common representation. All three are identified using "tags" and contain zero or more fields. In the symbol table, the tag is the name associated with the starting `stBlock` symbol for the structure's set of local symbols. Note that it may be empty because the tag is optional. Symbols for fields follow. The definition is completed by a block-end symbol matching the block-start symbol.

Figure 5–32 contains a graphical depiction of this set of symbols.

Figure 5–32: Structure Representation



The structure members have auxiliary table indices pointing to their type descriptions.

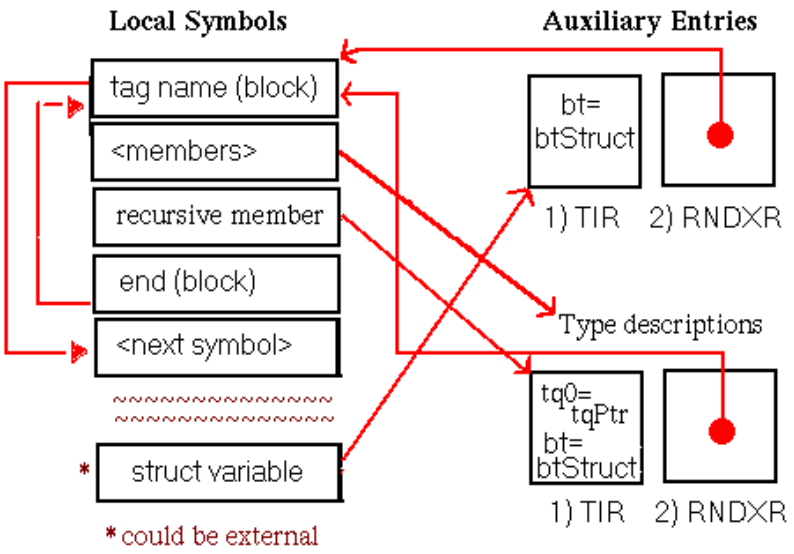
Untagged structures and unions are represented with a NULL tag name. Unnamed structures can be embedded in other structures and are represented as a NULL-named member of the outer structure. See Section 9.1.1 for an example of an unnamed structure.

Version Note

Unnamed member structures are supported in symbol table format **V3.13** and greater. As of Tru64 UNIX **V5.1** dbx will display structures with unnamed member structures, but neither dbx nor ladebug provide specific access to members of unnamed member structures.

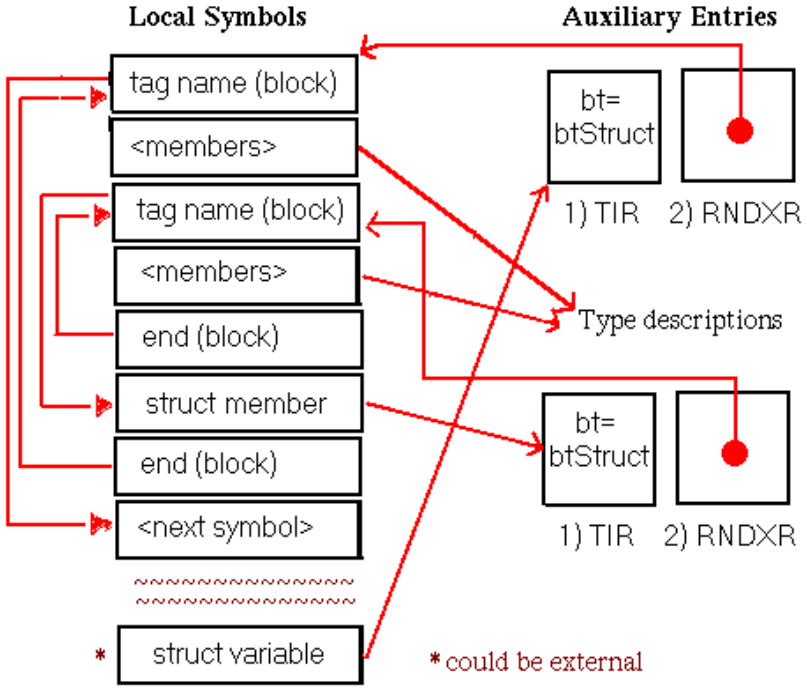
A structure can contain a field that is a pointer to itself. This field is represented by an `stMember` symbol with an auxiliary table entry that references the beginning of the structure's block of local symbols, as shown in Figure 5–33.

Figure 5–33: Recursive Structure Representation



When a field within a structure is itself a structure, the compiler may choose to generate the structure definitions either sequentially or embedded, as shown in Figure 5–34.

Figure 5–34: Nested Structure Representation



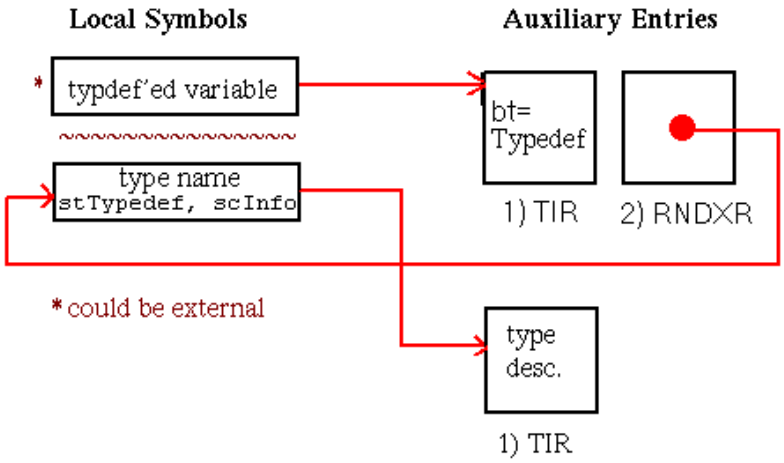
The following declaration might result in the nested structure representation:

```
struct line {
        struct point {
            float x, y;
        }  p1, p2;
};
```

5.3.8.4 Typedef Type

Most languages allow programmers to choose alternate names, or aliases, for data types. The alias created by such a facility (such as C's typedef) is represented as a single local symbol entry that has a pointer to its type description in the auxiliary table. The auxiliary entry contains a pointer to the definition of the type name, as shown in Figure 5–35.

Figure 5–35: Typedef Representation



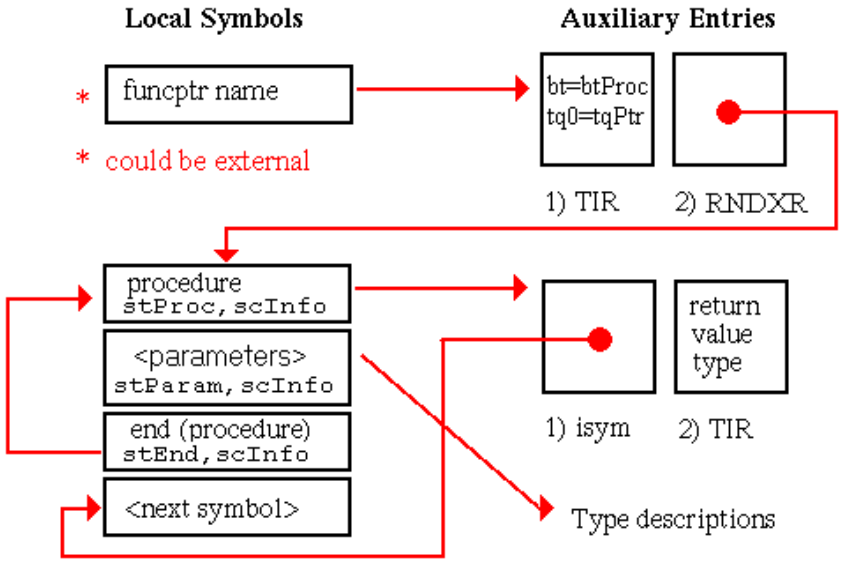
5.3.8.5 Function Pointer Type

Version Note

The following function pointer representation is the preferred representation for symbol table format **V3.13** and greater.

Languages such as C and C++, which allow pointers to functions, represent the type of the function pointer using a special `stProc`/`scInfo` block describing the parameters and return value for the function as shown in Figure 5–36.

Figure 5–36: Function Pointer Representation



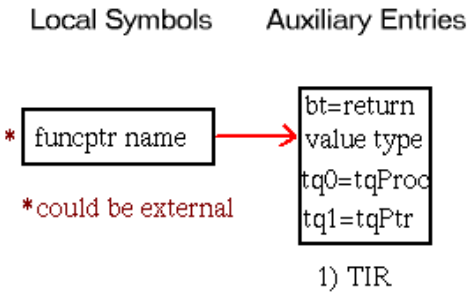
The `stProc/scInfo` entry has its value set to `-2`, which distinguishes it from similar entries used to represent procedures with no text and C++ member functions. The `stProc/scInfo` and `stEnd/scInfo` entries have null names in the function pointer representation. The parameters are optional and may or may not be named.

Version Note

For symbol table formats less than **V3.13** the preceding representation for function pointers is not supported, and the following alternate representation is used exclusively.

An alternate representation of function pointers is shown in Figure 5–37. This representation describes the return type of the function pointer but not its parameters, and it is valid for all symbol table format versions. The combination of type qualifiers `tqPtr` and `tqProc` is interpreted as "pointer to function returning". The function return type may be the base type (`bt`) in the `TIR` or it may be constructed from the base type augmented by additional type qualifiers.

Figure 5–37: Function Pointer Alternate Representation



5.3.8.6 Class Type (C++)

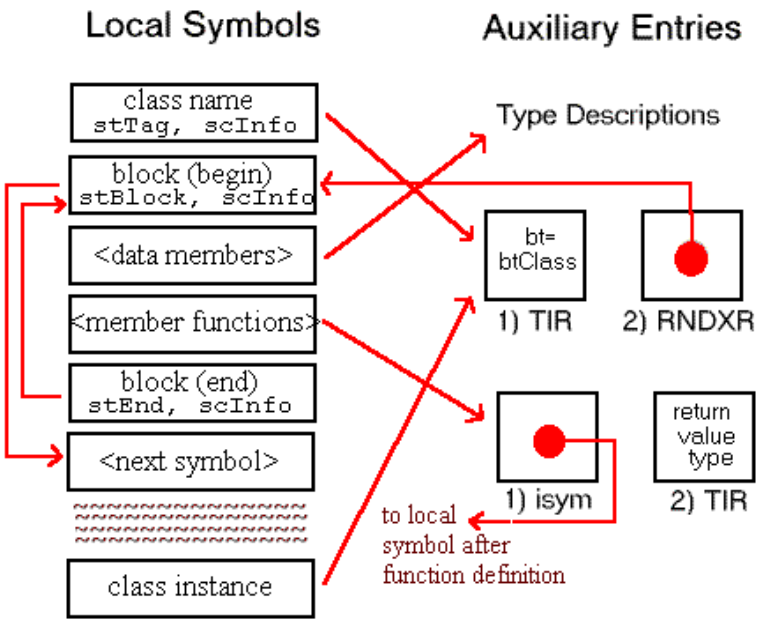
A C++ class resembles an extended C structure. One major distinction is that class fields (referred to as "members") can be functions as well as variables. The set of symbols created for a class is organized as follows:

- The name of the class
- A block symbol for scoping
- Data members
- Symbols associated with member functions. Each member function is represented by the normal set of symbols present for a function.
- Corresponding end symbols that denote the completion of the block and class.

Another characteristic of classes is that symbols are defined implicitly. For example, all classes have an `operator=` operator-overloading function included in the class definition and a `this` pointer to its own type as a parameter to all member functions. These symbols are always included explicitly in the symbol table description.

Figure 5–38 is a graphical representation of the set of symbols for a class.

Figure 5–38: Class Representation



Class members, including member functions, have auxiliary references that point to their type descriptions. Note that member functions are represented as prototypes. The set of symbols defining the member function is elsewhere in the symbol table. To locate the definition of a member function, a name lookup can be performed using the mangled name of the member function with its class name qualifier. See Section 5.3.10.3 for information on name mangling.

C++ structures, unions, and enumerated types are represented the same way as classes. The different data structures are distinguished by basic type value.

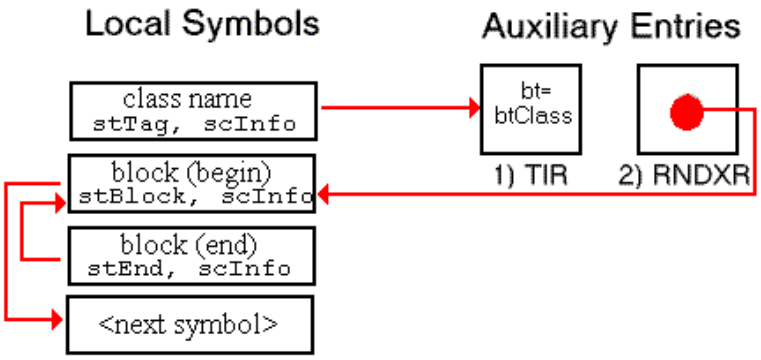
The symbol table does not represent class member access attributes.

Examples of base and derived classes can be found in Section 9.2.1.

5.3.8.6.1 Empty Class or Structure (C++)

The representation of an empty class in C++ is shown in Figure 5–39. Empty structures in C++ are represented in a similar manner with the `TIR.bt` set to `btStruct`.

Figure 5–39: Empty Class or Structure (C++)



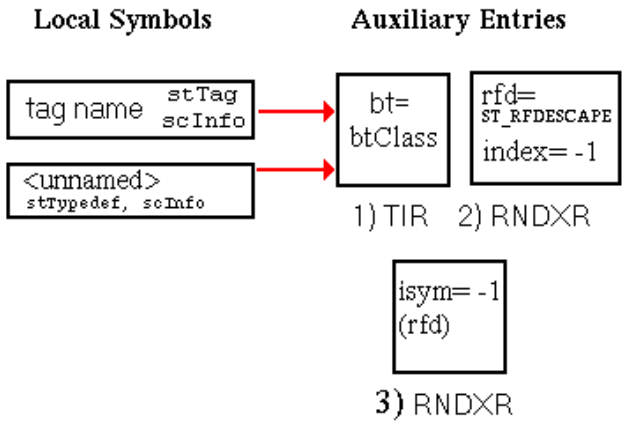
Version Note

This empty class or structure representation is supported in Tru64 UNIX **V5.1**. Prior to Tru64 UNIX **V5.1**, the default compilers did not distinguish empty classes and structures from opaque classes and structures. See Section 5.3.8.6.2 for more details.

5.3.8.6.2 Opaque Class or Structure (C++)

Opaque classes and structures are incomplete types. They have no member information, and they are distinguished from empty classes and structures that have no members. The representation of an opaque class in C++ is shown in Figure 5–40. Opaque structures in C++ are represented in a similar manner with `TIR.bt` set to `btStruct`.

Figure 5–40: Opaque Class or Structure (C++)



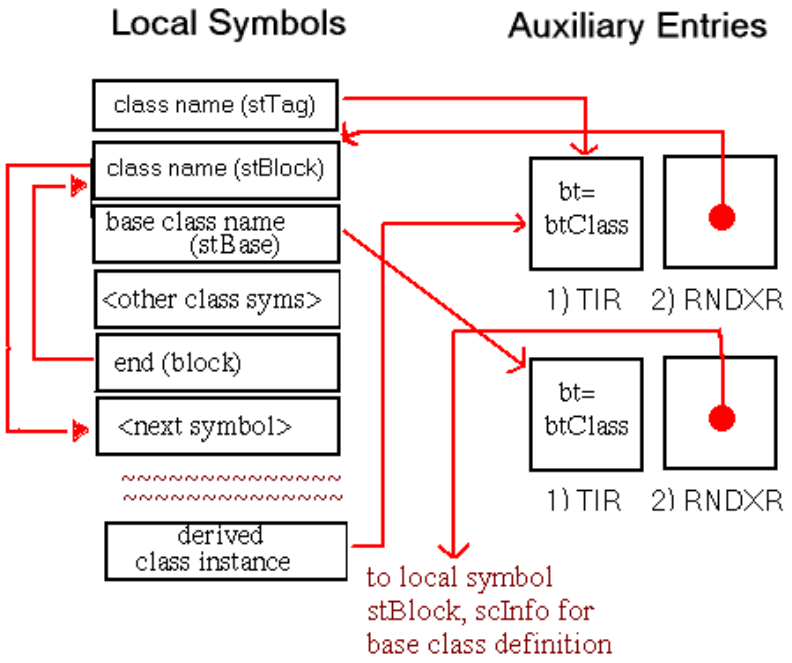
Version Note

Prior to Tru64 UNIX **V5.1** the default compilers used the preceding representation for empty classes and structures as well as opaque classes and structures.

5.3.8.6.3 Base and Derived Classes (C++)

Hierarchical groups of classes can be designed in C++. A base class serves as a wider classification for its derived classes, and a derived class has all of the members and methods of the base class, plus additional members of its own. In the symbol table, the set of symbols denoting a derived class is nearly identical to that for a non-derived class. The derived class includes an additional `stBase` or `stVirtBase` symbol that identifies its corresponding base class, and it does not need to duplicate the definitions for the base class members. This representation is shown in Figure 5–41.

Figure 5–41: Base Class Representation



The representation of virtual base classes for C++ relies on the definition of a special symbol that identifies the virtual base table. The name for this symbol is derived from the name of the class to which it belongs. For example, the virtual base table symbol for class `C5` would be named `"_btbl_2C5"`. This table contains entries for base class run-time descriptions.

A class can include the special member `_bptr`. This class member is a pointer to the virtual base table for that class.

The `value` field for a virtual base class symbol (`stVirtBase`/`scInfo`) serves as an index (starting at 1) into the virtual base class table.

5.3.8.7 Template Type (C++)

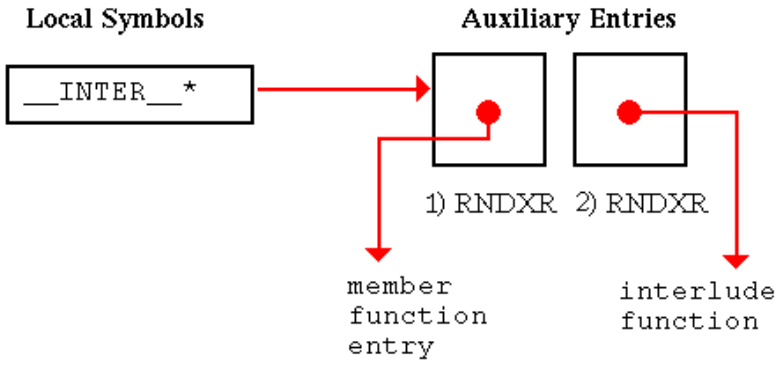
Templates are a C++-specific language construct allowing the parameterization of types. C++ class templates are represented in the symbol table for each

instantiation, but not for the template itself. The set of class symbols is unchanged from the set shown in Figure 5–38.

5.3.8.8 Interlude Type (C++)

Interludes are compiler generated functions in C++. They are represented in the local symbol table with special names starting with the "__INTER__" prefix. Their representation in the symbol table makes use of two RNDXR aux entries to identify the related member function and the actual interlude function, both of which are local symbol table entries.

Figure 5–42: Interlude Representation



5.3.8.9 Array Descriptor Type (Fortran90)

A Fortran90 array descriptor is a structure that describes an array: its location, dimensions, bounds, sizes, and other attributes. Array descriptors are described in detail in the Fortran 90 User Manual for Tru64 UNIX. Fortran90 includes several types of arrays for which the dimensions or dimension bounds are determined at run time: allocatable arrays, assumed shape arrays, and array pointers.

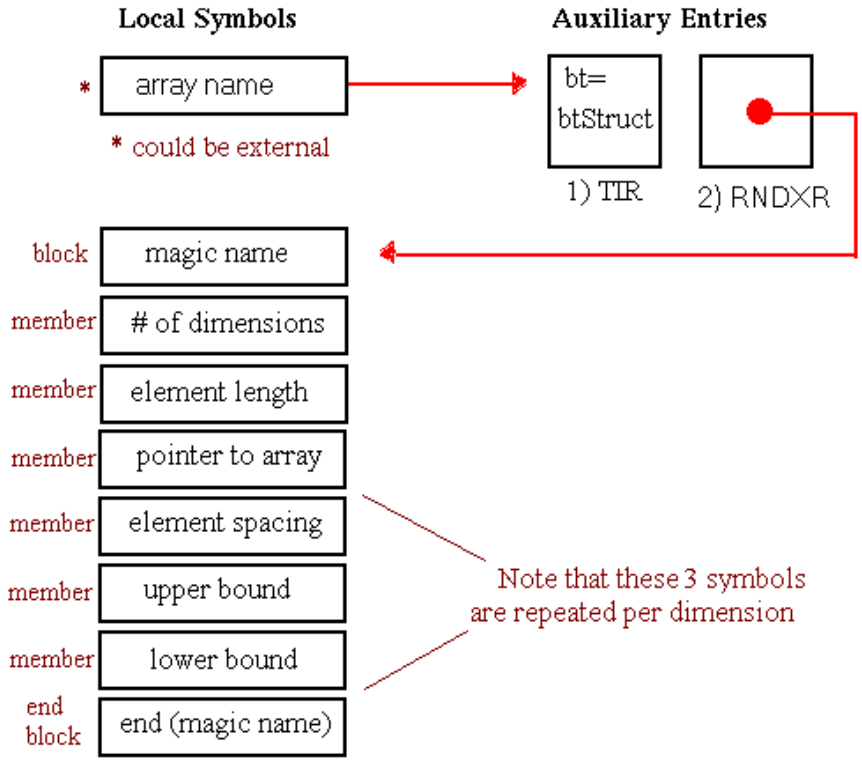
Two symbol table representations have been used for array descriptors. The current representation describes the array descriptor itself. The retired representation described attributes of the array known at compile time.

For both representations, symbols of this type point to a data location at which the array descriptor is allocated. One of the array descriptor fields contains a pointer to the actual array. Other fields are used to describe the attributes of the array. Fields that describe the number of dimensions and upper and lower bounds are filled in at run time.

By default, array descriptors are described by a structure tag representation. Most of the array descriptor fields are represented as structure members. (Excluded fields are not needed by debuggers.) Special tag names are used to identify array descriptor structure definitions: `$f90$f90_array_desc` (assumed-shape array), `$f90$f90_ptr_desc` (pointer to array) and `$f90$f90_alloc_desc` (allocatable array). Figure 5–43 shows the format of this representation.

Some compilers may emit other fields in addition to those shown in Figure 5–43. A consumer's ability to interpret additional fields depends on its knowledge of the producing compiler.

Figure 5–43: Array Descriptor Representation



An example of the default Fortran array descriptor representation can be found in Section 9.3.3.

Version Note

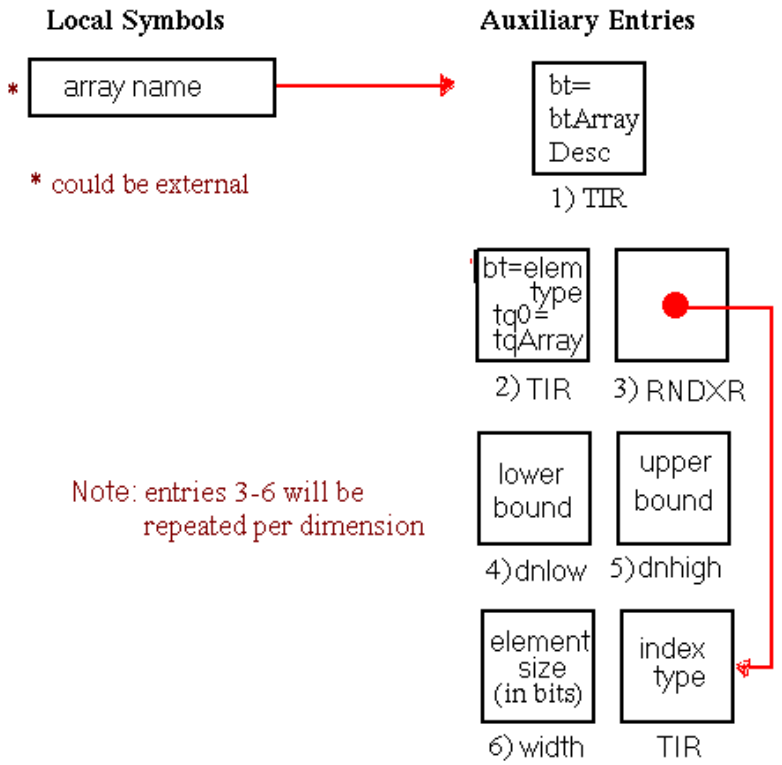
The following representation of Fortan array descriptors is supported in symbol table formats less than **V3.13**. It is not supported in symbol table format **V3.13** and greater.

This retired representation of Fortran array descriptors is substantially more compact in the local symbol table, but it provides no way to distinguish between the different array descriptor types.

The overloaded basic type value 28 indicates an array descriptor in the `TIR`, and dimension bounds are set to [1:1] indicating their true size is unknown. The alternate representation does not provide any information describing the contents of the array descriptor itself, so debuggers must assume a static representation for the descriptor and lookup the fields at their expected offsets.

Figure 5–44 shows this representation of array descriptors.

Figure 5–44: Array Descriptor Representation (retired)



5.3.8.10 Conformant Array Type (Pascal)

Full details are not currently available for Pascal's conformant array representation. A Pascal conformant array is very similar to Fortran's assumed shape arrays. It is an array parameter with upper and lower dimension bounds that are determined by the input argument. A conformant array is represented by an array descriptor. The special names used and the format of the array descriptor differ from those used for Fortran. The DEC Pascal release notes contain additional information on conformant arrays.

5.3.8.11 Variant Record Type (Pascal and Ada)

A variant record is an extension to the record data type, which is a Pascal or Ada data structure akin to a C structure and is represented in the same manner in the symbol table. The variant part of the record consists of sets of one or more fields associated with a range of values. Only one such set is part of the record, and it is selected based on the value of another record field. Any number of variant parts can be embedded in a single record.

Version Note

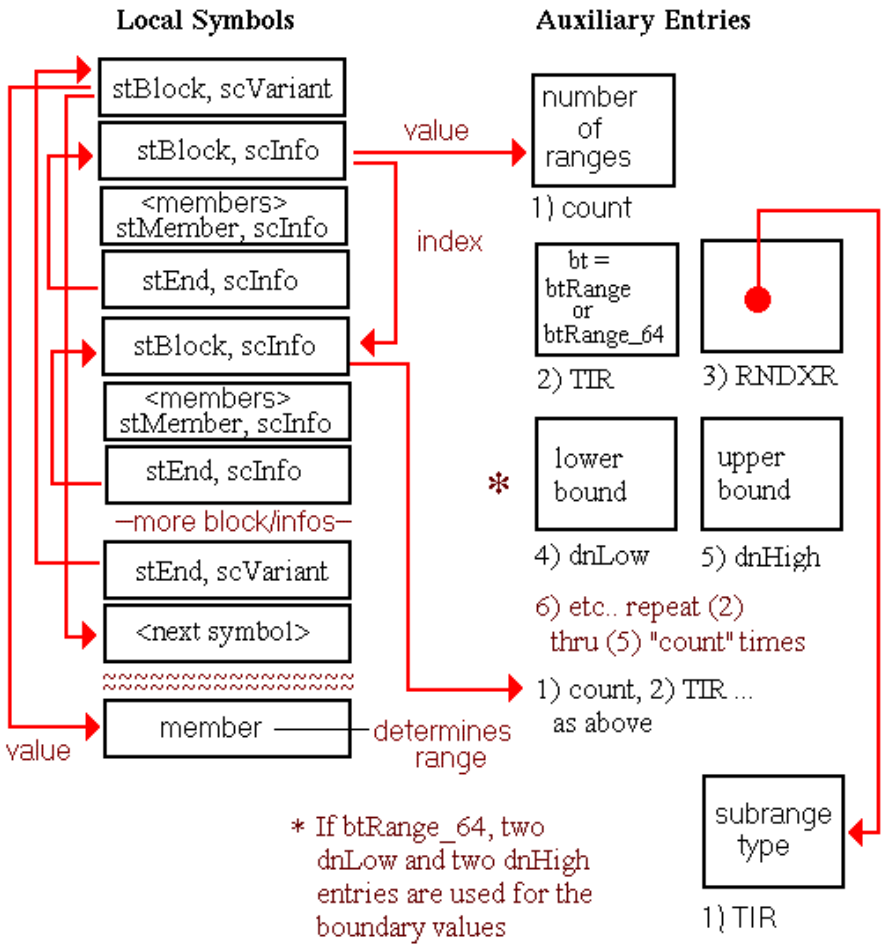
The following variant record representation is for symbol table format **V3.13** and greater.

The local symbol table entries for the variant part of a record are contained within a block with the storage class (`sc` value) `scVariant`. The `value` field of the `stBlock` entry contains the index of the local symbol entry for the member of the record whose value determines which variant arm is used. The variant block

contains multiple inner blocks, each representing a variant arm. The `value` field of each of these block entries is an auxiliary table index. Each auxiliary table entry starts with a `count`, which indicates how many range entries follow. The range entries describe the values associated with the block.

Figure 5–45 is a graphical representation of a variant record.

Figure 5–45: Variant Record Representation

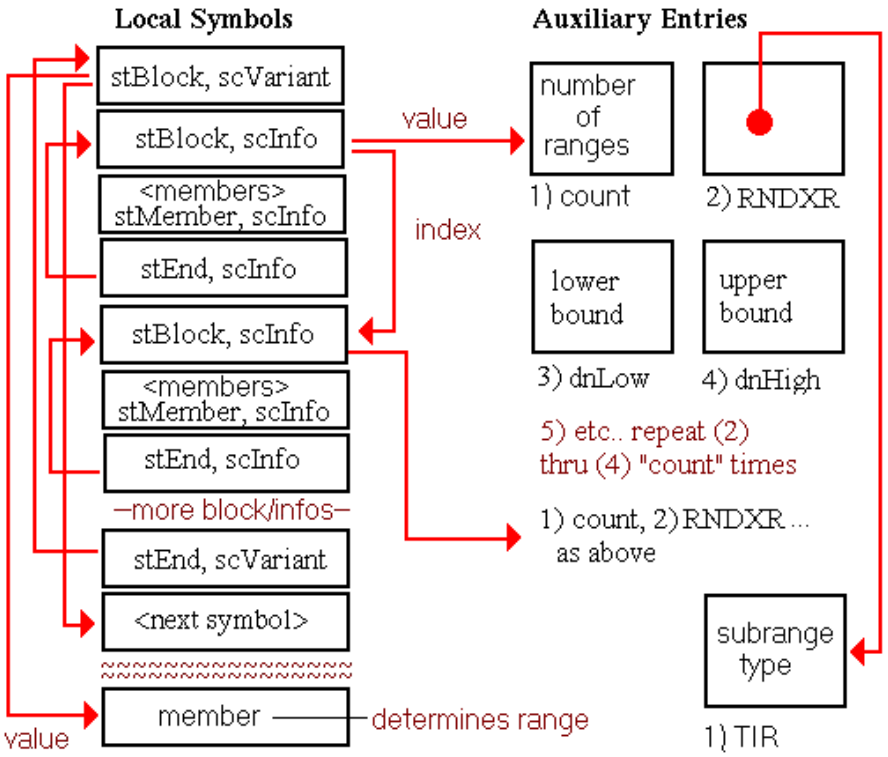


Version Note

The following variant record representation is for symbol table formats less than **V3.13**. It is not supported in symbol table format **V3.13** and greater.

The representation of variant records depicted in Figure 5–46 does not include `TIR` auxiliaries.

Figure 5–46: Variant Record Representation (retired)



An example of a Pascal variant record can be found in Section 9.4.3.

5.3.8.12 Subrange Type (Pascal and Ada)

A subrange data type defines a subset of the values associated with a particular ordinal type (the "base type" of the subrange). Ordinal types in Pascal include integers, characters, and enumerated types. The symbol table representation of a subrange uses the `btRange` or `btRange_64` type followed by an auxiliary index identifying the base type and entries providing the bounds of the subrange. The 32-bit representation is shown in Figure 5–47 and the 64-bit representation is shown in Figure 5–48.

Figure 5–47: Subrange Representation

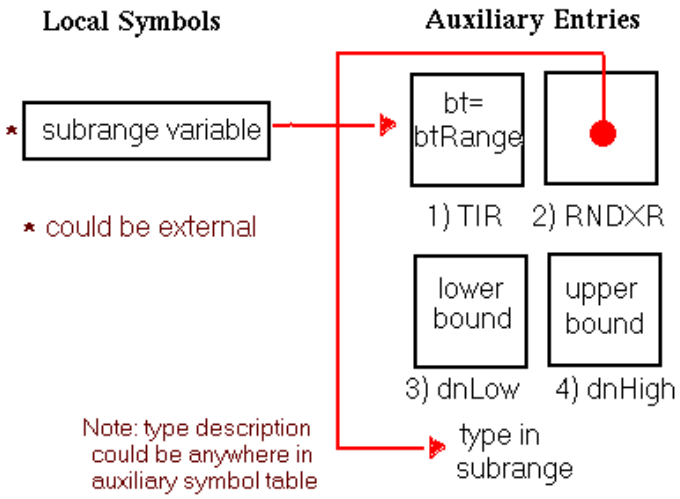
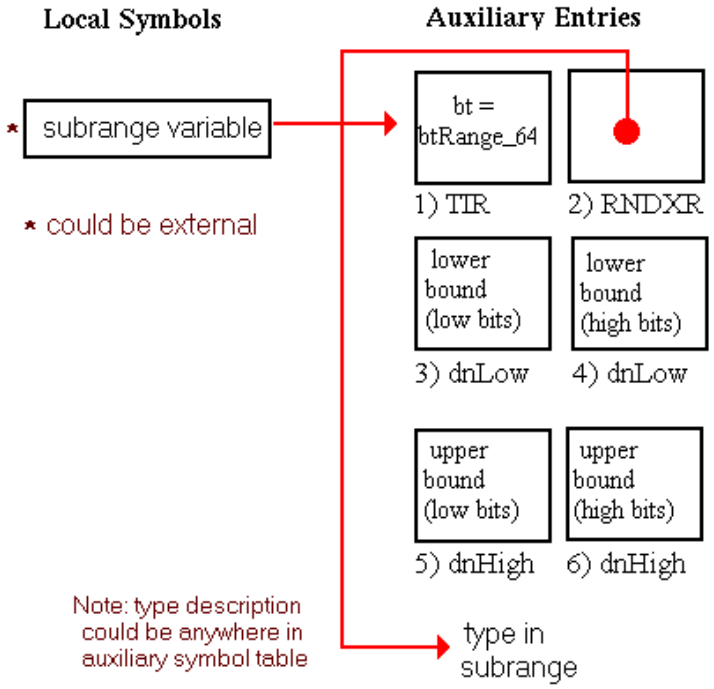


Figure 5–48: 64-bit Range Representation



Version Note

The 64-bit range representation is supported in symbol table format **V3.13** and greater.

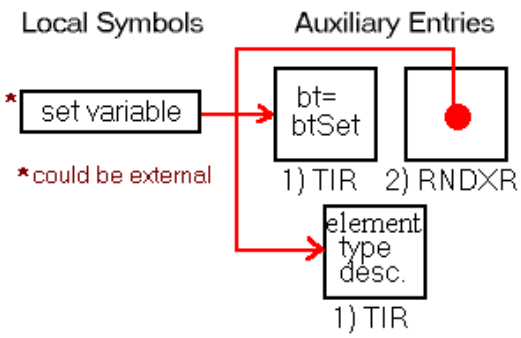
An example of a Pascal subrange can be found in Section 9.4.2.

5.3.8.13 Set Type (Pascal)

A set is a data type that groups ordinal elements in an unordered list. The arithmetic and logical operators are overloaded in Pascal; this enables them to be used with set variables to perform classic set operations such as union and

intersection. A special auxiliary type definition `btSet` exists to identify this type. The symbol table representation is depicted in Figure 5–49.

Figure 5–49: Set Representation



The element type for a set is typically a range or an enumeration. An example of a Pascal set can be found in Section 9.4.1.

5.3.9 Special Debug Symbols

A variety of special symbols are used throughout the symbol table to convey call frame information, special type semantics, or other language specific information. These names are reserved for use by compilers and other tools that produce Tru64 UNIX object files.

Table 5–14: Special Debug Symbols

Name	Purpose
Name	Purpose
`__StaticLink.*`	**(SV3.13 -)** Uplevel link. See Section 5.3.4.4.
`_BLNK__`	Fortran unnamed common block. See Section 5.3.6.6.
`MAIN__`	Fortran alias for main program unit. See Section 5.3.10.4.
ARGNAME`.len`	Generated parameter for Fortran routines. It contains the length of *ARGNAME*, a parameter of character type.
`.lb_<ARRAY>.<dim>` `.ub_<ARRAY>.<dim>`	Lower and upper bounds of particular dimensions of arrays - when the array has an explicit shape, yet some bounds come from non-constant specification expressions (array arguments in Pascal and Fortran routines).

`$f90$f90_array_desc`
`$f90$f90_alloc_desc`
`$f90$f90_ptr_desc`
	Variants of Fortran-90 described arrays (assumed shape, ALLOCATABLE, and POINTER, respectively). See Section 5.3.8.9.

`cray pointee`	Fortran-generated typedef describing the type of a variable pointed to by a CRAY pointer.
`pointer`	Fortran generated typedef describing the type of a scalar with the POINTER attribute.
`_DECCXX_generated_name_*`	DECC++ compiler-inserted name for unnamed classes and enumerations.
`this`	Hidden parameter in C++ member functions that is a pointer to the current instance of the class. See Section 5.3.8.6.
`__vptr`	Hidden C++ class member containing the virtual function table. See example in Section 9.2.2.

Table 5–14: Special Debug Symbols (cont.)

Name	Purpose
`__bptr`	Hidden C++ class member containing the virtual base class table. See example in Section 9.2.2.
`__vtbl_*`	Global symbols for C++ virtual function tables. See example in Section 9.2.2.
`__btbl_*`	Global symbols for C++ virtual base class tables. See example in Section 9.2.2.
`__control`	Hidden argument to C++ constructors controlling descent (in the face of virtual base classes).
`__t*__evdf`	Structure used to maintain a list of C++ global deconstructors.
`t*__iviw`	C++ static procedure used for global constructors.
`t*__evdw`	C++ static procedure used for global destructors.
`__t*_thunk`	C++ static procedure used to provide a defaulted argument value.
`__INTER__*`	C++ interlude. See example in Section 9.2.2.
`__N1*`	C++ unnamed namespaces. See example in Section 9.2.4.

5.3.10 Symbol Resolution

Among the linker's chief tasks is symbol resolution. Because most compilations involve multiple source files and virtually all programs rely on system libraries, a process is necessary to resolve conflicting uses of global symbol names. The linker must decide which symbol is referenced by a given name. This section highlights the major issues involved in that decision. Related information is contained in Section 6.3.4 and the *Programmer's Guide*.

Symbol table entries provide information relevant to performing symbol resolution. External symbols with a storage class of `sc(S)Undefined`, `sc(S)Common`, or `scTlsCommon` must be resolved before they are referenced. By default, the linker will not mark an object file with unresolved symbols as executable. However, linker options give programmers a fair measure of control over its symbol resolution behavior. See `ld`(1) for more information.

5.3.10.1 Library Search

Symbols referenced, but not defined in the main executable of an application must be matched with definitions in linked-in libraries. The linker combines objects, archives, and shared libraries while attempting to resolve all references to undefined symbols. The *Programmer's Guide* covers related topics in detail, such as how to specify libraries during compilation and the search order of libraries.

In general, main executable objects and shared libraries are searched before archive libraries. If no undefined external symbols remain, archive libraries in the library list do not have to be searched, because archive members are only loaded to resolve external references. Archives are not used to find "better" common definitions (see Section 5.3.10.2), and no archive definitions preempt symbol definitions from the main object or shared libraries.

5.3.10.2 Resolution of Symbols with Common Storage Class

Symbols with common storage class are a special category of global symbols that have a size but no allocated storage. Symbols with common storage class should

not be confused with Fortran common symbols, which are not represented by a single symbol table entry. (See Section 5.3.6.6 for a description of Fortran common symbols.) Common storage classes are `scCommon`, `scSCommon`, and `scTlsCommon`.

The symbol definition model used by Tru64 UNIX allows an unlimited number of common storage class symbols with the same name. Ultimately, the "best" of these must be selected (by the linker or the loader) during symbol resolution. The criteria used to select the best symbol definition include the symbol's allocation status and size.

The symbol table does not provide an "allocated common" storage class. Common storage class symbols adopt a new storage class when they are allocated. Typically, their new storage class is `scBss` or `scSBss` or `scTlsBss`. On the other hand, the dynamic symbol table does explicitly distinguish common storage class symbols that have been allocated. See Section 6.3.4 for more information on dynamic symbol resolution.

A symbol reference is resolved according to the following precedence rules:

1. Find a symbol definition that does not have a common storage class and is not identified as an allocated common in the dynamic symbol table.
2. Find the largest allocated common identified in the dynamic symbol table.
3. Find the largest common storage class symbol and allocate it. This step will be skipped when the linker produces a relocatable object file.

Precedence is given to symbol definitions with storage allocation to minimize load time common allocation and redundant storage allocations in shared objects. The loader is capable of allocating space for common storage class symbols, but this should only be necessary when a program references an allocated common symbol in a shared library that is later removed from that shared library.

Note that Fortran common block representations use common storage class symbols. Another very frequent occurrence of a common storage class symbol is a C-language global variable that does not have an initializer in its declaration.

5.3.10.3 Mangling and Demangling

Another issue related to symbol resolution is the need to "mangle" user-level identifiers. For example, C++ allows function overloading, prototyping, and the use of templates–all of which can result in the occurrence of the same names for different entities. The solution employed by the symbol table is to use mangled names that derive from the symbol's type signature.

Object file consumers, such as debuggers and object dumpers, need to "demangle" the identifiers so they can be output in a form that is recognizable to the user. For linking and loading, the mangled names are used for symbol resolution.

The encoding of C++ names is described in the manual *Using DEC C++ for Tru64 UNIX Systems*.

Other compilers may write symbol names that are modified by prepending or appending special characters such as dollar sign ($) or underscore (_) or by prepending qualifier strings such as file names or namespace names. Uppercasing of names is also common for certain languages such as Fortran. All of these transformations fall into the general category of mangled names. Refer to the release notes for specific compilers for additional information.

5.3.10.4 Mixed Language Resolution

Compilation of a program involving multiple source languages introduces additional symbol resolution issues. One important task is resolving the main

program entry point because conflicting "main" symbols may be present in the different files. For C and C++, the symbol "main" is the main program entry point, but for other languages, "main" will either be an alias for the main program or an interlude. DEC Fortran and DEC COBOL provide interludes that perform some language specific initializations and then call the real main program entry point. For DEC Fortran the main program is "MAIN__" and for DEC COBOL the main program is "__cobol_main". DEC Pascal provides a "main" symbol that aliases the actual main program symbol.

The symbols "MAIN__" and "__cobol_main" can both be present in a mixed language program, and either, neither, or both can be used by the program. Debuggers can set a breakpoint in the user's main program by applying some precedence for selecting the most appropriate symbol. For a mixed language program, there is a slight chance that "MAIN__" or "__cobol_main" will be present but never called.

5.3.10.5 TLS Symbols

TLS (Thread Local Storage) symbols, like non-TLS symbols, can be undefined or common. Unresolved TLS symbols are identified by the storage class `scTlsUndefined`, and TLS commons have the storage class `scTlsCommon`. The symbol resolution process for TLS names is similar, but separate; TLS symbols cannot be resolved to non-TLS symbols or vice versa.

TLS common symbols are resolved in the same manner as other common storage class symbols (see Section 5.3.10.2), except that, again, only TLS symbols are candidates for resolution.

Another rule special to TLS is that symbol definitions for TLS common and undefined symbols cannot be imported from shared libraries.

5.4 Language-Specific Symbol Table Features

Language-specific characteristics are pervasive in the symbol table, particularly in the local, external, and auxiliary symbol tables. See Section 5.2 and Section 5.3.7 for information on language-specific values.

The `lang` field of the file descriptor entry encodes the source language of the file. This field should be accessed prior to decoding symbolic information, especially type descriptions. This section highlights, by language, language-specific features represented in the symbol table. Additional information on certain features is available elsewhere in this chapter.

5.4.1 Fortran77 and Fortran90

In Fortran, it is possible to create multiple entry points in subroutines. A subroutine has one main entry point and zero or more alternate entry points, indicated by `ENTRY` statements. See Section 5.3.6.7 for their representation in the symbol table.

Fortran90 array descriptors include allocatable arrays, assumed-shape arrays, and pointers to arrays. Their representation in the symbol table is discussed in Section 5.3.8.9.

Modules provide another scoping level in Fortran90 programs. The symbol table representation for modules has not yet been implemented.

5.4.2 C++

C++ classes encapsulate functions and data inside a single structure. Classes are represented in the symbol table using a `btClass` basic type and the `stBlock`/`stEnd` scoping mechanism. See Section 5.3.8.6.

Templates provide for parameterized types. At present, no special symbol table values are related to templates. The template itself is not represented; rather, entries that correspond to each instantiation are generated. Template instantiations are distinguished by mangled names based on their type signatures.

C++ namespaces, like Fortran modules, offer an additional scope for program identifiers.

The C++ concepts of private, protected, and public data attributes are not currently represented in the symbol table. The C++ concept of "friend" classes and functions are also not represented.

5.4.3 Pascal and Ada

Pascal conformant arrays are function parameters with array dimensions that are determined by the arguments passed to the function at run time. See Section 5.3.8.10.

Variant records are an extension of the record data structure. Variant records allow different sets of fields depending on the value of a particular record member. See Section 5.3.8.11.

Nested procedures are supported in these languages. They are represented using standard scoping mechanisms discussed in Section 5.3.6 and uplevel references described in Section 5.3.4.4.

Sets and subranges are user-defined subsets of ordinal types. Sets are unordered groups of elements, which can be manipulated with the classic set operations. Subranges are ordered and are used with the usual operators. See Section 5.3.8.12 and Section 5.3.8.13.

Ada subtypes of ordinal types are represented in the same manner as Pascal subranges.

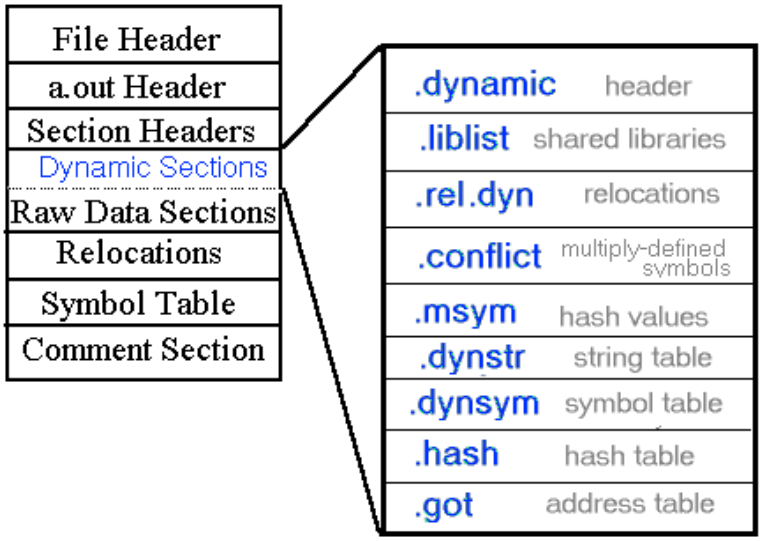
6

Dynamic Loading Information

The dynamic linker/loader (commonly referred to as the loader) is responsible for creating a dynamic executable's process image and placing it into system memory so that it can execute. The loader's functions include finding and mapping shared libraries, completing symbol resolution, and finalizing program addresses.

To accomplish these functions, the loader requires information on external symbols and shared libraries. The linker prepares this dynamic loading information for shared objects only. The dynamic loader then uses this information to create and map the process image. The dynamic information consists of the sections highlighted in Figure 6–1.

Figure 6–1: Dynamic Object File Sections



These sections are mapped with the text segment, except for the `.got` section, which contains the GOT (Global Offset Table). The GOT is part of the data segment because it must be written into when addresses are updated.

The function of each dynamic section can be summarized as follows:

- The `.dynamic` section serves as a header for the dynamic information.
- The `.dynsym` section contains the dynamic symbol table.
- The `.dynstr` section contains the names of dynamic symbols and shared library dependencies.
- The `.hash` section holds a hash table to provide quick access into the dynamic symbol table.
- The `.msym` table contains supplemental symbolic information, including pre-computed hash values and dynamic relocation indices.
- The `.liblist` section stores dependency information.

- The `.conflict` section contains a list of multiply-defined symbol names that must be resolved at load time.
- The `.rel.dyn` section contains dynamic relocation entries.
- The `.got` section contains one or more tables of 64-bit run-time addresses.

This chapter covers the dynamic sections and related topics. The actions of the system dynamic loader are explained in detail. Related material is available in the *Programmer's Guide* and `loader`(5).

6.1 New or Changed Dynamic Loading Information Features

Tru64 UNIX V5.0 supports depth-first symbol resolution order for individual shared objects. See `DT_SYMBOLIC` in Section 6.2.1 for details.

6.2 Structures, Fields, and Values for Dynamic Loading Information

All structures and macros are declared in the header file `coff_dyn.h` unless otherwise indicated.

6.2.1 Dynamic Header Entry

```
typedef struct {
        coff_int      d_tag;
        coff_uint     reserved;
        union {
            coff_uint d_val;
            coff_addr d_ptr;
        } d_un;
} Coff_Dyn;
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

Dynamic Header Entry Fields

`d_tag` — Indicates how the `d_un` field is to be interpreted.

`reserved` — Must be zero.

`d_val` — Represents integer values.

`d_ptr` — Represents virtual addresses. Virtual addresses stored in this field may not match the memory virtual addresses during execution. The dynamic loader computes actual addresses based on the virtual address from the file and the memory base address. Object files do not contain relocation entries to correct addresses in the dynamic section.

The `d_tag` requirements for dynamic executable files and shared library files are summarized in Table 6–1. "mandatory" indicates that the dynamic linking array must contain an entry of that type; "optional" indicates that an entry for the tag may exist but is not required.

Table 6–1: Dynamic Array Tags (`d_tag`)

Name	Value	`d_un`	Executable	Shared Library
`DT_NULL`	0	ignored	mandatory	mandatory
`DT_NEEDED`	1	d_val	optional	optional
`DT_PLTGOT`	3	d_ptr	optional	optional

Table 6–1: Dynamic Array Tags (`d_tag`) (cont.)

Name	Value	d_un	Executable	Shared Library
DT_HASH	4	d_ptr	mandatory	mandatory
DT_STRTAB	5	d_ptr	mandatory	mandatory
DT_SYMTAB	6	d_ptr	mandatory	mandatory
DT_STRSZ	10	d_val	optional	optional
DT_SYMENT	11	d_val	optional	optional
DT_INIT	12	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_SONAME	14	d_val	ignored	mandatory
DT_RPATH	15	d_val	optional	ignored
DT_SYMBOLIC	16	ignored	optional	optional
DT_REL	17	d_ptr	mandatory	mandatory
DT_RELSZ	18	d_val	mandatory	mandatory
DT_RELENT	19	d_val	optional	optional
DT_RLD_VERSION	0x70000001	d_val	mandatory	mandatory
DT_TIME_STAMP	0x70000002	d_val	optional	optional
DT_ICHECKSUM	0x70000003	d_val	optional	optional
DT_IVERSION	0x70000004	d_val	optional	optional
DT_FLAGS	0x70000005	d_val	optional	optional
DT_BASE_ADDRESS	0x70000006	d_ptr	optional	optional
DT_MSYM	0x70000007	d_ptr	optional	optional
DT_CONFLICT	0x70000008	d_ptr	optional	optional
DT_LIBLIST	0x70000009	d_ptr	optional	optional
DT_LOCAL_GOTNO	0x7000000A	d_val	mandatory	mandatory
DT_CONFLICTNO	0x7000000B	d_val	optional	optional
DT_LIBLISTNO	0x70000010	d_val	optional	optional
DT_SYMTABNO	0x70000011	d_val	mandatory	mandatory
DT_UNREFEXTNO	0x70000012	d_val	optional	optional
DT_GOTSYM	0x70000013	d_val	mandatory	mandatory
DT_HIPAGENO	0x70000014	d_val	optional	optional
DT_SO_SUFFIX	0x70000017	d_val	optional	optional

The uses of the various dynamic array tags are as follows:

DT_NULL — Marks the end of the array.

DT_NEEDED — Contains the string table offset of a null-terminated string that is the name of a needed library. The offset is an index into the table indicated in the DT_STRTAB entry. The dynamic array can contain multiple entries of this type. The order of these entries is significant.

DT_HASH — Contains the quickstart address of the symbol hash table.

DT_STRTAB	Contains the quickstart address of the string table.
DT_SYMTAB	Contains the quickstart address of the symbol table with Coff_Sym entries.
DT_STRSZ	Contains the size of the string table (in bytes).
DT_SYMENT	Contains the size of a symbol table entry (in bytes).
DT_INIT	Contains the quickstart address of the initialization function.
DT_FINI	Contains the quickstart address of the termination function.
DT_SONAME	Contains the string table offset of a null-terminated string that gives the name of the shared library file. The offset is an index into the table indicated in the DT_STRTAB entry.
DT_RPATH	Contains the string table offset of a null-terminated library search path string. The offset is an index into the table indicated in the DT_STRTAB entry.
DT_SYMBOLIC	The presence of this entry indicates that symbol references should be resolved using a depth-ring search of the shared object's dependencies. See Section 6.3.4.3 for a details on shared object search order. This dynamic entry is for information only. The search order is controlled by the DT_FLAGS setting that includes the RHF_RING_SEARCH and RHF_DEPTH_FIRST flags when DT_SYMBOLIC is added to the dynamic section. **Version Note** DT_SYMBOLIC is supported in Tru64 UNIX **V5.0** and greater.
DT_REL	Contains the address of the dynamic relocation table. If this entry is present, the dynamic structure must contain the DT_RELSZ entry.
DT_RELSZ	Contains the size (in bytes) of the dynamic relocation table pointed to by the DT_REL entry.
DT_RELENT	Contains the size (in bytes) of a DT_REL entry.
DT_RLD_VERSION	Contains the version number of the run-time linker interface. The version is: • 1 for executable objects that have a single GOT • 2 for executable objects that have multiple GOTs • 3 only for objects built on Tru64 UNIX **V2.x**

DT_TIME_STAMP	Contains a 32-bit time stamp.
DT_ICHECKSUM	Contains a checksum value computed from the names and other attributes of all symbols exported by the library.
DT_IVERSION	Contains the string table offset of a series of colon-separated versions. An index value of zero means no version string was specified.
DT_FLAGS	Contains a set of 1-bit flags. See Table 6–2 for a list of supported flag values.
DT_BASE_ADDRESS	Contains the quickstart base address of the object.
DT_CONFLICT	Contains the quickstart address of the .conflict section.
DT_LIBLIST	Contains the quickstart address of the .liblist section.
DT_LOCAL_GOTNO	Contains the number of local GOT entries. The dynamic array contains one of these entries for each GOT.
DT_CONFLICTNO	Contains the number of entries in the .conflict section.
DT_LIBLISTNO	Contains the number of entries in the .liblist section.
DT_SYMTABNO	Indicates the number of entries in the .dynsym section.
DT_UNREFEXTNO	Holds the index to the first dynamic symbol table entry that is an external symbol not referenced within the object.
DT_GOTSYM	Holds the index to the first dynamic symbol table entry that corresponds to an entry in the global offset table. The dynamic array contains one of these entries for each GOT.
DT_HIPAGENO	Not used by the default system loader. If present, must contain the value 0.
DT_SO_SUFFIX	Contains a shared library suffix that the loader appends to library names when searching for dependencies. This tag is used, for example, with Atom tools. Instrumented applications may be dependent on instrumented shared libraries identified by a tool-specific suffix.

All other tag values are reserved. Entries can appear in any order, except for the DT_NULL entry at the end of the array and the relative order of the DT_NEEDED entries.

Table 6–2: DT_FLAGS Flags

Flag	Value	Meaning
RHF_QUICKSTART	0x00000001	Object may be quickstarted by loader
RHF_NOTPOT	0x00000002	Hash size not a power of two

Table 6–2: DT_FLAGS Flags (cont.)

Flag	Value	Meaning
RHF_NO_LIBRARY_REPLACEMENT	0x00000004	Use default system libraries only
RHF_NO_MOVE	0x00000008	Do not relocate
RHF_TLS	0x04000000	Identifies objects that use TLS
RHF_RING_SEARCH	0x10000000	Symbol resolution same as DT_SYMBOLIC. This flag is only meaningful when combined with RHF_DEPTH_FIRST
RHF_DEPTH_FIRST	0x20000000	Depth-first symbol resolution
RHF_USE_31BIT_ADDRESSES	0x40000000	TASO (Truncated Address Support Option) objects

6.2.2 Dynamic Symbol Entry

```
typedef struct {
        coff_uint       st_name;
        coff_uint       reserved;
        coff_addr       st_value;
        coff_uint       st_size;
        coff_ubyte      st_info;
        coff_ubyte      st_other;
        coff_ushort     st_shndx;
} Coff_Sym;
```

SIZE - 24 bytes, ALIGNMENT - 8 bytes

See Section 6.3.3 for related information.

Dynamic Symbol Entry Fields

st_name — Contains the offset of the symbol's name in the dynamic string section.

reserved — Must be zero.

st_value — Contains the quickstart address if the symbol is defined within the object. Contains 0 for undefined external symbols, the alignment value for commons, or any arbitrary value for absolute symbols.

For undefined external conflict symbols (see Section 6.3.6.2) this field will contain the quickstart address of the symbol in the first shared library in which the linker found a definition of the symbol.

st_size — Identifies the size of symbols with common storage allocation; otherwise, contains the value zero. For STB_DUPLICATE symbols (see Table 6–4). The size field holds the index of the primary symbol.

st_info — Identifies the symbol's binding and type. The macros COFF_ST_BIND and COFF_ST_TYPE are used to access the individual values. See Table 6–3 and Table 6–4 for the possible values.

st_other — Currently has a value of zero and no defined meaning.

st_shndx — Identifies the symbol's dynamic storage class. See Table 6–5 for the possible values.

Table 6–3: Dynamic Symbol Type (st_info) Constants

Name	Value	Description
STT_NOTYPE	0	Indicates that the symbol has no type or its type is unknown.
STT_OBJECT	1	Indicates that the symbol is a data object.
STT_FUNC	2	Indicates that the symbol is a function.
STT_SECTION	3	Indicates that the symbol is associated with a program section.
STT_FILE	4	Indicates that the symbol is the name of a source file.

Table 6–4: Dynamic Symbol Binding (st_info) Constants

Name	Value	Description
STB_LOCAL	0	Indicates that the symbol is local to the object (or designated as hidden).
STB_GLOBAL	1	Indicates that the symbol is visible to other objects.
STB_WEAK	2	Indicates that the symbol is a weak global symbol.
STB_DUPLICATE	13	Indicates the symbol is a duplicate. (Used for objects that have multiple GOTs.)

Table 6–5: Dynamic Section Index (st_shndx) Constants

Name	Value	Description
SHN_UNDEF	0x0000	Indicates that the symbol is undefined.
SHN_ACOMMON	0xff00	Indicates that the symbol has common storage (allocated).
SHN_TEXT	0xff01	Indicates that the symbol is in a text segment.
SHN_DATA	0xff02	Indicates that the symbol is in a data segment.
SHN_ABS	0xfff1	Indicates that the symbol has an absolute value.
SHN_COMMON	0xfff2	Indicates that the symbol has common storage (unallocated).

6.2.3 Dynamic Relocation Entry

```
typedef struct {
        coff_addr       r_offset;
        coff_uint       r_info;
        coff_uint       reserved;
} Coff_Rel;
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

See Section 6.3.5 for related information.

Dynamic Relocation Entry Fields

r_offset — Indicates the quickstart address within the object that contains the value requiring relocation.

r_info — Indicates the relocation type and the index of the dynamic symbol that is referenced. The macros COFF_R_SYM and COFF_R_TYPE access the individual attributes. The relocation type must be R_REFQUAD, R_REFLONG, or R_NULL.

reserved	Must be zero.

6.2.4 Msym Table Entry

```
typedef struct {
        coff_uint ms_hash_value;
        coff_uint ms_info;
} Coff_Msym;
```

SIZE - 8 bytes, ALIGNMENT - 4 bytes

See Section 6.3.3.4 for related information.

Msym Table Entry Fields

ms_hash_value	Contains the hash value computed from the name of the corresponding dynamic symbol.
ms_info	Contains both the dynamic relocation index and the symbol flags field. The macros COFF_MS_REL_INDEX and COFF_MS_FLAGS are used to access the individual values. The dynamic relocation index identifies the first entry in the .rel.dyn section that references the dynamic symbol corresponding to this msym entry. If the index is 0, no dynamic relocations are associated with the symbol. The symbol flags field is reserved for future use and should be zero.

6.2.5 Library List Entry

```
typedef struct {
        coff_uint l_name;
        coff_uint l_time_stamp;
        coff_uint l_checksum;
        coff_uint l_version;
        coff_uint l_flags;
} Coff_Lib;
```

SIZE - 20 bytes, ALIGNMENT - 4 bytes

See Section 6.3.2 for related information.

Library List Entry Fields

l_name	Records the name of a shared library dependency. The value is a string table index. This name can be a full pathname, relative pathname, or file name.
l_time_stamp	Records the time stamp of a shared library dependency. The value can be combined with the l_checksum value and the l_version string to form a unique identifier for this shared library file.
l_checksum	Records the checksum of a shared library dependency.
l_version	Records the interface version of a shared library dependency. The value is a string table index.
l_flags	Specifies a set of 1-bit flags. The l_flags field can have one or more of the flags described in Table 6–6.

Table 6–6: Library List Flags

Name	Value	Description
LL_EXACT_MATCH	0x01	Requires that the run-time dynamic shared library file match exactly the shared library file used at static link time.
LL_IGNORE_INT_VER	0x02	Ignores any version incompatibility between the dynamic shared library file and the shared library file used at link time.
LL_USE_SO_SUFFIX	0x04	Marks shared library dependencies that should be loaded with a suffix appended to the name. The DT_SO_SUFFIX entry in the .dynamic section records the name of this suffix. This is used by object instrumentation tools to distinguish instrumented shared libraries.
LL_NO_LOAD	0x08	Marks entries for shared libraries that are not loaded as direct dependencies of an object. Object instrumentation tools may use LL_NO_LOAD entries to set the LL_USE_SO_SUFFIX for dynamically loaded shared libraries or for indirect shared library dependencies.

If neither LL_EXACT_MATCH nor LL_IGNORE_INT_VER bits are set, the dynamic loader requires that the version of the dynamic shared library match at least one of the colon-separated version strings indexed by the l_version string table index.

6.2.6 Conflict Entry

```
typedef struct {
        coff_uint   c_index;
} Coff_Conflict;
```

SIZE - 4 bytes, ALIGNMENT - 4 bytes

The conflict entry is an index into the dynamic symbols (.dynsym) section. See Section 6.3.6.2 for related information.

6.2.7 GOT Entry

```
typedef struct {
        coff_addr   g_index;
} Coff_Got;
```

SIZE - 8 bytes, ALIGNMENT - 8 bytes

The GOT entry is a 64-bit address. Most GOT entries map to dynamic symbols. See Section 6.3.3 for details.

6.2.8 Hash Table Entry

The hash table is implemented as an array of 32-bit values. The structure is declared internal to system utilities.

See Section 6.3.3.5 for more information.

6.2.9 Dynamic String Table

The dynamic string table consists of null-terminated character strings. The strings are of varying length and separated only by a single character. Offsets into the dynamic string table give the number of bytes from the beginning of the string space to the beginning of the name in question.

Offset 0 in the dynamic string table is reserved for the null string.

6.3 Dynamic Loading Information Usage

6.3.1 Shared Object Identification

A shared object is either a dynamic executable or a shared library. The file header flags indicate whether the object is a shared object and, if so, what type of shared object it is. The layout of the object is also stated in the file header. Normally shared objects use a `ZMAGIC` image layout (see Section 2.3.2.3).

Additional information on the shared object is located in the dynamic header (`.dynamic` section). When the dynamic loader is invoked by the kernel's `exec()` routine, this header information is read.

The kernel and loader take the following steps upon receiving a user command to execute a dynamic executable:

1. User enters command.
2. Shell calls `exec()` in kernel.
3. `exec()` opens the file and reads the file header.
4. If the file is a dynamic executable, `exec()` calls `/sbin/loader`.
5. The loader then:
    a. Reads file header and dynamic header information.
    b. Maps the executable into memory.
    c. Locates each shared library dependency, maps it into memory, and relocates it if necessary.
    d. Resolves symbols for all shared objects.
    e. Sets the heap address.
    f. Transfers control to program entry point.
6. The program entry point (`__start` in `crt0.o`) then:
    a. Calls special symbol `__istart` which invokes the loader routine to run INIT routines
    b. Calls `main()` with `__Argc`, `__Argv`, `__environ` and `_auxv`.

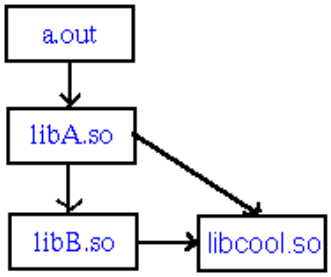
6.3.2 Shared Library Dependencies

Dynamic executables usually rely on shared libraries. At load time, these shared libraries must be located, validated, and mapped with the process image.

If an executable object refers to a symbol whose definition resides in a shared library, the executable is dependent on that library. This relationship is described as a direct dependency. A shared library dependency also exists if a library is used by any previously identified dependency. This is an indirect dependency for the executable.

In the example shown in Figure 6–2, `libA`, `libB`, and `libcool` are all shared library dependencies for `a.out`. The library `libA` is a direct dependency, and the others are indirect dependencies.

Figure 6–2: Shared Library Dependencies



Although the possibility of duplicate dependencies exists, as in the preceding example, each library is mapped only once with the image. The linker also prevents recursive inclusion, which could occur in a case of cyclic dependencies.

6.3.2.1 Identification

A shared object's dependencies are stored in its `.liblist` entries and in `DT_NEEDED` entries in the `.dynamic` section. The linker records this information as dependencies are encountered.

The library list (`.liblist` section) has name, timestamp, checksum, and version information for every entry, along with a flags field. Taken together, the timestamp and checksum value and the version string form a unique identifier for a shared library. An entry is created for each shared library dependency.

A `DT_NEEDED` tag in the dynamic header also indicates a shared library dependency. The value of the entry is the string table offset for the needed library's name. Note that this representation of the dependency information is redundant with that contained in the library list. The loader relies on the library list only. The `DT_NEEDED` entries are maintained for historical reasons.

As an example, an object linked against `libc` has the following dependency information:

```
     ***DYNAMIC SECTION***

     LIBLISTNO: 1.
     LIBLIST:   0x0000000120000690
     NEEDED:    libc.so


     ***LIBRARY LIST SECTION***

     Name             Time-Stamp        CheckSum   Flags Version
a.out:
     libc.so      May 19 22:18:46 1996 0xf937323b     0 osf.1
```

A shared library's checksum is computed by the linker when the library is created or updated, and the value is written into the dynamic header. When an application is linked against the library, the linker copies the library's current checksum into its entry in the application's `.liblist`.

The checksum computation is a summation of the names of dynamic symbols that meet the following criteria:

- Defined
- Not local
- Not hidden
- Not duplicate

Common storage class symbol names are included, along with their size. Weak symbols are included, but the calculation for weak symbols differs from that used for non-weak symbols.

For a single symbol, the checksum is computed using this algorithm :

```
if (SYMBOL.st_shndx == SHN_COMMON || SYMBOL.st_shndx == SHN_ACOMMON)
    CHECKSUM = SYMBOL.st_size
else
    CHECKSUM = 0

for (# of characters in symbol name)
    CHECKSUM = (CHECKSUM << 5) + character_value

if (weak symbol)
    CHECKSUM = (CHECKSUM << 5) + CHECKSUM + 1
```

A change in the number of weak symbols or a change in the size of a common storage class symbol is therefore reflected in the checksum. However, the checksum calculation is insensitive to symbol reordering.

The checksums for all symbols included are summed to produce the shared object's checksum.

6.3.2.2 Searching

After loading an executable, the loader loads the executable's shared library dependencies. The loader searches for shared libraries that match the names contained in the executable's `.liblist` entries. Subject to the search guidelines described in this section, the loader will load the first matching shared library that it finds for each dependency.

Certain directories are searched by default, in the following order:

1. `/usr/shlib`
2. `/usr/ccs/lib`
3. `/usr/lib/cmplrs/cc`
4. `/usr/lib`
5. `/usr/local/lib`
6. `/var/shlib`

The loader's search path can be altered by several methods:

- `-soname` linker option
- `-rpath` linker option
- environment variables

The `-soname` option is used to set internal shared library names. The default soname is the output file name of the library when it is built. The linker uses an soname value to record shared library dependencies in the library list. Dependencies containing pathnames are located without prepending search directories to their paths. A pathname is identified by the presence of one or more slashes in the string.

The RPATH is included in a shared object's `.dynamic` section under an entry tagged `DT_RPATH`. It is a colon-separated list of shared library search directories. The RPATH is set using the `-rpath` linker option. The loader will search RPATH directories prior to searching `LD_LIBRARY_PATH` and default directories.

The environment variables that impact the search order are `LD_LIBRARY_PATH` and `_RLD_ROOT`. `LD_LIBRARY_PATH` has the same format as RPATH.
No root directories are prepended to the `LD_LIBRARY_PATH` directories.
`LD_LIBRARY_PATH` can also be set by a program before it calls `dlopen()`.

The _RLD_ROOT environnment variable is a colon-separated list of "root" directories that are prepended to other search directories. It modifies RPATH and the default search directories.

The precedence (highest to lowest) of search directories used by the loader is as follows:

1. soname (if it includes a path)
2. _RLD_ROOT + RPATH
3. LD_LIBRARY_PATH
4. _RLD_ROOT + default search directories

When using non-system libraries, it is often necessary to specify the search path rather than relying on the defaults. Here is one example:

```
$ ld -shared -o my.so mylib.o -lc
$ cc -o hello hello.c my.so
$ hello
7526:hello: /sbin/loader: Fatal Error: cannot map my.so
$ LD_LIBRARY_PATH=.
$ export LD_LIBRARY_PATH
$ hello
Hello, World!
```

6.3.2.3 Validation

One of the loader's jobs is to ensure that correct shared libraries are available to the program. Shared library versioning is used to distinguish incompatible versions of shared libraries. The loader tests for matching versions when shared library dependencies are loaded. If the application is found to be incompatible with a needed shared library, the program may have to be recoded or relinked. Causes of binary incompatibility include altered global data definitions and changes to documented interfaces.

Each shared library is built with a version identifier. This identifier is recorded in the .dynamic section with the tag DT_IVERSION. Each entry in the dependency information (.liblist section) also records the version identifier of a shared library dependency. The -set_version linker option is used to provide the version identifier. Without this option, the linker will build a shared library with a null version. Version identifiers can be any ASCII string.

Version checking can also be controlled by the user. The linker option -exact_version leads to more rigorous version testing by the loader. When this option is in effect, timestamps and checksums are checked in addition to version numbers. The linker-recorded dependency information for the timestamp and checksum must precisely match the load-time values for all shared libraries. Normally, a mismatch leads to additional symbol resolution work instead of a rejected object.

Version checking can be disabled through use of the loader environment variable _RLD_ARGS. Setting this variable to -ignore_all_versions disables version testing for all shared library dependencies. Setting it to -ignore_version with a library name parameter turns off version checking for that specific dependency.

By default, versions are checked, but not checksums or timestamps. If version testing fails, the loader searches for the matching version of the shared library.

The version identifiers are used to locate version-specific libraries. The loader looks for these libraries in:

1. *dirname*/*version_id*
2. /usr/shlib/*version_id*

where *dirname* is the first directory where a library with a matching name but non-matching version is found.

For example, if an application needs version 1 of a shared library but the loader first encounters version 2, it continues looking for the correct version.

6.3.2.3.1 Backward Compatibility

When shared libraries are modified and new versions built, the older versions are frequently retained to support previously linked applications. Maintaining multiple versions of the library helps ensure backward compatibility for existing applications even after binary-incompatible changes have been made.

Backward-compatible shared libraries can be:

- Complete independent shared libraries
- Partial shared libraries that import missing symbols from other versions of the same shared libraries

The advantage of partial shared libraries is that they require less disk space; a disadvantage is that they require more swap space.

The linker's `-L` option can be used to link with backward-compatible shared libraries. Warnings are generated when a shared library is linked with dependencies on different versions of the same shared library. However, the linker tests direct dependencies only. The option `-transitive_link` should be used to uncover all multiple-version dependencies.

Multiple versions of the same shared library can only be loaded to support partial shared library dependencies. Otherwise, dependencies on multiple versions of a library are invalid.

Figure 6–3 shows examples of valid uses of multiple versions.

Figure 6–3: Valid Shared Library with Multiple Versions

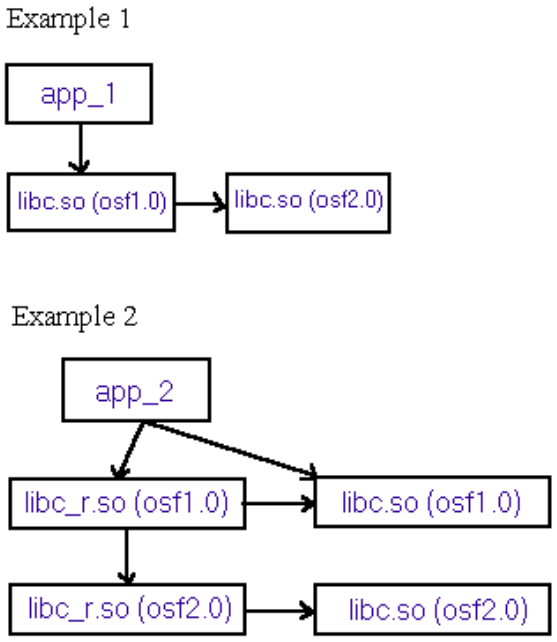
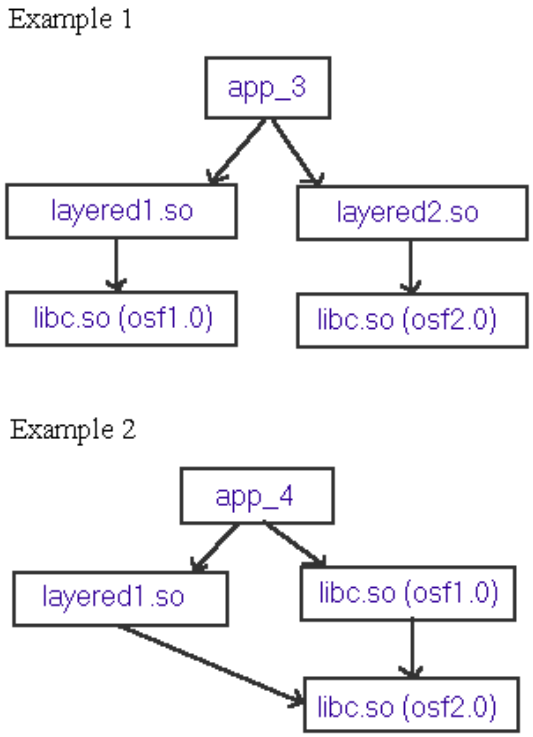


Figure 6–4 shows examples of invalid uses of multiple versions.

Figure 6–4: Invalid Shared Library with Multiple Versions



6.3.2.4 Loading

The executable object is placed in memory first, at the segment base addresses designated by the linker and recorded in the `a.out` header. These addresses are never changed during the lifetime of the executable's image. After the executable file's segments have been mapped into memory, shared library dependencies are loaded. Shared library dependencies are mapped recursively.

The linker chooses quickstart addresses for the text and data regions of shared libraries. The loader attempts to map shared libraries to their quickstart addresses. If this attempt fails because another library has already been mapped to the same address range, the library is relocated to a different address. Note that this problem could be caused by a library mapped by another process. The system tries to map no more than one shared library at a particular virtual address range, system-wide.

Additional dependencies, not present in the library list, can be dynamically loaded using a `dlopen()` call. Again, the loader will attempt to load the library at its quickstart addresses and will relocate it if necessary.

When a shared library is relocated, its text and data segments must move the same distance in memory. By fixing the distance between these segments at link time, the number of dynamic relocations is minimized and restricted to the data segment.

6.3.2.4.1 Dynamic Loading and Unloading

Dependencies can be loaded and unloaded during execution by using the `dlopen()` and `dlclose()` system functions.

The `dlopen()` routine accepts a library name and loads the library and its dependencies. The loader resolves all symbols in all shared objects while processing a `dlopen()` call. If the library was previously loaded, `dlopen()` re-resolves global symbols and returns a handle without loading any new objects.

The loader maintains a count of references made to all shared objects that have been loaded. For example, if `libm.so` is dependent upon `libc.so`, `libc`'s reference count is incremented when the libraries are loaded. This reference counting is part of an effort to ensure that a library is never unloaded prematurely. As an additional precaution to avoid unloading a library that is still needed, the number of existing `dlopen()` handles is tracked by the loader. This `dlopen()` count is incremented each time a `dlopen()` call is made for a particular object.

The `dlclose()` routine unloads a shared library and its dependencies. It accepts a handle that was returned by `dlopen()`.

The `dlclose()` routine will not unload shared libraries that are still in use. Both the `dlopen()` count and the reference count are checked and should be zero before a library is unloaded.

The `dlclose()` routine cannot unload an executable. It is designed for shared libraries only. It also cannot unload a shared library that was not dynamically loaded by `dlopen()`.

Objects with TLS data can be dynamically loaded or unloaded during process execution. A new TLS region is allocated for all existing threads when an object with TLS data is loaded. Similarly, the TLS region will be deallocated for all threads when the object is unloaded.

6.3.3 Dynamic Symbol Information

The dynamic symbol table is created at link time for shared objects. Its primary purpose is to enable dynamic symbol resolution. Run-time address information for dynamic symbols is contained in the GOT section (`.got`).

The dynamic symbol section (`.dynsym`) provides information on globally scoped symbols that are defined or used by the object. This section consists of a table of dynamic symbol entries. The entries are ordered as follows:

1. A single null entry
2. Symbols local to the object
3. Unreferenced global symbols
4. Referenced global symbols (corresponding to GOT entries)
5. Relocations-referenced global symbols (corresponding to special final GOT)

Local symbols are global in scope but are not exported to other objects. The local portion of the dynamic symbol table contains system symbols representing the sections of the object: `.text`, `.data`, and other linker-defined symbols. Typically, they do not have GOT entries.

Unreferenced globals are symbols that can be exported but are not referenced by the defining object. They are present in the dynamic symbol table so that other shared objects can import and use them. Unreferenced globals do not have GOT entries.

Referenced globals are exported and are used internally. Dynamic symbols in this category have global GOT entries.

Global symbols that are referenced only by the object's dynamic relocation entries are grouped at the end of the dynamic symbol table, corresponding to a special final GOT. These symbols require GOT entries to record their run-time addresses used in processing dynamic relocations. This special GOT is only used by the loader and is never directly referenced by the program itself.

All linker-defined TLS symbols (see Section 2.3.7) have dynamic symbol entries.

Note that the dynamic symbol table itself is never relocated; it contains only link-time addresses (in the `st_value` field).

6.3.3.1 Symbol Look-Up

Dynamic symbol look-up is performed by the `dlsym()`(*handle*,*name*) routine. The routine searches for the symbol name beginning in the object associated with the handle. By default, the search is breadth-first. The search is depth-first for objects that were built with the linker's `-B symbolic` option and for objects that were loaded with neither the `RTLD_LOCAL` nor `RTLD_GLOBAL` `dlopen()` flag. If the handle is null, the routine performs a depth-first search beginning at the main executable.

It is important to use the `dlsym()` interface for symbol look-up to avoid using an outdated address. This problem can be caused by an improper compiler assumption that a symbol's address will not change after load time. A symbol's address may be cached as an optimization and not reloaded thereafter. However, that address may be changed during execution as the result of dynamic loading and unloading.

6.3.3.2 Scope and Binding

The concept of scope in the dynamic symbol table differs somewhat from the concept of scope in the debug symbol table because the dynamic symbol table contains only global user-program symbols. The terms "local" and "external" thus have different meanings in this context.

The two scoping levels for symbols in the dynamic symbol table are object scope and process scope. A symbol with object scope is local to the shared object and can only be referenced in the library or executable where it is defined. A symbol with process scope is visible to all program components, and may be referenced anywhere. A symbol with process scope can also be preempted by a higher-precedence definition in another shared object.

Note that the distinction between object scope and process scope does not correspond directly to the local/global symbol division in the dynamic symbol table. All symbols in the local part of the table have object scope, but global dynamic symbols can be internal to the object as well. Another factor, called binding, comes into play.

The possible bind values in the dynamic symbol table are local, global, weak, and duplicate. These values are encoded in the `st_info` field of the dynamic symbol entry. (See Section 6.2.2 for details.)

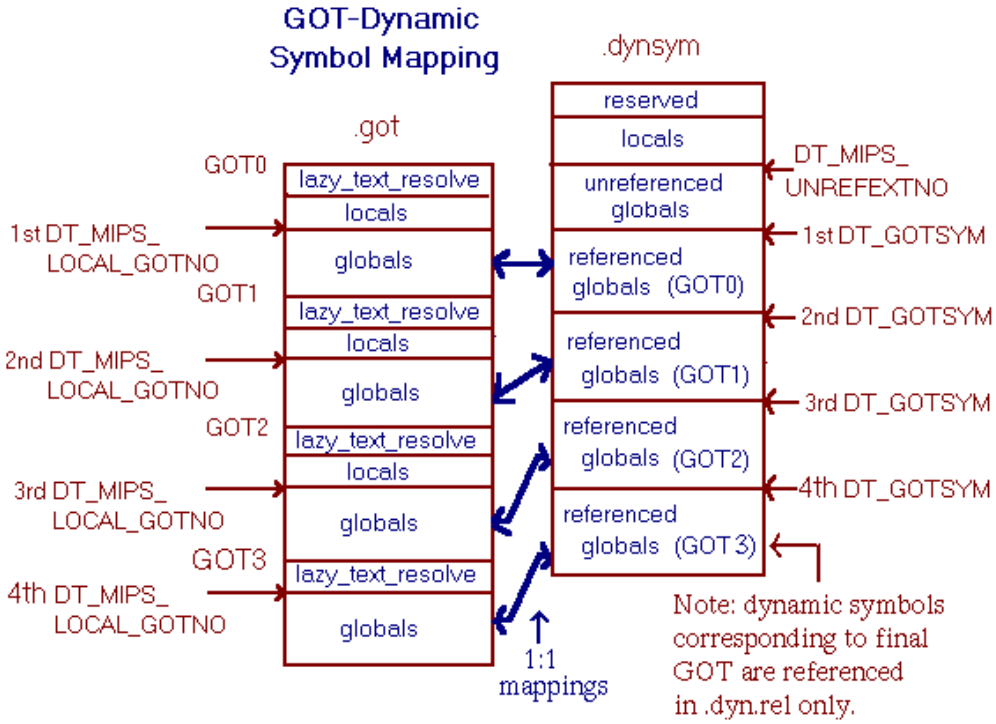
Users are able to designate global symbols as "hidden". In the dynamic symbol table, hidden symbols have a local binding. This representation ensures that they will not be exported from the object and will not preempt any other symbol definition. Also, internal references to hidden symbols will not be preempted. The linker's "`-hidden_symbol` *`symbol`*" option can be used to specify a hidden symbol.

Weak symbols are also a special-case category of global symbols that have the same scope as globals but a lower precedence for symbol resolution conflicts. See Section 6.3.4.2 for details.

6.3.3.3 Multiple GOT Representation

The GOT contains address information for all referenced external symbols in the dynamic symbol table. Observe that the GOT is the source of final, run-time addresses, whereas the symbol table contains only link-time addresses. To access a dynamic symbol, the GOT must be referenced. To associate GOT entries with dynamic symbol table entries, the symbol table and GOT are aligned as shown in Figure 6–5.

Figure 6–5: Dynamic Symbol Table and Multiple-GOT



Note that the GOT also contains entries that do not correspond to dynamic symbols. These are placed at the top of each GOT table.

The maximum number of entries in a GOT is 8189. A single GOT may be sufficient to represent all necessary addresses for an object, but one or more additional GOTs are sometimes required, as illustrated in Figure 6–5. One GOT table can contain entries from multiple input objects, but a single object's entries cannot be split between two tables. The linker also builds a separate, final GOT for relocatable global symbols, referenced only in the dynamic relocation section. These constraints generally result in some unused GOT entries at the bottom of each table.

The loader recognizes a multiple-GOT object by examining the dynamic header. A `DT_GOTSYM` entry exists in the dynamic header for each GOT. This entry holds the index of the first dynamic symbol table entry corresponding to a GOT entry. A `DT_LOCAL_GOTNO` entry exists for each GOT as well. This entry contains the index of the first global entry in that GOT. The number of `DT_GOTSYM` entries and `DT_LOCAL_GOTNO` entries in the dynamic header should match. They are also expected to occur in ascending numerical order.

The first (zero-indexed) entry for every GOT in a multiple-GOT object points to the loader's `lazy_text_resolve()` entry point. In the final GOT (consisting of relocatable symbols), it is present even though it is unused.

Multiple-GOT objects may contain duplicate symbols. A symbol appears only once per GOT, but it can be duplicated in other GOTs. All duplicate symbols, marked in the symbol table as `STB_DUPLICATE`, have an associated primary symbol. The primary symbol is simply the first instance of a duplicate symbol. The `st_size` field for a duplicate symbol is the dynamic symbol table index of the primary symbol. When a symbol is resolved in a multiple-GOT situation, all duplicates must be found and resolved as well.

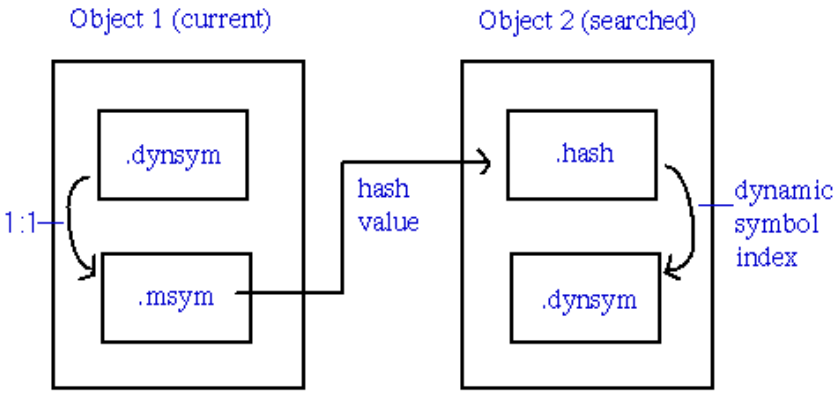
6.3.3.4 Msym Table

The msym table, which is stored in the `.msym` section of a shared object file, maps dynamic symbol hash values to the first of any dynamic relocations for that symbol. This optional section is included for performance reasons by building shared objects with the linker's `-msym` option.

An entry in the msym table contains a hash value and an information field. The information field can be masked to obtain a dynamic relocation index and a flags field. The size of the msym table is the same as the size of the dynamic symbol table; the two tables line up directly and have matching indices.

The msym table is referenced repeatedly when an object is opened. The loader resolves symbols by searching all shared objects for matching definitions. The search requires a hash value computed from the symbol name. The msym table provides precomputed hash values for symbols to avoid the costly hash computation at load time.

Figure 6–6: Msym Table



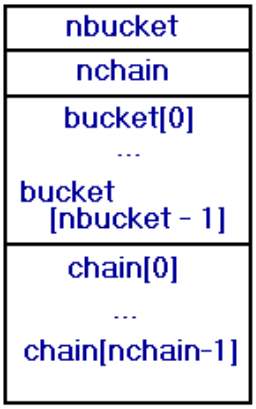
If the `.msym` section is not present in a shared object, the loader will create the table each time that the object is loaded. For this reason, it is often preferable to specify the `.msym` section's inclusion when building shared objects.

6.3.3.5 Hash Table

A hash table, stored in the `.hash` section of a shared object file, provides fast access to symbol entries in the dynamic symbol section. The table is implemented as an array of 32-bit integers.

The hash table has the format shown in Figure 6–7.

Figure 6–7: Hash Table



The entries in the hash table contain the following information:

- The `nbucket` entry indicates the number of entries in the `bucket` array.
- The `nchain` entry indicates the number of entries in the `chain` array.
- The `bucket` and `chain` arrays both hold dynamic symbol table indices, and the entries in `chain` parallel the dynamic symbol table. The value of `nchain` is equal to the number of symbol table entries. Symbol table indices can be used to select `chain` entries.

The hashing function accepts a symbol name and returns the hash value, which can be used to compute a `bucket` index. If the hashing function returns the value *X* for a name, *X*`%`*nbucket* is the bucket index. The hash table entry `bucket[`*X*`%nbucket]` gives an index, `Y`, into the dynamic symbol table.

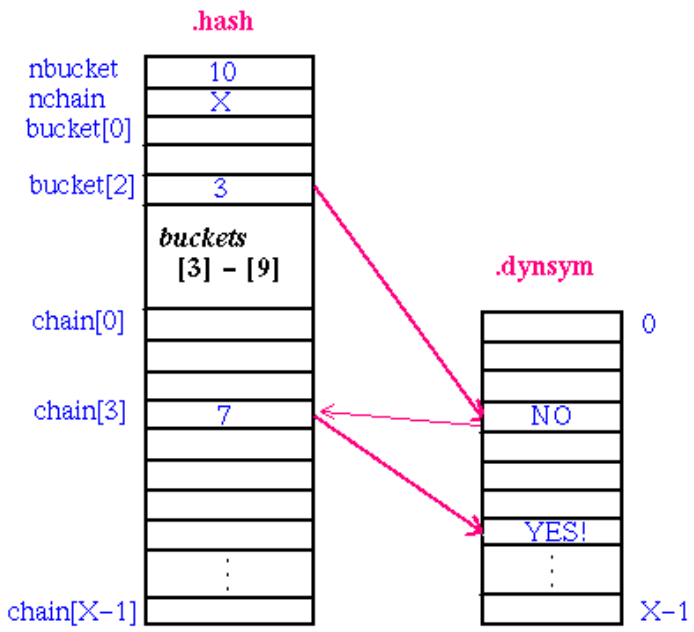
The loader must determine whether the indexed symbol is the correct one. It checks the corresponding dynamic symbol's hash value in the msym table and its name.

If the symbol table entry indicated is not the correct one, the hash table entry `chain[Y]` indicates the next symbol table entry for a dynamic symbol with the same hash value. The indexed symbol is again checked by the loader. If it is incorrect, the same index is used in the `chain` array to try the next symbol that has the same hash value. The `chain` links can be followed in this manner until the correct symbol table entry is located or until the `chain` entry contains the value `STN_UNDEF`.

As an example, assume that a symbol with the hash value 12 is sought. If there are ten buckets, the calculation `12 % 10` gives the bucket index 2, which signifies the third bucket. A bucket index translates into a hash table index as `bucket[i]=hash[i+2]`. If that bucket contains a 3, the dynamic symbol table entry with an index of 3 is checked. If the symbol is incorrect, the hash table entry `chain[3]` is accessed to get the next possible symbol index. A chain index translates into a hash table index as `chain[i]=hash[nbucket+2+i]`. If `chain[3]` is 7, the dynamic symbol table entry with an index of 7 is checked. If it is the correct symbol, the search is successful and halts.

The structures used in this example are shown in Figure 6–8.

Figure 6–8: Hashing Example



6.3.4 Dynamic Symbol Resolution

The dynamic loader must perform symbol resolution for unresolved symbols that remain after link time. A post-link unresolved symbol is one that was not defined in a shared object or in any of the shared object's shared library dependencies searched by the linker. If a dependency is changed before execution or additional libraries are dynamically loaded, the loader will attempt to resolve the symbol.

The linker accepts unresolved symbols when linking shared objects and records them in the dynamic symbol (`.dynsym`) section. The loader recognizes an unresolved symbol by a symbol type of undefined (`st_shndx == SHN_UNDEF`) and a symbol value of zero (`st_value == 0`) in the dynamic symbol table. For such symbols, the GOT value distinguishes imported symbols from symbols that are unresolved across all shared objects.

Table 6-7 gives a rough idea of different categories of symbols and how they are represented in the dynamic symbol table. Run-time addresses are stored in the GOT. They can be pre-computed by the linker and adjusted at load time.

Table 6–7: Dynamic Symbol Categories

Description	Type	Section	Value	GOT
defined item	`STT_OBJECT`, `STT_FUNC`	`SHN_TEXT`, `SHN_DATA`, `SHN_ACOMMON`	address	address
imported function	`STT_FUNC`	`SHN_UNDEF`	0	address (in defining object)
imported data	`STT_OBJECT`	`SHN_UNDEF`	0	address (in defining object)
common	`STT_OBJECT`	`SHN_COMMON`	alignment	address of allocated common (in defining object)

Table 6–7: Dynamic Symbol Categories (cont.)

Description	Type	Section	Value	GOT
unresolved function	STT_FUNC	SHN_UNDEF	0	lazy text stub address
unresolved data	STT_OBJECT	SHN_UNDEF	0	0

The loader performs symbol resolution during initial load of a program. The amount of symbol resolution work required by a program varies (see Section 6.3.4.6).

The loader can also perform dynamic symbol resolution for particular symbols during program execution. If new dependencies are added or existing dependencies are rearranged, externally visible symbols (those with process scope) may be bound to a new address. Rebinding after a `dlopen()` or `dlclose()` call is only performed for symbol references in shared libraries that were not loaded with a `dlopen()` flag of `RTLD_LOCAL` or `RTLD_GLOBAL`.

Unresolved text symbols can be resolved at run time instead of load time (see Section 6.3.4.5).

6.3.4.1 Symbol Preemption and Namespace Pollution

A namespace is a scope within which symbol names should all be unique. In a namespace, a given name is bound to a single item, wherever it may be used. This generic use of the term "namespace" is distinct from the C++ namespace construct, which is discussed in Section 5.3.6.4.

Dynamic executables running on Tru64 UNIX share a namespace with their shared library dependencies. This policy is implemented with symbol preemption. Symbol preemption, also referred to as "hooking", is a mechanism by which all references to a multiply-defined symbol are resolved to the same instance of the symbol.

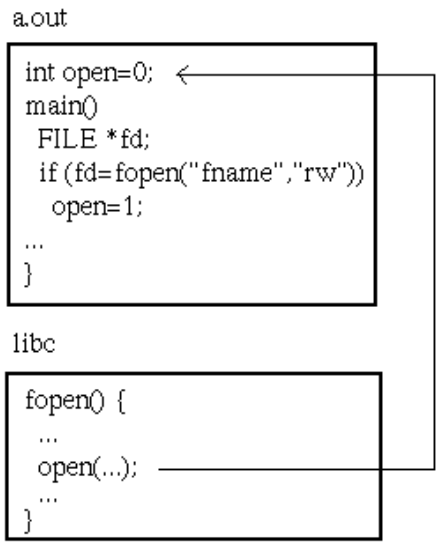
Advantages of symbol preemption include:

- All shared objects use one global namespace.
- Dynamic and static executables behave more consistently.
- Applications can replace library routines to debug, improve, or customize them.

Disadvantages include extra load time for symbol resolution and potential problems resulting from namespace pollution.

Namespace pollution can occur during the use of shared libraries. A library routine may malfunction if it calls or accesses a global symbol that is redefined by another shared library or application. Figure 6–9 presents an example of this situation.

Figure 6–9: Namespace Pollution



Namespace pollution is partly covered by ANSI standards. Namespace conflicts that occur between `libc` and ANSI-compliant programs must not affect the behavior of ANSI-defined functions implemented in `libc`.

The identifiers reserved for use by the library are:

- Names beginning with underscores
- ANSI-defined symbols (`fopen()`, `malloc()`, and so forth)

All other names are available to user programs. User versions of non-reserved identifiers preempt library versions.

Historically, system libraries have used many unreserved symbols. To achieve compliance with the ANSI standard, global symbols have undergone a name change. Documented interfaces have been retained as weak symbols (see Section 6.3.4.2). Their strong counterparts have names that are formed by prepending two underscores to the corresponding weak symbol's name.

Hidden symbols do not cause namespace pollution problems and cannot be preempted because they are not exported from the shared object where they are defined.

The linker options `-hidden_symbol` and `-exported_symbol` turn the hidden attribute on or off for a given symbol name. The options `-hidden` and `-non_hidden` turn the hidden attribute on or off for all subsequent symbols.

TLS data symbols have the same name scope as hidden symbols. The names are not shared among multiple threads.

6.3.4.2 Weak Symbols

Weak symbols are global symbols that have a lower precedence in symbol resolution than other globals. Strong symbols are any symbols that are not marked as weak.

Weak symbols can be used as aliases for other weak or strong symbols. This technique can be useful when it is desirable to provide both a low-precedence name and a high-precedence name for the same data item or procedure. When the weak symbol is referenced, its strong counterpart is the one actually used.

This aliasing approach employing weak symbols is used in `libc.so` to avoid namespace pollution problems. In the example in Figure 6–10, the strong symbol definition in the application takes precedence over the weak library definition, and the program functions properly.

Figure 6–10: Weak Symbol Resolution (I)

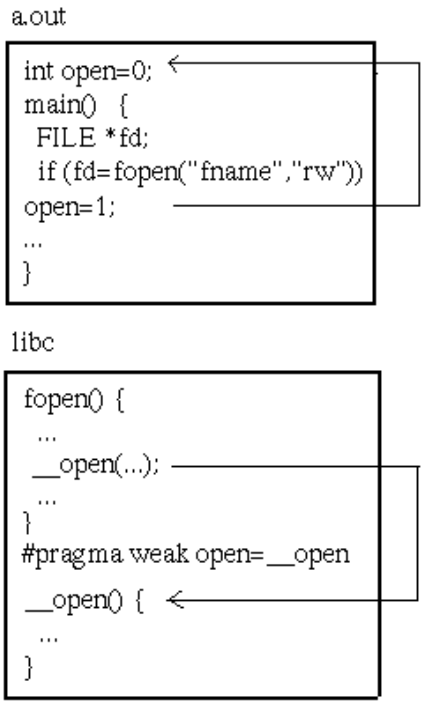
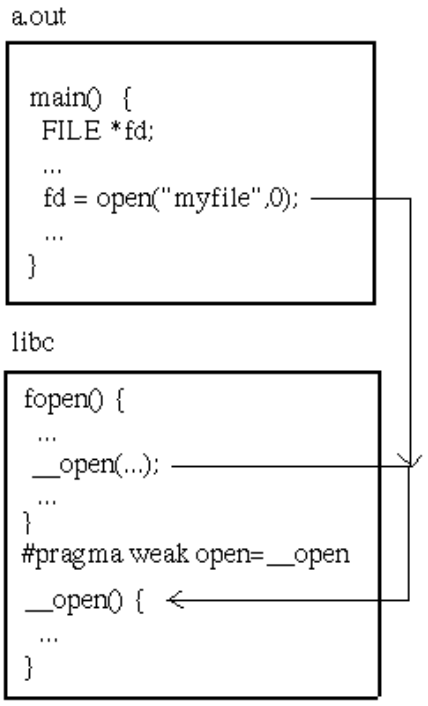


Figure 6–11: Weak Symbol Resolution (II)



If no non-weak `open` symbols were defined, references to `open` would bind to the weak symbol definition in `libc.so`, as shown in Figure 6–11.

Weak symbols can also be used to prevent multiple symbol definition errors or warnings when linking. Neither the linker nor loader require a weak symbol to be aliased to a strong symbol, but the loader will attempt to find a matching strong symbol for any weak symbol it is attempting to resolve.

To find a weak symbol's strong counterpart, the loader follows these steps:

1. Use hash lookup to find *__name*
2. If *__name* is not found or not a match, test each dynamic symbol for matching attributes
3. If a strong matching symbol is found check for a preempting symbol definition in another shared object

Matching symbols will have the same `st_value`, `COFF_ST_TYPE`(`st_info`) and `st_shndx`.

A weak symbol is identified in the dynamic symbol table by a `STB_WEAK` bind value. In the external symbol table, a weak symbol has its `weak_ext` flag set in the `EXTR` entry.

Users can specify weak symbols using the `.weakext` assembler directive or the C `#pragma weak` preprocessor directive.

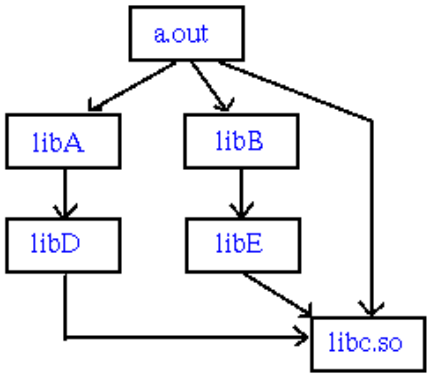
6.3.4.3 Search Order

The dynamic symbol resolution policy, or symbol search order, defines the order in which the loader searches for symbol definitions in a dynamic executable, its

shared library dependencies, and shared libraries added to the process image by `dlopen()`.

Default search order is a breadth-first, left-to-right traversal of the shared library dependency graph.

Figure 6–12: Symbol Resolution Search Order



The search order in Figure 6–12 is: `a.out libA libB libc.so libD libE`

Objects loaded dynamically by `dlopen()` are appended to the search order established at load time. However, `dlopen()` options will determine whether a dynamically loaded object's symbols are visible to objects that do not include it in their dependency lists. See `dlopen`(3) for details.

Alternate search orders can be specified using linker or loader options. The `-B symbolic` linker option marks an object to be loaded with "depth ring" search order. This search order consists of a two-step process:

1. Depth-first search the referencing object and its dependencies
2. Depth-first search from the main executable

Using the depth ring search policy and the dependency graph from Figure 6–12, the search order is:

From	Search Order
`a.out`	`a.out libA libD libc.so libB libE`
`libA`	`libA libD libc.so a.out libB libE`
`libB`	`libB libE libc.so a.out libA libD`
`libD`	`libD libc.so a.out libA libB libE`
`libE`	`libE libc.so a.out libA libD libB`
`libc.so`	`libc.so a.out libA libD libB libE`

6.3.4.4 Precedence

The highest-to-lowest precedence order for dynamic symbol resolution is:

1. Strong text or data
2. Strong largest allocated common
3. Weak data
4. Weak largest allocated common

5. Largest common
6. Weak text

In case (5), the loader allocates the common symbol. This situation only arises when an object containing an allocated common of the same name has been changed between link time and load time or is dynamically unloaded during run time. The linker will always allocate a common storage class symbol, but if there are multiple occurrences of that symbol, the others are retained as unallocated commons.

When symbols have equal precedence, the loader relies on the search order to choose the correct definition for the symbol.

6.3.4.5 Lazy Text Resolution

Lazy text resolution allows programs to execute without resolving text symbols that are never referenced.

Programs with unresolved text symbols are linked with stub routines. When a program or library calls a stub routine, the stub calls the loader's `lazy_text_resolve()` entry point with a dynamic symbol index as an argument. The loader then resolves the text symbol. Subsequent calls will use the true address, which has replaced the stub in the appropriate GOT entry.

The dynamic symbol table does not contain any explicit information that indicates whether a text symbol has a stub associated with it. The loader looks for the following clues instead:

- Symbol's `st_shndx` is `SHN_UNDEF`
- Symbol's `st_value` is zero
- Symbol's GOT entry is not 0 and is in text segment's address range

The environment variable `LD_BIND_NOW` controls the loader's text resolution mode. If the variable has a non-null value, the bind mode is immediate. If the value is null, the bind mode is deferred. Immediate binding requires all symbols to be resolved at load time. Deferred binding allows text symbols to be resolved at run time using lazy text evaluation. The default is deferred binding.

See Section 3.3.3 for related information.

6.3.4.6 Levels of Resolution

Conditions may exist that cause the loader to do more symbol resolution work for some programs than for others. The amount of symbol resolution work that is necessary can have a significant impact on a program's start-up time.

Descriptions of the possible levels of dynamic symbol resolution follow.

Quickstart Resolution

Minimal symbol resolution. For details on quickstart, see Section 6.3.6.

Timestamp Resolution

Moderate symbol resolution. This is used when any of the following are true:

- The executable or one of its dependencies has indirect dependencies that it was not linked with.
- The executable or one of its dependencies has unresolved text symbols that are used in dynamic relocations.
- A shared library dependency was rebuilt so that the timestamp no longer matches the dependency information in the executable.

Checksum Resolution

Extensive symbol resolution. This is used when a shared library dependency has been rebuilt and its checksum no longer matches the dependency information in the executable. The checksum changes if any of the following conditions are met:

- Global symbols are added
- Global symbols are deleted
- Global symbols change from strong to weak or vice versa
- Common storage class symbols' sizes change.

Immediate Binding Resolution

Re-resolve symbols marked `SHN_UNDEF` for immediate binding. This is used by `dlopen()` to apply immediate binding symbol resolution to shared objects that were previously resolved with deferred binding.

6.3.5 Dynamic Relocation

The dynamic relocation section describes all locations that must be adjusted within the object if an object is loaded at an address other than its linked base address.

Although an object may have multiple relocation sections, the linker concatenates all relocation information present in its input objects. The dynamic loader is thus faced with a single relocation table. This dynamic relocation table is stored in the `.rel.dyn` section and is ordered by the corresponding dynamic symbol index.

Offset 0 in the dynamic relocation table is reserved for a null entry with all fields zeroed.

All dynamic relocations must be of the type `R_REFQUAD` or `R_REFLONG`. This simplifies the dynamic relocation process. These two relocation types are sufficient to represent all information that is necessary to accomplish dynamic relocations. Dynamic relocation entries must only apply to addresses in an object's data segment. The object's text segment must not contain any relocatable addresses.

Relocation entries are updated during dynamic symbol resolution. When a dynamic symbol's value changes, any dynamic relocations associated with that symbol must be updated. To update the entries, the relocation value is computed by subtracting the old value of the from the new value. This value is then added to the contents of the relocation targets. The old value of a dynamic symbol is always stored in a GOT entry. The new value of a dynamic symbol is stored in that GOT entry after dynamic relocations are processed.

Relocation types other than `R_REFQUAD` and `R_REFLONG` are not allowed for dynamic relocations because no other relocation types apply to absolute addresses stored in data. Most relocation types apply to values that need to be computed at link time and do not change at run time.

A dynamic executable or shared library may also contain preserved normal relocation sections. If normal relocation entries are present, the loader ignores them.

6.3.6 Quickstart

Quickstart is a loading technique that uses predetermined addresses to run a program that depends on shared libraries. It is particularly useful for applications that rely on shared libraries that change infrequently.

The linker chooses quickstart addresses for all shared library dependencies when a dynamic executable is linked. These addresses are stored in the registry file

normally named so_locations. For details on the shared library registry file, refer to the *Programmer's Guide*.

Any modification to a shared library impairs quickstarting of applications that depend on that library. If a shared library dependency has changed, it may be possible to use the fixso utility to update the application and thus enable quickstart to succeed.

To verify that an application is quickstarted, use the -quickstart_only loader option. For example:

```
% setenv _RLD_ARGS -quickstart_only
% a.out
1834:a.out: /sbin/loader: Fatal Error: quickstart requirements not met
```

Additional information on quickstart is available in the *Programmer's Guide*.

6.3.6.1 Quickstart Levels

Not all shared objects can be successfully quickstarted. If an executable cannot be quickstarted, it still runs, but start up is slower. Quickstarting is possible for programs requiring minimal symbol resolution at load time. A dynamic executable is quickstarted if:

- The object's mapped virtual address matches the quickstart address chosen by the linker.
- The object's dependencies have not been modified incompatibly since the object was linked.
- The object's indirect dependencies are all included as direct dependencies.
- The object's dependencies also meet quickstart criteria.

Each quickstart requirement that is not met by a dynamic executable and its dependencies leads to additional symbol resolution work.

- If all quickstart requirements are met, only undefined and multiply defined symbols need to be resolved.
- If the mapped address differs from the quickstart address, addresses of defined symbols must be adjusted.
- If the timestamp has been changed, external (imported) symbols must be resolved.
- If the checksum has been changed, all symbols must be resolved.

At this point, the timesaving advantage of quickstarting has disappeared.

For quickstart purposes, a link-time shared library matches its associated load-time shared library if the timestamp and checksum are unchanged. If they have been changed, using the fixso tool may remedy the situation and enable quickstart to succeed.

6.3.6.2 Conflict Table

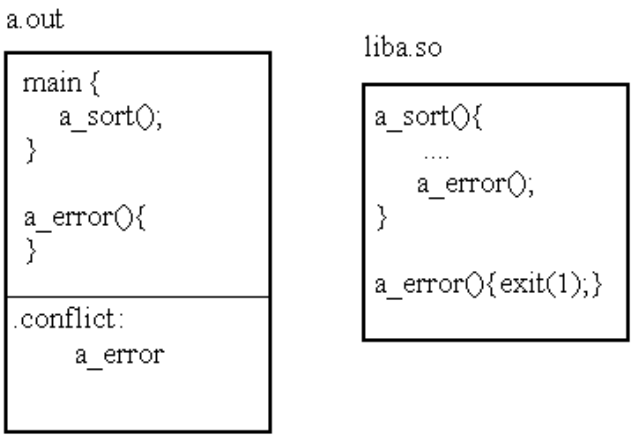
The conflict table, stored in the .conflict section, contains a list of symbols that are multiply defined and must be resolved by the loader. The conflict table is used only when full quickstarting is possible. If any changes preventing quickstart have occurred, the loader resorts to other methods of symbol resolution.

The linker records conflicts in a shared object's .conflict section if a second definition is found for a previously-defined symbol. Common storage class symbols are not considered conflicts unless they are allocated in more than one shared object.

Weak symbols aliased to a newly resolved conflict entry are also treated as conflicts. This means the loader does not have to search for weak symbols matching conflict symbols. The weak symbols are added to the conflict list for the first shared library that defined the symbol in question as well as the library where the conflicting definition was found.

Figure 6–13 shows a simple example of the use of conflict entries.

Figure 6–13: Conflict Entry Example



In this example, the `a.out` executable has been linked with liba.so, and a single conflict has been recorded for the symbol `a_error()`. The conflict is recorded in the executable file at link time because both the executable and shared library define the symbol. At run time, any calls to `a_error()` from `a_sort()` will be preempted by the definition of `a_error()` in the `a.out` executable. Without the conflict entry, the call to `a_error()` would not be preempted properly when `a.out` is quickstarted.

6.3.6.3 Repairing Quickstart

The `fixso` utility updates shared libraries to permit quickstarting of applications that utilize them, even if the libraries have changed since the executable was originally linked against them. Given a shared object as input, it updates the object and its dependencies to make them meet quickstart criteria. The library changes handled by `fixso` are timestamp and checksum discrepancies.

The `fixso` utility creates a breadth-first list of the object's dependencies. It then handles conflicts present in the conflict table. Next, `fixso` resolves globals, updating global symbol values, dynamic relocation entries, and GOT entries where necessary. Lastly, if these actions are successful, `fixso` resets the timestamp and checksum of its target object.

When a dependency is discovered during processing, `fixso` automatically opens the associated object and adds it to the object list if possible. The dependency will be found and opened if it is located in the default library search path, the path indicated by the `LD_LIBRARY_PATH` environment variable, or the path specified in the command line. Otherwise, it may be necessary to run the `fixso` program on the library separately, before fixing the target object.

Some changes made to shared libraries cannot be reconciled by `fixso`. The `fixso` utility does not support:

- Increases in size required in the conflict list (new conflicts)

- Movement of the library in memory
- Discrepancies in interface versions
- Changes to a library's path
- Discrepancies in soname values

7

Comment Section

The Tru64 UNIX object file format supports a mechanism for storing information that is not part of a program's code or data and is not loaded into memory during execution. The comment section (`.comment`) is used for this purpose. Typically, this section contains information that describes an object but is not required for the correct operation of the object. Any kind of object file can have a comment section.

Version Note

Prior to Tru64 UNIX **V5.0** the system linker ignores comment sections in input objects.

7.1 New and Changed Comment Section Features

Tru64 UNIX V5.1 introduces the following new features for comment sections:

- New comment subsection types (see Table 7–1)

Version 3.13 of the object file format introduces the following new features for comment sections:

- New comment subsection types (see Table 7–1)
- Tag descriptors for describing comment subsections (see Section 7.3.4.1)
- Toolversion information for tool specific versioning of object files (see Section 7.3.4.2)

7.2 Structures, Fields, and Values of the Comment Section

All declarations described in this section are found in the header file `scncomment.h`.

7.2.1 Subsection Headers

The comment section begins with a set of header structures, each describing a separate subsection.

```
typedef struct {
        coff_uint       cm_tag;
        coff_uint       cm_len;
        coff_ulong      cm_val;
} CMHDR;
```

SIZE - 16 bytes, ALIGNMENT - 8 bytes

Subsection Header (`CMHDR`) Fields

`cm_tag` — Identifies the type of data in this subsection of the `.comment` section. This value may be recognized by system tools. If it is not recognized, generic processing occurs, as described in Section 7.3.3. Refer to Table 7–1 for a list of system-defined comment tags.

`cm_len` — Specifies the unpadded length (in bytes) of this subsection's data. If `cm_len` is zero, the data is stored in the `cm_val`

field. The padded length is this value rounded up to the nearest 16-byte boundary.

`cm_val` — Provides either a pointer to this subsection's data or the data itself. If `cm_len` is nonzero, `cm_val` is a relative file offset to the start of the data from the beginning of the `.comment` section. If `cm_len` is zero, this field contains all data for that subsection. In the latter case, the size of the data is considered to be the size of the field (8 bytes).

Table 7–1: Comment Section Tag Values

Tag	Value	Description
CM_END	0	Last subsection header. Must be present.
CM_CMSTAMP	3	First subsection header. The cm_val field contains a version stamp that identifies the version of the comment section format. The current definition of CM_VERSION is 0. Must be present.
CM_COMPACT_RLC	4	Compact relocation data. See Section 4.4 for details.
CM_STRSPACE	5	**(V5.0 -)** Generic string space.
CM_TAGDESC	6	**(V5.0 -)** Subsection containing flags that tell tools how to process unfamiliar subsections. See Section 7.2.2 and Section 7.3.4.1.
CM_IDENT	7	**(V5.0 -)** Identification string. Reserved for system use.
CM_TOOLVER	8	**(V5.0 -)** Tool-specific version information. See Section 7.3.4.2.
CM_II_CHECKSUMS	9	**(V5.1 -)** Checksum data for Atom incremental instrumentation. Reserved for future use.
CM_II_ATOMARGS	10	**(V5.1 -)** Atom argument data for incremental instrumentation. Reserved for future use.
CM_II_TOOLARGS	11	**(V5.1 -)** Atom tool argument string for incremental instrumentation. Reserved for future use.
CM_II_ANALADDRS	12	**(V5.1 -)** Analysis address information for Atom incremental instrumentation. Reserved for future use.
CM_FLOAT_TYPE	13	**(not supported)** Floating point type used in compilation. The value field will be set to one of: F_TANDEM_FLOATTYPE_UNUSED, F_TANDEM_FLOATTYPE_TANDEM, F_TANDEM_FLOATTYPE_NEUTRAL, F_TANDEM_FLOATTYPE_IEEE
CM_II_OBJID	14	**(V5.1 -)** Object identification number for Atom incremental instrumentation. Reserved for future use.
CM_LINKERDEF	15	**(V5.1 -)** Relocation information for linker-defined symbols. See Section 4.5
CM_LOUSER	0x80000000	Beginning of user tag value range (inclusive).
CM_HIUSER	0xffffffff	End of user tag value range (inclusive).

Version Note

The CM_FLOAT_TYPE tag is reserved for use on Tandem big-endian systems. It is not supported on Tru64 UNIX.

7.2.2 Tag Descriptor Entry

Tag descriptors are used to specify behavior for tools that modify object files and potentially affect the accuracy of comment subsection data. They are especially useful as processing guidelines for tools that do not understand certain subsections. Tools which have specific knowledge of certain comment subsection types can ignore the tag descriptor settings for subsection type. The tag descriptors are stored in the raw data of the CM_TAGDESC subsection. See Section 7.3.4.1 for more information.

```
typedef struct {
        coff_uint       tag;
        cm_flags_t      flags;
} cm_td_t;
```

SIZE - 8 bytes, ALIGNMENT - 4 bytes

Tag Descriptor Fields

tag	Tag value of subsection being described.
flags	Flag settings. See Section 7.2.2.1.

7.2.2.1 Comment Section Flags

```
typedef struct {
        coff_uint       cmf_strip   :3;
        coff_uint       cmf_combine :5;
        coff_uint       cmf_modify  :4;
        coff_uint       reserved    :20;
} cm_flags_t;
```

SIZE - 4 bytes, ALIGNMENT - 4 bytes

Comment Section Flags Fields

cmf_strip	Tells tools that perform stripping operations whether to strip comment section data.
cmf_combine	Tells tools how to combine multiple input subsections of the same.
cmf_modify	Tells tools that modify single object files how to rewrite the input comment section in the output object.

Table 7–2: Strip Flags

Name	Value	Description
CMFS_KEEP	0x0	Do not remove this subsection when performing stripping operations.
CMFS_STRIP	0x1	Remove this subsection if stripping the entire symbol table.
CMFS_LSTRIP	0x2	Remove this subsection if stripping local symbolic information or if fully stripping the symbol table.

Table 7–3: Combine Flags

Name	Value	Description
CMFC_APPEND	0x0	Concatenate multiple instances of input subsection data.
CMFC_CHOOSE	0x1	Choose one instance of input subsection data (randomly).
CMFC_DELETE	0x2	Do not output this subsection.

Table 7–3: Combine Flags (cont.)

Name	Value	Description
CMFC_ERRMULT	0x3	Raise an error if multiple instances of this subsection are encountered as input.
CMFC_ERROR	0x4	Raise an error if a subsection of this type is encountered as input.

Table 7–4: Modify Flags

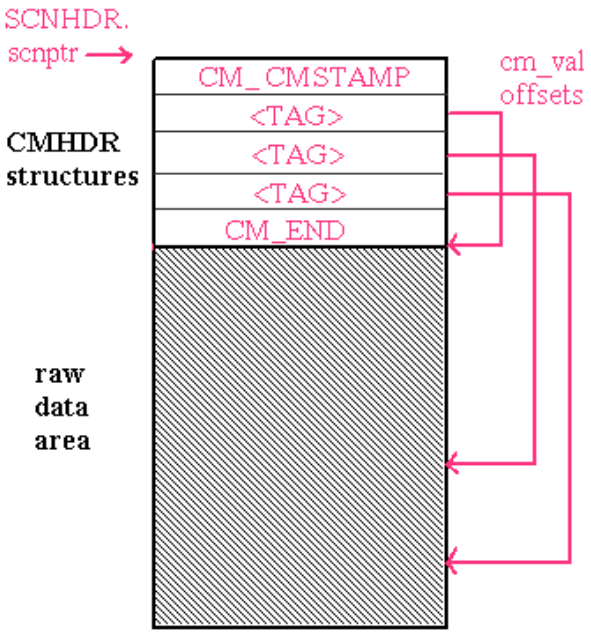
Name	Value	Description
CMFM_COPY	0x0	Copy this subsection's data unchanged from the input object to the output object.
CMFM_DELETE	0x1	Do not output a subsection of this type.
CMFM_ERROR	0x2	Raise an error if a subsection of this type is encountered as input.

7.3 Comment Section Usage

7.3.1 Comment Section Formatting Requirements

The comment section is divided between subsection header structures and an unstructured raw data area. The subsection headers contain tags that identify the data stored in the subsequent raw data area. Each header describes a different subsection. The raw data for all subsections follows the last header, as shown in Figure 7–1.

Figure 7–1: Comment Section Data Organization



Begin and end marker tags are used to denote the boundaries of the structured portion of the comment section. The begin marker is CM_CMSTAMP, which contains a comments section version stamp, and the end marker is CM_END. If either of these

headers is missing or the version indicated by the value of CM_CMSTAMP is invalid, the comment section is considered invalid.

The ordering of the subsection headers and their corresponding raw data do not need to match. Nor is the density of the raw data area guaranteed. However, all subsection headers must be contiguous: no other data can be placed between them. Furthermore, a one-to-one relationship must exist between the subsection headers that point into the raw data and the data itself. Subsection raw data must not overlap.

The interpretation of the cm_val field depends on the cm_len field. When cm_len is zero, cm_val contains arbitrary data whose interpretation depends on the value in the cm_tag field. When cm_len is non-zero, cm_val contains a relative file offset from the start of the comment section into the raw data area.

The start of data allocated in the raw data area must be octaword (16-byte) aligned for each subsection. Zero-byte padding is inserted at the end of each data item as necessary to maintain this alignment. The value stored in cm_len represents the actual length of the data, not the padded length. Tools manipulating this data must calculate the padded length.

7.3.2 Comment Section Contents

The comment section can contain various types of information. Each type of information is stored in its own subsection of the comment section. Each subsection must have a unique tag value within the section.

The comment section can include supplemental descriptive information about the object file. For instance, the tag CM_IDENT points to one or more ASCII strings in the raw data area that serve to identify the module. Use of this tag is reserved for compilation system object producers such as compilers and assemblers.

User-defined comment subsections are also possible. The CM_LOUSER and CM_HIUSER tags delimit the user-defined range of tag values. Potential uses include product version information and miscellaneous information targeted for specific consumers.

Although no restrictions are put on the type or amount of information that can be placed in the comment section, it is important to be aware that users have the capability to remove the section entirely (by using the command ostrip -c) and that object file consumers may ignore its presence.

The minimal valid comment section consists of a CM_CMSTAMP header and a CM_END header. Because no structure field in the object file format holds the number of subsections in the comment section, the presence of the CM_END header is crucial. Without it, a consumer cannot determine the number of subsections present.

7.3.3 Comment Section Processing

Many tools that handle objects read or write the comment section. Some tools, such as the linker and mcs, perform special processing of comment section data. Others may be interested in extracting certain subsections. Most object-handling tools provided on the system access the comment section to check for tool-specific version information (see Section 7.3.4.2).

The linker is both a consumer and producer of the comment section. As with other object file sections, the linker must combine multiple input comment sections to form a single output section. When comment sections are encountered in input object files, the linker reads subsection headers and merges the raw data according to its own defaults and the flag settings of any tag descriptors that are present.

The mcs utility provides comment section manipulation facilities. This tool allows users to add, modify, delete, or print the comment section from the command line. The mcs tool can only process objects that already have a .comment section header, but actual .comment section data is not required. Compilers and assemblers frequently write object files which have zero-sized .comment sections.

The operations performed by mcs do not affect the object's suitability for linking or execution. See the mcs(1) man page for more details.

Stripping tools, such as strip and ostrip, also process the comment section. They read the tag descriptors to determine what subsections to remove. The cmf_strip field of the tag descriptor specifies the stripping behavior. If the cmf_strip field is set to CMFS_STRIP that subsection will be removed if an object is fully stripped. If the cmf_strip field is set to CMFS_LSTRIP for a particular subsection type, that subsection will be removed if an object is fully stripped or locally stripped.

7.3.4 Special Comment Subsections

Comment subsections can have particular structures or semantics that a consumer must know to be able to read and process them correctly. Two system-defined subsections with special formatting and processing rules are the tag descriptors (CM_TAGDESC) and the tool-specific version information (CM_TOOLVER).

Another special subsection contains compact relocation data (CM_COMPACT_RLC). This topic is covered in Section 4.4.

7.3.4.1 Tag Descriptors (CM_TAGDESC)

Version Note

Tag descriptors are supported in object format **V3.13** and greater.

The tag descriptor subsection contains a table of tags and their corresponding flag settings. This information tells tools how to handle unfamiliar subsections. The CM_TAGDESC subsection may not be present, and if present, it may not contain entries for subsections that are present. Also, a tag descriptor may be present for a subsection that is not found in the object.

A list of possible tag descriptor flag settings can be found in Section 7.2.2.1. Flag settings are divided into three categories based on the categories of object tools that need to modify the comment section:

1. Tools that strip object files
2. Tools that combine multiple instances of comment section data
3. Tools that modify and rewrite single object files

The default flag settings for user subsections that do not have tag descriptors are CMFS_KEEP, CMFC_APPEND, and CMFM_COPY. Tools that strip or rewrite objects should not modify subsection data for comment subsections marked with these default flag settings. A tool that combines multiple instances of subsection data, should concatenate the subsection raw data for same-type input subsections marked with the default flag settings.

A tool can ignore the tag descriptor flags and default flag settings for a subsection if it recognizes the subsection type and understands how to process its data.

Some of the system tags have different defaults. These are shown in Table 7–5. However, tag descriptors in the CM_TAGDESC subsection can be used to override the default settings for system tag values as well as user tag values.

Table 7–5: Default System Tag Flags

Tag	Default Flag Settings
CM_END	CMFS_KEEP, CMFC_CHOOSE, CMFM_COPY
CM_CMSTAMP	CMFS_KEEP, CMFC_CHOOSE, CMFM_COPY
CM_COMPACT_RLC	CMFS_STRIP, CMFC_DELETE, CMFM_DELETE
CM_STRSPACE	CMFS_KEEP, CMFC_APPEND, CMFM_COPY
CM_TAGDESC	CMFS_KEEP, CMFC_CHOOSE, CMFM_COPY
CM_IDENT	CMFS_KEEP, CMFC_APPEND, CMFM_COPY
CM_TOOLVER	CMFS_KEEP, CMFC_CHOOSE, CMFM_COPY
CM_II_CHECKSUMS	CMFS_STRIP, CMFC_ERROR, CMFM_COPY
CM_II_ATOMARGS	CMFS_STRIP, CMFC_ERROR, CMFM_COPY
CM_II_TOOLARGS	CMFS_STRIP, CMFC_ERROR, CMFM_COPY
CM_II_ANALADDRS	CMFS_STRIP, CMFC_ERROR, CMFM_COPY
CM_II_OBJID	CMFS_STRIP, CMFC_ERROR, CMFM_COPY
CM_LINKERDEF	CMFS_STRIP, CMFC_ERROR, CMFM_DELETE

Because the size of a tag descriptor entry is fixed, a consumer can determine the number of entries by dividing the size of the subsection by the size of a single tag descriptor (see Section 7.2.2). If cm_len is set to zero, a single tag descriptor is stored as immediate data.

7.3.4.2 Tool Version Information (CM_TOOLVER)

Version Note

Tool versions are supported in object format **V3.13** and greater.

The CM_TOOLVER subsection contains tool-specific version entries for system tools that process object files. If present, this subsection may have any number of entries. This subsection can also be used to record version information for non-system tools.

Each tool version entry consists of three parts:

1. Tool name (null-terminated character string)
2. Tool version number (unsigned 8-byte unaligned numeric value)
3. Printable version string (null-terminated character string)

The number of tool version entries cannot be determined from the subsection header because the entries vary in length. The data must be read until the entry sought is found or until the end of the subsection's data is reached.

The encoding of the tool version number is generally tool dependent. The only requirement is that the value, viewed as an unsigned long, must be monotonically increasing with time.

Typically, an object file consumer uses the tool version information to verify its ability to handle an input object file. The consumer uses an API (see libst reference pages) to look for a tool version entry with a tool name matching its own (part one of the entry). If found, the version number (part two of the entry) must not exceed the version number of the tool. Otherwise, the tool will print a message instructing the user to obtain the newer version of the tool, using the printable version string (part three of the entry). This mechanism can be used as a warning to customers of a necessary upgrade to a newer release of a product, for instance.

As an example, a compiler might produce object files with new symbol table information that causes an old version of the ladebug debugger to produce a fatal error. To provide more user-friendly behavior for old versions of the debugger, the compiler outputs a tool version entry:

1. "ladebug"
2. 2
3. "5.0A-BL5"

This entry occupies 25 bytes. The debugger recognizes its name in the entry and compares the version number "2" with the version number it was built with. (Note that the version number is most likely meaningless to an end user of the debugger.) In this case, assume that the installed debugger's version number is "1". The message "Please obtain version 5.0A-BL5" is output to the user.

Note that the numeric tool version number can be unaligned. This is an exception to the general rule requiring alignment of numeric data.

8

Archives

An archive is a collection of files stored and treated as a single entity. They are used most commonly to implement libraries of relocatable objects. These libraries simplify linking in a program development environment by allowing the manipulation of one archive file instead of dozens or hundreds of object files.

This chapter covers the archive file format and usage. The archiver is the tool used to create and manage archives. See ar(1) for more information on its facilities.

8.1 New and Changed Archive Features

Tru64 UNIX V5.0 introduces archive support for extended user and group ids (see ar_uid and ar_gid in Section 8.2.2)

8.2 Structures, Fields, and Values for Archives

All declarations in this section are from the header file ar.h.

See Section 8.3.1 for more information on the organization of object file contents.

8.2.1 Archive Magic String

The archive magic string identifies a file as an archive.

```
#define ARMAG "!<arch>\n"
#define SARMAG 8
```

8.2.2 Archive Header

```
struct ar_hdr {
        char    ar_name[16];
        char    ar_date[12];
        char    ar_uid[6];
        char    ar_gid[6];
        char    ar_mode[8];
        char    ar_size[10];
        char    ar_fmag[2];
} AR_HDR;
```

SIZE - 60 bytes, ALIGNMENT - 1 byte

Archive Header Fields

ar_name

File member name, blank-terminated if the length of the name is less than 16 bytes.

File member names that are 16 characters or longer are stored in the special file member called the file member name table. In that case, this field contains */offset* where *offset* indicates the byte offset of the file name within the table. The offset is a decimal number.

The prefix ARSYMPREF, defined as the 16-byte blank-terminated character string _____64ELEL_, is stored in this field for the special file member called the symbol definitions (symdef) file and is used to identify that file. The ar tool marks an out of date

symdef file by changing the last L in the name to an X (_____64ELEX_).

The blank-terminated name // is stored in this field to identify the file member name table.

ar_date — File member date (decimal).

ar_uid — File member user id (decimal).

For a file with a user id greater than USHRT_MAX (65535U), this field will contain //*value* where *value* is a 4-byte unsigned integer.

Version Note

Large user ids are supported in Tru64 UNIX **V5.0** and greater.

ar_gid — File member group id (decimal).

For a file with a group id greater than USHRT_MAX (65535U), this field will contain //*value* where *value* is a 4-byte unsigned integer.

Version Note

Large group ids are supported in Tru64 UNIX **V5.0** and greater.

ar_mode — File member mode (octal).

ar_size — File member size (decimal). Sizes reflect padding for the symdef file and the file name table, but not for file member contents. File members always start on even byte boundaries. Therefore, if the ar_size field indicates an odd length, it should be rounded up to the next even number.

ar_fmag — Archive magic string. The possible values are shown in Table 8–1.

Table 8–1: Archive Magic Strings

Symbol	Value	Meaning
ARFMAG	"'\n"	File member. May be a special file member or any type of file other than a compressed object file.
ARFZMAG	"Z\n"	Compressed object file member.

General Note:

Archive header fields are stored as character strings and must be converted to numeric types.

8.2.3 Hash Table (`ranlib`) Structure

This structure is found only inside the special file member called the "symdef file". See Section 8.3.2 for related information.

```
struct  ranlib {
        union {
            int      ran_strx;
        } ran_un;
        int          ran_off;
};
```

SIZE - 8 bytes, ALIGNMENT - 4 bytes

Ranlib Structure Fields

`ran_strx`	Symdef string table index for this symbol's name.
`ran_off`	Byte offset from the beginning of the archive file to the archive header of the member that defines this symbol.

General Note:

The `ran_un` union of this structure has only one field, as shown, for historical reasons.

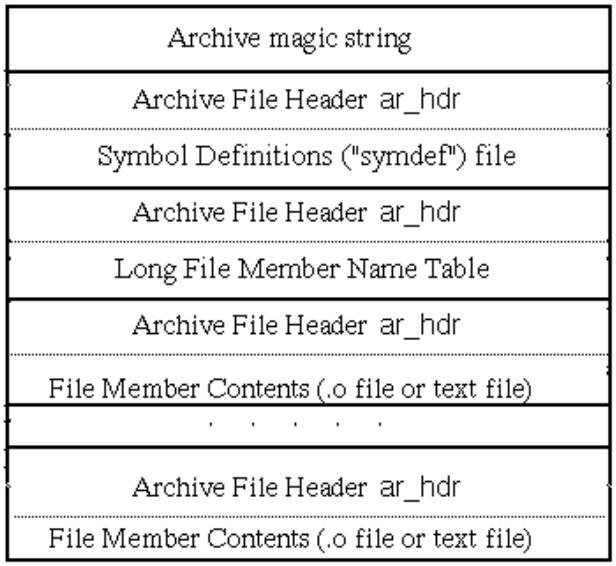
8.3 Archive Implementation

8.3.1 Archive File Format

The first `SARMAG` (8) bytes in an archive file identify it as an archive. To verify that a file is an archive, these bytes should be compared with the archive magic string, defined as `ARMAG` in the header file `ar.h`.

An archive file consists of the magic string followed by multiple file members, each of which is preceded by an archive file member header. File members can be object files, compressed object files, text files, or files of any other type, and an archive can contain a mix of file types. A file member can also be one of two special file members: the symbol definition (or symdef file) or the file member name table. Figure 8–1 illustrates this file layout.

Figure 8–1: Archive File Organization



The symdef file, if present, is the first file member of an archive. Section 8.3.2 for details on the symdef file.

The file member name table consists of file member names that are too long to fit into the 16-byte name field of the archive header. If no file member names are 16 characters or longer, this table is not created. If the table is needed, it is either the first file member or the second (following the symdef file.

The member header for the file name table might look like this:

```
struct arhdr {
        ar_name = "//              ";
        ar_date = "871488454    ";
        ar_uid  = "0     ";
        ar_gid  = "0     ";
        ar_mode = "0      ";
        ar_size = "54        ";
        ar_fmag = "'\n";
}
```

Names in the file member name table are separated by a slash (`/`) and a linefeed (`\n`). For example, the contents of the file name table for an archive with three long object file names might look like this:

```
st_cmrlc_basic.o/
st_cmrlc_print.o/
st_object_type.o/
```

The file member header for a file member whose name is stored in the file name table (in this case, the object `st_cmrlc_print.o`) might look like this:

```
struct arhdr {
        ar_name = "/18             ";
        ar_date = "871414955    ";
        ar_uid  = "9442     ";
        ar_gid  = "0     ";
        ar_mode = "100600  ";
        ar_size = "47296     ";
        ar_fmag = "'\n";
}
```

8.3.2 Symdef File Implementation

The symdef file contains external symbol information for all object file members within an archive. When present, the symdef file is the first file member of the archive. The member header for an up-to-date symdef file might look as follows:

```
struct arhdr {
        ar_name = "_________64ELEL_  ";
        ar_date = "871488454    ";
        ar_uid = "0     ";
        ar_gid = "0     ";
        ar_mode = "0      ";
        ar_size = "8238      ";
        ar_fmag = "'\n";
}
```

The symdef file is present if at least one archive file member is an object file. The linker uses it when searching for symbol definitions, as long as the file is up to date. Whenever an archive is modified, the symdef file must be updated or its member name must be changed to reflect the fact that it is outdated (see Section 8.2.2).

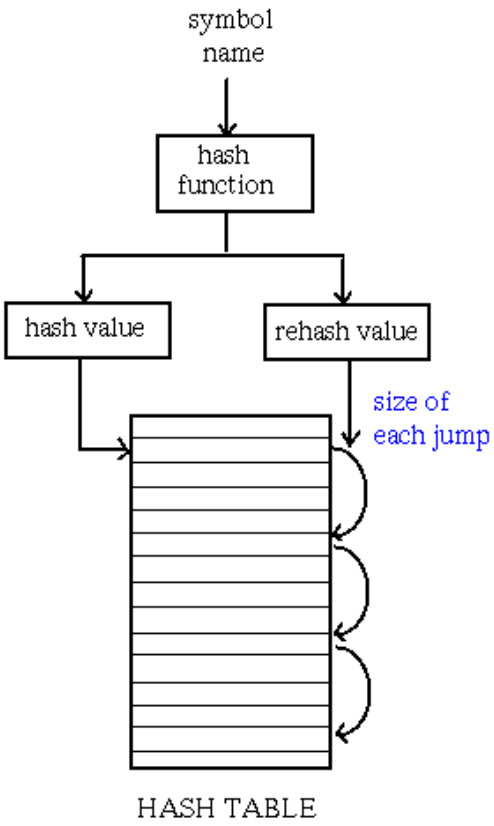
The symdef file consists of a hash table and a string table. The contents of the symdef file are as follows:

1. hash table size: 4 bytes indicating the number of `ranlib` structures in the hash table
2. hash table: array of `ranlib` structures
3. string table size: 4 bytes indicating the size, in bytes, of the symdef string table
4. string table: string space containing symbol names

At a minimum, the symdef file should contain the sizes of the hash and string tables, even if the tables are empty.

The hash table contains a `ranlib` structure for each externally visible symbol defined in any of the archive file members. The total size of the hash table is two times the number of symbols rounded to the next highest power of two. Each symbol has a private hash chain that is used for symbol lookup, as shown in Figure 8–2.

Figure 8–2: Symdef File Hash Table



The hash function produces two values for any name it is given: a hash value and a rehash value. The hash value is used for the first lookup. If the symbol found is not the right one, the rehash value is used for chaining. The chain is followed until the correct symbol is found or until the search returns to the symbol where it began.

The linker uses the hash structure field `ran_off` to locate a symbol's definition in the archive. This field contains the byte offset from the beginning of the archive file to the file member header of the member containing the symbol's definition.

Note that symbols appear only once in the symdef file hash table, regardless of how many file members define them.

8.4 Archive Usage

8.4.1 Role As Libraries

One important use of archives is to serve as static libraries that programs can link against. Such archives contain a collection of relocatable object files that can be selectively included in an executable image as required. Archive libraries are the only libraries used in creating static executables. They can also be used in conjunction with shared libraries in dynamic executables.

The linker searches archive libraries during symbol resolution. See the *Programmer's Guide* or `ld`(1) for more information.

8.4.2 Portability

The archive file format is designed to meet current UNIX standards in order to assure portability with other UNIX systems.

The format of compressed object files within archives is specific to Tru64 UNIX. See Section 1.4.3 for details.

9

Symbol Table Examples

This chapter contains sample programs that illustrate the symbol table representations of various language constructs. The examples are organized by source language and each consists of a program listing and the partial symbol table contents for that program. The system symbol table dumpers `stdump`(1) and `odump`(1) were used to produce the output.

9.1 C

9.1.1 Unnamed Structure

See Section 5.3.8.3 for related information.

Source Listing

```
Struct S1 {
   int abc;
   struct {int x; signed int y; unsigned int z;};
   int rst;
} s1;
```

Symbol Table Contents

```
File 0 Local Symbols:

 0. (0)(   0)  unname.c   File     Text      symref  12
 1. (1)( 0xc)             Block    Info      symref  6
 2. (2)(   0)  x          Member   Info      [ 3] int
 3. (2)(0x20)  y          Member   Info      [ 3] int
 4. (2)(0x40)  z          Member   Info      [ 4] unsigned int
 5. (1)(   0)             End      Info      symref 1
 6. (1)(0x14)  S1         Block    Info      symref 11
 7. (2)(   0)  abc        Member   Info      [ 3] int
 8. (2)(0x20)             Member   Info      [ 5] struct(file 0,
                                                index 2)
 9. (2)(0x80)  rst        Member   Info      [ 3] int
10. (1)(   0)  S1         End      Info      symref 6
11. (0)(   0)  unname.c   End      Text      symref 0

Externals Table:

 0. (file 0)(0x14)  s1    Global   Common    [7] struct(file 0,
                                                 index 6)
```

9.2 C++

9.2.1 Base and Derived Classes

See Section 5.3.8.6 for related information.

Source Listing

```
#include <iostream.h>

class employee {
        char *name;
        short age;
        short deparment;
        int salary;

public:

        static int stest;
        employee *next;
```

```
        void print() const;
};

class manager : public employee {
        employee emp;
        employee *group;
        short level;

public:

        void print() const;
};

void employee::print() const
{
  cout << "name is " << name << '\n';
}

void manager::print() const
{
  employee::print();
}

void f()
{
  manager m1,m2;
  employee e1, e2;
  employee *elist;

  elist=&m1;
  m1.next=&e1;
  e1.next=&m2;
  m2.next=&e2;
  e2.next=0;
}
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) bs6.cxx    File       Text       symref 51
  1. ( 1)(   0) employee   Tag        Info       [25] Class(extended file 0,
                                                      index 2)
  2. ( 1)(0x18) employee   Block      Info       symref 17
  3. ( 2)(   0) name       Member     Info       [28] Pointer to char
  4. ( 2)(0x40) age        Member     Info       [29] short
  5. ( 2)(0x50) deparment  Member     Info       [29] short
  6. ( 2)(0x60) salary     Member     Info       [30] int
  7. ( 2)(0x80) next       Member     Info       [31] Pointer to
                                                      Class(extended file 0,
                                                      index 2)
  8. ( 2)(   0) employee::stest
                           Static     Info       [30] int
  9. ( 2)(   0) employee::print(void) const
                           Proc       Info       [43] endref 12, void
 10. ( 3)(   0) this       Param      Info       [40] Const Pointer to Const
                                                      Class(extended file 0,
                                                      index 2)
 11. ( 2)(   0) employee::print(void) const
                           End        Info       symref 9
 12. ( 2)(   0) employee::operator =(const employee&)
                           Proc       Info       [57] endref 16, Reference
                                                      Class(extended file 0,
                                                      index 2)
 13. ( 3)(   0) this       Param      Info       [48] Const Pointer to
                                                      Class(extended file 0,
                                                      index 2)
 14. ( 3)(   0)            Param      Info       [54] Reference Const
                                                      Class(extended file 0,
                                                      index 2)
 15. ( 2)(   0) employee::operator =(const employee&)
                           End        Info       symref 12
 16. ( 1)(   0) employee   End        Info       symref 2
 17. ( 1)(   0) manager    Tag        Info       [61] Class(extended file 0,
                                                      index 18)
 18. ( 1)(0x40) manager    Block      Info       symref 31
 19. ( 2)(   0) employee   Base Class Info       [25] Class(extended file 0,
                                                      index 2)
 20. ( 2)(0xc0) emp        Member     Info       [25] Class(extended file 0,
                                                      index 2)
 21. ( 2)(0x180) group     Member     Info       [31] Pointer to Class(extended
```

```
                                                    file 0, index 2)
 22. ( 2)(0x1c0) level     Member     Info       [29] short
 23. ( 2)(   0) manager::print(void) const
                           Proc       Info       [73] endref 26, void
 24. ( 3)(   0) this       Param      Info       [70] Const Pointer to Const
                                                    Class(extended file 0,
                                                    index 18)
 25. ( 2)(   0) manager::print(void) const
                           End        Info       symref 23
 26. ( 2)(   0) manager::operator =(const manager&)
                           Proc       Info       [90] endref 30, Reference
                                                    Class(extended file 0,
                                                    index 18)
 27. ( 3)(   0) this       Param      Info       [81] Const Pointer to
                                                    Class(extended file 0,
                                                    index 18)
 28. ( 3)(   0)            Param      Info       [87] Reference Const
                                                    Class(extended file 0,
                                                    index 18)
 29. ( 2)(   0) manager::operator =(const manager&)
                           End        Info       symref 26
 30. ( 1)(   0) manager    End        Info       symref 18
 31. ( 1)(   0) employee::print(void) const
                           Proc       Text       [414] endref 36, void
 32. ( 2)( 0x9) this       Param      Register   [416] Const Pointer to Const
                                                     Class(extended file 0,
                                                     index 2)
 33. ( 2)(0x18)            Block      Text       symref 35
 34. ( 2)(0x60)            End        Text       symref 33
 35. ( 1)(0x70) employee::print(void) const
                           End        Text       symref 31
 36. ( 1)(0x70) manager::print(void) const
                           Proc       Text       [419] endref 41, void
 37. ( 2)( 0x9) this       Param      Register   [421] Const Pointer to Const
                                                     Class(extended file 0,
                                                     index 18)
 38. ( 2)(0x18)            Block      Text       symref 40
 39. ( 2)(0x2c)            End        Text       symref 38
 40. ( 1)(0x3c) manager::print(void) const
                           End        Text       symref 36
 41. ( 1)(0xac) f(void)    Proc       Text       [424] endref 50, void
 42. ( 2)( 0x8)            Block      Text       symref 49
 43. ( 3)(-64)  m1         Local      Abs        [61] Class(extended file 0,
                                                    index 18)
 44. ( 3)(-128) m2         Local      Abs        [61] Class(extended file 0,
                                                    index 18)
 45. ( 3)(-152) e1         Local      Abs        [25] Class(extended file 0,
                                                    index 2)
 46. ( 3)(-176) e2         Local      Abs        [25] Class(extended file 0,
                                                    index 2)
 47. ( 3)(   0) elist      Local      Register   [31] Pointer to Class(extended
                                                    file 0, index 2)
 48. ( 2)(0x28)            End        Text       symref 42
 49. ( 1)(0x30) f(void)    End        Text       symref 41
 50. ( 0)(   0) bs6.cxx    End        Text       symref 0
```

9.2.2 Virtual Function Tables and Interludes

Source Listing

```
class Base1 {
  public:
    virtual int virtual_mem_func() { return 1; }
};

class Base2 : virtual public Base1 {
  public:
    virtual int virtual_mem_func() { return 2; }
};

class Base3 : public Base2 {
  public:
    virtual int virtual_mem_func() { return 3; }
};


int foo(Base1 *b1) {
    return  b1->virtual_mem_func();
}

int main() {
```

```
    Base1 *b1;
    Base2 *b2;
    Base3 *b3;

    int i,j,k;

    i = foo(b1);
    j = foo(b2);
    k = foo(b3);
    return 0;
}
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) interlude.cxx
                           File       Text       symref 113
  1. ( 1)(   0) Base1      Tag        Info       [17] Class(extended file 0,
                                                      index 2)
  2. ( 1)( 0x8) Base1      Block      Info       symref 19
  3. ( 2)(   0) __vptr     Member     Info       [20] Pointer to Array
                                                      [(extended file 0, aux
                                                      3)0-1:64] of Virtual func
                                                      table
  4. ( 2)(   0) Base1::Base1(void)
                           Proc       Info       [35] endref 7, Reference
                                                      Class(extended file 0,
                                                      index 2)
  5. ( 3)(   0) this       Param      Info       [32] Const Pointer to
                                                      Class(extended file 0,
                                                      index 2)
  6. ( 2)(   0) Base1::Base1(void)
                           End        Info       symref 4
  7. ( 2)(   0) Base1::Base1(const Base1&)
                           Proc       Info       [45] endref 11, Reference
                                                      Class(extended file 0,
                                                      index 2)
  8. ( 3)(   0) this       Param      Info       [32] Const Pointer to
                                                      Class(extended file 0,
                                                      index 2)
  9. ( 3)(   0)            Param      Info       [42] Reference Const
                                                      Class(extended file 0,
                                                      index 2)
 10. ( 2)(   0) Base1::Base1(const Base1&)
                           End        Info       symref 7
 11. ( 2)(   0) Base1::operator =(const Base1&)
                           Proc       Info       [49] endref 15, Reference
                                                      Class(extended file 0,
                                                      index 2)
 12. ( 3)(   0) this       Param      Info       [32] Const Pointer to
                                                      Class(extended file 0,
                                                      index 2)
 13. ( 3)(   0)            Param      Info       [42] Reference Const
                                                      Class(extended file 0,
                                                      index 2)
 14. ( 2)(   0) Base1::operator =(const Base1&)
                           End        Info       symref 11
 15. ( 2)( 0x1) Base1::virtual_mem_func(void)
                           Proc       Info       [53] endref 18, int
 16. ( 3)(   0) this       Param      Info       [32] Const Pointer to
                                                      Class(extended file 0,
                                                      index 2)
 17. ( 2)(   0) Base1::virtual_mem_func(void)
                           End        Info       symref 15
 18. ( 1)(   0) Base1      End        Info       symref 2
 19. ( 1)(   0) Base2      Tag        Info       [55] Class(extended file 0,
                                                      index 20)
 20. ( 1)(0x18) Base2      Block      Info       symref 42
 21. ( 2)(   0) __vptr     Member     Info       [20] Pointer to Array
                                                      [(extended file 0, aux
                                                      3)0-1:64] of Virtual func
                                                      table
 22. ( 2)(0x40) __bptr     Member     Info       [20] Pointer to Array
                                                      [(extended file 0, aux
                                                      3)0-1:64] of Virtual func
                                                      table
 23. ( 2)(   0) Base1      Virtual Base Class
                                      Info       [17] Class(extended file 0,
                                                      index 2)
 24. ( 2)(   0) Base2::Base2(void)
```

```
                          Proc       Info       [67] endref 28, Reference
                                                     Class(extended file 0,
                                                     index 20)
25. ( 3)(   0) this       Param      Info       [64] Const Pointer to
                                                     Class(extended file 0,
                                                     index 20)
26. ( 3)(   0) <control>  Param      Info       [ 3] int
27. ( 2)(   0) Base2::Base2(void)
                          End        Info       symref 24
28. ( 2)(   0) Base2::Base2(const Base2&)
                          Proc       Info       [77] endref 33, Reference
                                                     Class(extended file 0,
                                                     index 20)
29. ( 3)(   0) this       Param      Info       [64] Const Pointer to
                                                     Class(extended file 0,
                                                     index 20)
30. ( 3)(   0) <control>  Param      Info       [ 3] int
31. ( 3)(   0)            Param      Info       [74] Reference Const
                                                     Class(extended file 0,
                                                     index 20)
32. ( 2)(   0) Base2::Base2(const Base2&)
                          End        Info       symref 28
33. ( 2)(   0) Base2::operator =(const Base2&)
                          Proc       Info       [81] endref 38, Reference
                                                     Class(extended file 0,
                                                     index 20)
34. ( 3)(   0) this       Param      Info       [64] Const Pointer to
                                                     Class(extended file 0,
                                                     index 20)
35. ( 3)(   0) <control>  Param      Info       [ 3] int
36. ( 3)(   0)            Param      Info       [74] Reference Const
                                                     Class(extended file 0,
                                                     index 20)
37. ( 2)(   0) Base2::operator =(const Base2&)
                          End        Info       symref 33
38. ( 2)( 0x1) Base2::virtual_mem_func(void)
                          Proc       Info       [85] endref 41, int
39. ( 3)(   0) this       Param      Info       [64] Const Pointer to
                                                     Class(extended file 0,
                                                     index 20)
40. ( 2)(   0) Base2::virtual_mem_func(void)
                          End        Info       symref 38
41. ( 1)(   0) Base2      End        Info       symref 20
42. ( 1)(   0) Base3      Tag        Info       [87] Class(extended file 0,
                                                     index 43)
43. ( 1)(0x18) Base3      Block      Info       symref 65
44. ( 2)(   0) __vptr     Member     Info       [20] Pointer to Array
                                                     [(extended file 0, aux
                                                     3)0-1:64] of Virtual func
                                                     table
45. ( 2)(0x40) __bptr     Member     Info       [20] Pointer to Array
                                                     [(extended file 0, aux
                                                     3)0-1:64] of Virtual func
                                                     table
46. ( 2)(   0) Base2      Base Class Info       [55] Class(extended file 0,
                                                     index 20)
47. ( 2)(   0) Base3::Base3(void)
                          Proc       Info       [99] endref 51, Reference
                                                     Class(extended file 0,
                                                     index 43)
48. ( 3)(   0) this       Param      Info       [96] Const Pointer to
                                                     Class(extended file 0,
                                                     index 43)
49. ( 3)(   0) <control>  Param      Info       [ 3] int
50. ( 2)(   0) Base3::Base3(void)
                          End        Info       symref 47
51. ( 2)(   0) Base3::Base3(const Base3&)
                          Proc       Info       [109] endref 56, Reference
                                                      Class(extended file 0,
                                                      index 43)
52. ( 3)(   0) this       Param      Info       [96] Const Pointer to
                                                     Class(extended file 0,
                                                     index 43)
53. ( 3)(   0) <control>  Param      Info       [ 3] int
54. ( 3)(   0)            Param      Info       [106] Reference Const
                                                      Class(extended file 0,
                                                      index 43)
55. ( 2)(   0) Base3::Base3(const Base3&)
                          End        Info       symref 51
56. ( 2)(   0) Base3::operator =(const Base3&)
                          Proc       Info       [113] endref 61, Reference
                                                      Class(extended file 0,
```

```
                                                    index 43)
57. ( 3)(   0) this       Param      Info       [96] Const Pointer to
                                                    Class(extended file 0,
                                                    index 43)
58. ( 3)(   0) <control>  Param      Info       [ 3] int
59. ( 3)(   0)            Param      Info       [106] Reference Const
                                                    Class(extended file 0,
                                                    index 43)
60. ( 2)(   0) Base3::operator =(const Base3&)
                          End        Info       symref 56
61. ( 2)( 0x1) Base3::virtual_mem_func(void)
                          Proc       Info       [117] endref 64, int
62. ( 3)(   0) this       Param      Info       [96] Const Pointer to
                                                    Class(extended file 0,
                                                    index 43)
63. ( 2)(   0) Base3::virtual_mem_func(void)
                          End        Info       symref 61
64. ( 1)(   0) Base3      End        Info       symref 43
65. ( 1)(   0) __INTER__Base3_virtual_mem_func_Base1_Base2_Xv
                          Interlude  Info       thunk(extended file 0, index
                                                    61), proc(extended file
                                                    0, index 104)
66. ( 1)(   0) __INTER__Base2_virtual_mem_func_Base1_Xv
                          Interlude  Info       thunk(extended file 0, index
                                                    38), proc(extended file
                                                    0, index 108)
67. ( 1)(0x160) __vtbl_5Base1
                          Static     SData      [126] Const Array [(extended
                                                    file 0, aux 3)0-0:64] of
                                                    Pointer to void
68. ( 1)(0x168) __vtbl_5Base2
                          Static     SData      [126] Const Array [(extended
                                                    file 0, aux 3)0-0:64] of
                                                    Pointer to void
69. ( 1)(0x170) __btbl_5Base2
                          Static     SData      [138] Const Array [(extended
                                                    file 0, aux 3)0-0:64] of
                                                    long
70. ( 1)(0x178) __vtbl_5Base15Base2
                          Static     SData      [126] Const Array [(extended
                                                    file 0, aux 3)0-0:64] of
                                                    Pointer to void
71. ( 1)(0x180) __vtbl_5Base3
                          Static     SData      [126] Const Array [(extended
                                                    file 0, aux 3)0-0:64] of
                                                    Pointer to void
72. ( 1)(0x188) __btbl_5Base3
                          Static     SData      [138] Const Array [(extended
                                                   file 0, aux 3)0-0:64] of long
73. ( 1)(0x190) __vtbl_5Base15Base25Base3
                          Static     SData      [126] Const Array [(extended
                                                    file 0, aux 3)0-0:64] of
                                                    Pointer to void
74. ( 1)(   0) Base1::virtual_mem_func(void)
                          StaticProc Text       [152] endref 79, int
75. ( 2)( 0x1) this       Param      Register   [32] Const Pointer to
                                                    Class(extended file 0,
                                                    index 2)
76. ( 2)( 0x4)            Block      Text       symref 78
77. ( 2)( 0x8)            End        Text       symref 76
78. ( 1)( 0xc) Base1::virtual_mem_func(void)
                          End        Text       symref 74
79. ( 1)(0x14) Base2::virtual_mem_func(void)
                          StaticProc Text       [154] endref 84, int
80. ( 2)( 0x1) this       Param      Register   [64] Const Pointer to
                                                    Class(extended file 0,
                                                    index 20)
81. ( 2)( 0x4)            Block      Text       symref 83
82. ( 2)( 0x8)            End        Text       symref 81
83. ( 1)( 0xc) Base2::virtual_mem_func(void)
                          End        Text       symref 79
84. ( 1)(0x28) Base3::virtual_mem_func(void)
                          StaticProc Text       [156] endref 89, int
85. ( 2)( 0x1) this       Param      Register   [96] Const Pointer to
                                                    Class(extended file 0,
                                                    index 43)
86. ( 2)( 0x4)            Block      Text       symref 88
87. ( 2)( 0x8)            End        Text       symref 86
88. ( 1)( 0xc) Base3::virtual_mem_func(void)
                          End        Text       symref 84
89. ( 1)(0x34) foo(Base1*) Proc      Text       [158] endref 94, int
90. ( 2)( 0x9) b1         Param      Register   [29] Pointer to Class(extended
```

```
                                                    file 0, index 2)
 91. ( 2)(0x10)            Block      Text       symref 93
 92. ( 2)(0x28)            End        Text       symref 91
 93. ( 1)(0x38) foo(Base1*) End       Text       symref 89
 94. ( 1)(0x6c) main       Proc       Text       [160] endref 104, int
 95. ( 2)( 0xc)            Block      Text       symref 103
 96. ( 3)(-8)   b1         Local      Abs        [29] Pointer to Class(extended
                                                    file 0, index 2)
 97. ( 3)(-16)  b2         Local      Abs        [61] Pointer to Class(extended
                                                    file 0, index 20)
 98. ( 3)( 0x9) b3         Local      Register   [93] Pointer to Class(extended
                                                    file 0, index 43)
 99. ( 3)(-24)  i          Local      Abs        [ 3] int
100. ( 3)(-28)  j          Local      Abs        [ 3] int
101. ( 3)(-32)  k          Local      Abs        [ 3] int
102. ( 2)(0x70)            End        Text       symref 95
103. ( 1)(0x80) main       End        Text       symref 94
104. ( 1)(0x20) __INTER__Base3_virtual_mem_func_Base1_Base2_Xv
                           StaticProc Text       [162] endref 108, btNil
105. ( 2)(   0)            Block      Text       symref 107
106. ( 2)(0x28)            End        Text       symref 105
107. ( 1)( 0x8) __INTER__Base3_virtual_mem_func_Base1_Base2_Xv
                           End        Text       symref 104
108. ( 1)( 0xc) __INTER__Base2_virtual_mem_func_Base1_Xv
                           StaticProc Text       [164] endref 112, btNil
109. ( 2)(   0)            Block      Text       symref 111
110. ( 2)(0x14)            End        Text       symref 109
111. ( 1)( 0x8) __INTER__Base2_virtual_mem_func_Base1_Xv
                           End        Text       symref 108
112. ( 0)(   0) interlude.cxx
                           End        Text       symref 0
```

9.2.3 Namespace Definitions and Uses

See Section 5.3.6.4 for related information.

Source Listing

```
ns1.h:

namespace ns1 {
    class Cobj {};
    extern int i1;
}

ns2.h:

namespace ns1 {
    int x1(void);
}

ns.C:

#include "ns1.h"
#include "ns2.h"

namespace ns1 {
    extern int part3;
}

int ns1::i1 = 1000;
int ns1::part3 = 3;
int ns1::x1(void) {
    using namespace ns1;
    return i1*10;
}
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) ns.C          File      Text    symref 7
  1. ( 1)(   0) ns1::x1(void) Proc      Text    [4] endref 6, int
  2. ( 2)(   0)               Using     Info    [6] symref(file 1, index 1)
  3. ( 2)( 0x8)               Block     Text    symref 5
  4. ( 2)(0x14)               End       Text    symref 3
  5. ( 1)(0x18) ns1::x1(void) End       Text    symref 1
  6. ( 0)(   0) ns.C          End       Text    symref 0

File 1 Local Symbols:
```

```
  0. ( 0)(   0) ns1.h          File      Text    symref 8
  1. ( 1)(   0) ns1            Namespace Info    symref 7
  2. ( 2)(   0) ns1::x1(void) Proc      Info    [2] endref 4, int
  3. ( 2)(   0) ns1::x1(void) End       Info    symref 2
  4. ( 2)(   0) i1             Member    Info    [4] int
  5. ( 2)(   0) part3          Member    Info    [4] int
  6. ( 1)(   0) ns1            End       Info    symref 1
  7. ( 0)(   0) ns1.h          End       Text    symref 0

Externals Table:

0. (file 0)(0x50) ns1::i1       Global    SData   [3] int
1. (file 0)(0x58) ns1::part3    Global    Sdata   [3] int
2. (file 0)(   0) ns1::x1(void) Proc      Text    symref 1
```

9.2.4 Unnamed Namespaces

See Section 5.3.6.4.3 for related information.

Source Listing

uns.C:

```
namespace {
    int usv1;
    int usv2;
}

int privat(void) {
    return usv1 + usv2;
}
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) uns.C                      File      Info  symref 13
  1. ( 1)(   0) __N1AqSbNU3PTf             Namespace Info  symref 5
  2. ( 2)(   0) <unnamed namespace>::usv1 Member    Info  [3] int
  3. ( 2)(   0) <unnamed namespace>::usv2 Member    Info  [3] int
  4. ( 1)(   0) __N1AqSbNU3PTf             End       Info  symref 1
  5. ( 1)(   0)                            Using     Info  [4] symref(file 0, index 1)
  6. ( 1)(0x50) <unnamed namespace>::usv1 Static    SBss  [3] int
  7. ( 1)(0x54) <unnamed namespace>::usv2 Static    SBss  [3] int
  8. ( 1)(   0) privat(void)               Proc      Text  [5] endref 12, int
  9. ( 2)( 0x8)                           Block     Text  symref 11
 10. ( 2)(0x1c)                           End       Text  symref 9
 11. ( 1)(0x20)                           End       Text  symref 8
 12. ( 0)(   0)                           End       Text  symref 0
```

9.2.5 Namespace Aliases

See Section 5.3.6.4.2 for related information.

Source Listing

alias.C:

```
namespace long_namespace_name {
    extern int nmem;
}

int get_nmem(void) {
    namespace nknm = long_namespace_name;
    namespace nknm2 = nknm;
    return nknm::nmem;
}
```

Symbol Table Contents

```
File 0 Local Symbols

  0. ( 0)(   0) alias.C             File      Text   symref 11
  1. ( 1)(   0) long_namespace_name Namespace Info   symref 4
  2. ( 2)(   0) nmem                Member    Info   [3] int
  3. ( 1)(   0) long_namespace_name End       Info   symref 1
```

```
  4. ( 1)(   0) get_nmem(void)      Proc      Text   [4] endref 10, int
  5. ( 2)( 0x8)                     Block     Text   symref 9
  6  ( 2)(   0) nknm                Alias     Info   [5] symref(file 0,index 1)
  7  ( 2)(   0) nknm2               Alias     Info   [6] symref(file 0,index 6)
  8. ( 2)(0x10)                     End       Text   symref 5
  9. ( 1)(0x14) get_nmem(void)      End       Text   symref 4
 10. ( 0)(   0) alias.C             End       Text   symref 0

Externals Table

0. (file 0)(0x4) long_namespace_name::nmem Global   Undefined  [3]int
1. (file 0)(  0) get_nmem(void)            Proc     Text       symref 4
```

9.2.6 Exception-Handling

See Section 3.3.8 for related information.

Source Listing

```
#include <iostream.h>

class Vector {
        int *p;
        int sz;

  public:
        enum { max=1000 };

        Vector(int);

        class Range { };
        class Size { };

        int operator[](int i);

}; // Vector

Vector::Vector(int i) {
        if (i>max) throw Size();
        p=new int[i];
        if (p) sz=i;
        else sz=0;
}

int Vector::operator[](int i) {
        if (0<=i && i<sz) return p[i];
        throw Range();
}

void f() {
        int i;

        try {
                cout<<"size?";
                cin>>i;
                Vector v(i);
                cout<<v[i]<<"\n";
        }

        catch (Vector::Range) {
                cout<< "bad news; outta here...\n";
        }

        catch (Vector::Size) {
                cout<< "can't initialize to that size...\n";
        }

} // f

main() {

        f();
}
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) multiexc.cxx File     Text       symref 83
  1. ( 1)(   0) Vector     Tag        Info       [16] Class(extended file 0,
```

```
                                                                index 2)
 2. ( 1)(0x10) Vector     Block      Info       symref 40
 3. ( 2)(   0) <generated_name_0005>
                          Tag        Info       [19] enum(extended file 0,
                                                     index 4)
 4. ( 2)(   0) <generated_name_0005>
                          Block      Info       symref 7
 5. ( 3)(0x3e8) max       Member     Info       [ 2] btNil
 6. ( 2)(   0) <generated_name_0005> End        Info       symref 4
 7. ( 2)(   0) Range      Tag        Info       [22] Class(extended file 0,
                                                     index 8)
 8. ( 2)( 0x1) Range      Block      Info       symref 14
 9. ( 3)(   0) Vector::Range::operator =(const Vector::Range&)
                          Proc       Info       [40] endref 13, Reference
                                                     Class(extended file 0,
                                                     index 8)
10. ( 4)(   0) this       Param      Info       [31] Const Pointer to
                                                     Class(extended file 0,
                                                     index 8)
11. ( 4)(   0)            Param      Info       [37] Reference Const
                                                     Class(extended file 0,
                                                     index 8)
12. ( 3)(   0) Vector::Range::operator =(const Vector::Range&)
                          End        Info       symref 9
13. ( 2)(   0) Range      End        Info       symref 8
14. ( 2)(   0) Size       Tag        Info       [44] Class(extended file 0,
                                                     index 15)
15. ( 2)( 0x1) Size       Block      Info       symref 21
16. ( 3)(   0) Vector::Size::operator =(const Vector::Size&)
                          Proc       Info       [62] endref 20, Reference
                                                     Class(extended file 0,
                                                     index 15)
17. ( 4)(   0) this       Param      Info       [53] Const Pointer to
                                                     Class(extended file 0,
                                                     index 15)
18. ( 4)(   0)            Param      Info       [59] Reference Const
                                                     Class(extended file 0,
                                                     index 15)
19. ( 3)(   0) Vector::Size::operator =(const Vector::Size&)
                          End        Info       symref 16
20. ( 2)(   0) Size       End        Info       symref 15
21. ( 2)(   0) p          Member     Info       [66] Pointer to int
22. ( 2)(0x40) sz         Member     Info       [ 3] int
23. ( 2)(   0) Vector::Vector(int)
                          Proc       Info       [76] endref 27, Reference
                                                     Class(extended file 0,
                                                     index 2)
24. ( 3)(   0) this       Param      Info       [73] Const Pointer to
                                                     Class(extended file 0,
                                                     index 2)
25. ( 3)(   0) i          Param      Info       [ 3] int
26. ( 2)(   0) Vector::Vector(int)
                 End      Info       symref 23
27. ( 2)(   0) Vector::Vector(const Vector&)
                 Proc     Info       [86] endref 31, Reference Class(extended
                                                     file 0, index 2)
28. ( 3)(   0) this       Param      Info       [73] Const Pointer to
                                                     Class(extended file 0,
                                                     index 2)
29. ( 3)(   0)            Param      Info       [83] Reference Const
                                                     Class(extended file 0,
                                                     index 2)
30. ( 2)(   0) Vector::Vector(const Vector&)
                          End        Info       symref 27
31. ( 2)(   0) Vector::operator [](int)
                          Proc       Info       [90] endref 35, int
32. ( 3)(   0) this       Param      Info       [73] Const Pointer to
                                                     Class(extended file 0,
                                                     index 2)
33. ( 3)(   0) i          Param      Info       [ 3] int
34. ( 2)(   0) Vector::operator [](int)
                          End        Info       symref 31
35. ( 2)(   0) Vector::operator =(const Vector&)
                          Proc       Info       [92] endref 39, Reference
                                                     Class(extended file 0,
                                                     index 2)
36. ( 3)(   0) this       Param      Info       [73] Const Pointer to
                                                     Class(extended file 0,
                                                     index 2)
37. ( 3)(   0)            Param      Info       [83] Reference Const
                                                     Class(extended file 0,
                                                     index 2)
```

```
38. ( 2)(   0) Vector::operator =(const Vector&)
                          End         Info        symref 35
39. ( 1)(   0) Vector     End         Info        symref 2
40. ( 1)(   0) __throw_Q16Vector4Size
                          Tag         Info        [96] struct(extended file 0,
                                                       index 41)
41. ( 1)(0x10) __throw_Q16Vector4Size
                          Block       Info        symref 45
42. ( 2)(   0) type_signature
                          Member      Info        [99] Pointer to char
43. ( 2)(0x40) thunk      Member      Info        [99] Pointer to char
44. ( 1)(   0) __throw_Q16Vector4Size
                          End         Info        symref 41
45. ( 1)(0x3c0) __throw_Q16Vector4Size
                          Static      Data        [176] Array [(extended file 7,
                                                        aux 9)0-1:128] of
                                                        struct(extended file 0,
                                                        index 41)
46. ( 1)(0x3a0) __throw_Q16Vector5Range
                          Static      Data        [176] Array [(extended file 7,
                                                        aux 9)0-1:128] of
                                                        struct(extended file 0,
                                                        index 41)
47. ( 1)(   0) Vector::Vector(int)
                          Proc        Text        [184] endref 57, Reference
                                                        Class(extended file 0,
                                                        index 2)
48. ( 2)( 0xa) this       Param       Register    [73] Const Pointer to
                                                       Class(extended file 0,
                                                       index 2)
49. ( 2)( 0x9) i          Param       Register    [ 3] int
50. ( 2)(0x20)            Block       Text        symref 56
51. ( 3)(-8)   __t8       Local       Abs         [44] Class(extended file 0,
                                                       index 15)
52. ( 3)(0x3c0) __throw_Q16Vector4Size
                          Static      Data        indexNil
53. ( 3)(-16)  __t9       Local       Abs         [10] unsigned long
54. ( 3)(-24)  __t10      Local       Abs         [194] Pointer to Array
                                                        [(extended file 7, aux
                                                        9)0-0:32] of int
55. ( 2)(0x74)            End         Text        symref 50
56. ( 1)(0xb4) Vector::Vector(int)
                          End         Text        symref 47
57. ( 1)(0xb4) Vector::operator [](int)
                          Proc        Text        [200] endref 65, int
58. ( 2)(0x28) this       Param       Abs         [73] Const Pointer to
                                                       Class(extended file 0,
                                                       index 2)
59. ( 2)( 0x9) i          Param       Register    [ 3] int
60. ( 2)(0x1c)            Block       Text        symref 64
61. ( 3)(-16)  __t11      Local       Abs         [22] Class(extended file 0,
                                                       index 8)
62. ( 3)(0x3a0) __throw_Q16Vector5Range
                          Static      Data        indexNil
63. ( 2)(0x44)            End         Text        symref 60
64. ( 1)(0x7c) Vector::operator [](int)
                          End         Text        symref 57
65. ( 1)(0x130) f(void)   Proc        Text        [202] endref 78, void
66. ( 2)(0x1c)            Block       Text        symref 77
67. ( 3)(-32)  i          Local       Abs         [ 3] int
68. ( 3)(-48) __current_try_block_decl
                          Local       Abs         indexNil
69. ( 3)(0x28)            Block       Text        symref 72
70. ( 4)(-24)  v          Local       Abs         [16] Class(extended file 0,
                                                       index 2)
71. ( 3)(0xab)            End         Text        symref 69
72. ( 3)(0xac)            Block       Text        symref 74
73. ( 3)(0xe3)            End         Text        symref 72
74. ( 3)(0xe4)            Block       Text        symref 76
75. ( 3)(0x113)           End         Text        symref 74
76. ( 2)(0x11c)           End         Text        symref 66
77. ( 1)(0x130) f(void)   End         Text        symref 65
78. ( 1)(0x260) main      Proc        Text        [204] endref 82, int
79. ( 2)(0x10)            Block       Text        symref 81
80. ( 2)(0x18)            End         Text        symref 79
81. ( 1)(0x24) main       End         Text        symref 78
82. ( 0)(   0) multiexc.cxx End       Text        symref 0
```

9.3 Fortran

9.3.1 Common Data

See Section 5.3.6.6 for related information.

Source Listing

```
comm.f:

C  main program
        INTEGER IND, CLASS(10)
        REAL MARKS(50)
        COMMON CLASS,MARKS,IND
        CALL EVAL(5)
        STOP
        END
C
        SUBROUTINE EVAL(PERF)
        INTEGER PERF,JOB(10),PAR
        REAL GRADES(50)
        COMMON JOB,GRADES,PAR
        RETURN
        END
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) comm.f     File       Text       symref 13
  1. ( 1)(   0) comm$main_ Proc       Text       [25] endref 6, btNil
  2. ( 2)(0x10)            Block      Text       symref 5
  3. ( 3)(   0) _BLNK__    Static     Common     [39] struct(extended file 1,
                                                      index 1)
  4. ( 2)(0x44)            End        Text       symref 2
  5. ( 1)(0x44) comm$main_  End        Text       symref 1
  6. ( 1)(0x44) eval_      Proc       Text       [42] endref 12, btNil
  7. ( 2)(   0) PERF       Param      VarRegister [11] 32-bit long
  8. ( 2)( 0x4)            Block      Text       symref 11
  9. ( 3)(   0) _BLNK__    Static     Common     [56] struct(extended file 2,
                                                      index 1)
 10. ( 2)( 0x4)            End        Text       symref 8
 11. ( 1)( 0x8) eval_      End        Text       symref 6
 12. ( 0)(   0) comm.f     End        Text       symref 0

File 1 Local Symbols:

  0. ( 0)(   0) _BLNK__    File       Text       symref 7
  1. ( 1)(0xf4) _BLNK__    Block      Common     symref 6
  2. ( 2)(0x780) IND       Member     Info       [ 5] 32-bit long
  3. ( 2)(   0) CLASS      Member     Info       [ 6] Array [(extended file 0,
                                                      aux 11)1-10:4] of 32-bit
                                                      long
  4. ( 2)(0x140) MARKS     Member     Info       [12] Array [(extended file 0,
                                                      aux 11)1-50:4] of float
  5. ( 1)(   0)            End        Common     symref 1
  6. ( 0)(   0) _BLNK__    End        Text       symref 0

File 2 Local Symbols:

  0. ( 0)(   0) _BLNK__    File       Text       symref 7
  1. ( 1)(0xf4) _BLNK__    Block      Common     symref 6
  2. ( 2)(   0) JOB        Member     Info       [ 5] Array [(extended file 0,
                                                      aux 11)1-10:4] of 32-bit
                                                      long
  3. ( 2)(0x780) PAR        Member     Info       [11] 32-bit long
  4. ( 2)(0x140) GRADES     Member     Info       [12] Array [(extended file 0,
                                                       aux 11)1-50:4] of float
  5. ( 1)(   0)            End        Common     symref 1
  6. ( 0)(   0) _BLNK__    End        Text       symref 0

Externals table:
0. (file  0) (   0) MAIN__     Proc       Text       symref 1
1. (file  0) (0xf4) _BLNK__    Global     Common     indexNil
2. (file  0) (   0) comm$main_ Proc       Text       symref 1
3. (file  0) (0x44) eval_      Proc       Text       symref 6
4. (file  0) (   0) for_stop   Proc       Undefined  indexNil
5. (file  0) (   0) for_set_reentrancy
                               Proc       Undefined  indexNil
6. (file  0) (   0) _fpdata    Global     Undefined  indexNil
```

```
                        ***FILE DESCRIPTOR TABLE***

  filename              address            vstamp -g sex      lang flags
       cbLine  ---------------iBase/count----------------------------------
     lnOffset     sym      line      pd    string     opt     aux   rfd
comm.o:
  comm.f                0x0000000000000000 0x0000  0  el   Fortran readin
            0       0         0       0         0       0       0     0
            5      13        20       2        44       0      59     0
  _BLNK__               0x0000000000000000 0x0000  0  el   Fortran merge
            0      13         0       2        44       0      59     0
            0       7         0       0        33       0      18     0
  _BLNK__               0x0000000000000000 0x0000  0  el   Fortran merge
            0      20         0       2        77       0      77     0
            0       7         0       0        32       0      18     0
```

9.3.2 Alternate Entry Points

See Section 5.3.6.7 for related information.

Source Listing

```
aent.f:

      program entryp

      print *, "In entryp, the main routine"
      call anentry()
      call anentry1(2,3)
      call anentry1a(2,3,4,5,6,7)
      call asubr()
      print *, "exiting..."

      end

      subroutine asubr
      real*4 areal /1.2345E-6/
      print *, "In asubr"
      return

      entry anentry
      print *, "In anentry"
      return

      entry anentry1(a,b,c,d,e,f)
      a = 1
      b = 2
      print *, "In anentry1"
      return

      include 'entrya.h'

      entry anentry2(b,a)
      print *, "In anentry2"
                return

      entry anentry3
      include 'entryb.h'
               return

      end
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) aent.f     File       Text       symref 30
  1. ( 1)(   0) entryp_    Proc       Text       [ 4] endref 5, btNil
  2. ( 2)(0x14)            Block      Text       symref 4
  3. ( 2)(0xf8)            End        Text       symref 2
  4. ( 1)(0x108) entryp_   End        Text       symref 1
  5. ( 1)(0x108) asubr_    Proc       Text       [ 6] endref 29, btNil
  6. ( 2)(0x20)            Block      Text       symref 28
  7. ( 3)(0x610) AREAL     Static     Data       [ 8] float
  8. ( 3)(0x17c) anentry_  Proc       Text       [ 9] endref -1, btNil
  9. ( 4)(0x1f0) anentry1_ Proc       Text       [11] endref -1, btNil
 10. ( 5)( 0xa) A          Param      VarRegister [ 8] float
 11. ( 5)( 0x9) B          Param      VarRegister [ 8] float
```

```
 12. ( 5)(-144) C          Param      Var        [ 8] float
 13. ( 5)(-152) D          Param      Var        [ 8] float
 14. ( 5)(-160) E          Param      Var        [ 8] float
 15. ( 5)(-168) F          Param      Var        [ 8] float
 16. ( 5)(0x290) anentry1a_ Proc      Text       [13] endref -1, btNil
 17. ( 6)( 0xa) A          Param      VarRegister [ 8] float
 18. ( 6)( 0x9) B          Param      VarRegister [ 8] float
 19. ( 6)(-144) C          Param      Var        [ 8] float
 20. ( 6)(-152) D          Param      Var        [ 8] float
 21. ( 6)(-160) E          Param      Var        [ 8] float
 22. ( 6)(-168) F          Param      Var        [ 8] float
 23. ( 6)(0x330) anentry2_ Proc       Text       [15] endref -1, btNil
 24. ( 7)( 0x9) B          Param      VarRegister [ 8] float
 25. ( 7)( 0xa) A          Param      VarRegister [ 8] float
 26. ( 7)(0x3ac) anentry3_ Proc       Text       [17] endref -1, btNil
 27. ( 7)(0x384)           End        Text       symref 6
 28. ( 6)(0x3a0) asubr_    End        Text       symref 5
 29. ( 5)(   0) aent.f     End        Text       symref 0

Externals table:
0. (file 0) (   0) MAIN__       Proc       Text       symref 1
1. (file 0) (   0) entryp_      Proc       Text       symref 1
2. (file 0) (0x108) asubr_      Proc       Text       symref 5
3. (file 0) (0x290) anentry1a_ Proc       Text       symref 16
4. (file 0) (0x1f0) anentry1_  Proc       Text       symref 9
5. (file 0) (0x17c) anentry_   Proc       Text       symref 8
6. (file 0) (   0) for_set_reentrancy
                               Proc       Undefined  indexNil
7. (file 0) (   0) for_write_seq_lis
                               Proc       Undefined  indexNil
8. (file 0) (0x330) anentry2_  Proc       Text       symref 23
9. (file 0) (0x3ac) anentry3_  Proc       Text       symref 26
10.(file 0) (   0) _fpdata     Global     Undefined  indexNil


                        ***PROCEDURE DESCRIPTOR TABLE***

    name      prof rfrm  isym iline  iopt   regmask  regoff  fpoff fp
   address    guse gpro lnOff lnLow lnHigh fregmask  frgoff lcloff pc

aent.o:
  aent.f           [0 for 7]
    entryp_     0   0      1     0    -1  0x04000200   -112    112 30
      0x000     1   8      0     1    10  0x00000000      0      0 26
    asubr_      0   0      5    66    -1  0x04001e00   -256    256 30
      0x108     1   8      8    12    37  0x00000000      0      0 26
    anentry_    0   0      8    95    -1  0x04001e00   -256    256 30
      0x17c     1   8     11    17    -1  0x00000000      0      0 26
    anentry1_   0   0      9   124    -1  0x04001e00   -256    256 30
      0x1f0     1   8     14    21    -1  0x00000000      0      0 26
    anentry1a_  0   0     16   164    -1  0x04001e00   -256    256 30
      0x290     1   8     20     1    -1  0x00000000      0      0 26
    anentry2_   0   0     23   204    -1  0x04001e00   -256    256 30
      0x330     1   8     25    29    -1  0x00000000      0      0 26
    anentry3_   0   0     26   235    -1  0x04001e00   -256    256 30
      0x3ac     1   8     28    33    -1  0x00000000      0      0 26
```

9.3.3 Array Descriptors

See Section 5.3.8.9 for related information.

Source Listing

arraydescs.f:

```
! -*- Fortran -*-

        integer, allocatable, dimension(:,:) :: alloc_int_2d
        real, pointer, dimension(:) :: pointer_real_1d

        allocate(alloc_int_2d(10,20))

        call zowie(alloc_int_2d)

        end

contains

        subroutine zowie(assumed_int_2d)
         integer, dimension(:,:) :: assumed_int_2d
```

```
        print *, assumed_int_2d
        return
       end subroutine
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) arraydescs.f File     Text       symref 43
  1. ( 1)(   0) main$arraydescs_
                           Proc       Text       [ 4] endref 26, btNil
  2. ( 2)(0x40) $f90$f90_array_desc
                           Block      Info       symref 10
  3. ( 3)(   0) dim        Member     Info       [ 6] 8-bit int
  4. ( 3)(0x40) element_length Member     Info       [ 7] 32-bit long
  5. ( 3)(0x80) ptr        Member     Info       [ 9] Pointer to float
  6. ( 3)(0x140) ies1      Member     Info       [10] 32-bit long
  7. ( 3)(0x180) ub1       Member     Info       [11] 32-bit long
  8. ( 3)(0x1c0) lb1       Member     Info       [12] 32-bit long
  9. ( 2)(   0) $f90$f90_array_desc
                           End        Info       symref 2
 10. ( 2)(0x58) $f90$f90_array_desc
                           Block      Info       symref 21
 11. ( 3)(   0) dim        Member     Info       [16] 8-bit int
 12. ( 3)(0x40) element_length
                           Member     Info       [17] 32-bit long
 13. ( 3)(0x80) ptr        Member     Info       [19] Pointer to 32-bit long
 14. ( 3)(0x140) ies1      Member     Info       [20] 32-bit long
 15. ( 3)(0x180) ub1       Member     Info       [21] 32-bit long
 16. ( 3)(0x1c0) lb1       Member     Info       [22] 32-bit long
 17. ( 3)(0x200) ies2      Member     Info       [23] 32-bit long
 18. ( 3)(0x240) ub2       Member     Info       [24] 32-bit long
 19. ( 3)(0x280) lb2       Member     Info       [25] 32-bit long
 20. ( 2)(   0) $f90$f90_array_desc
                           End        Info       symref 10
 21. ( 2)(0x14)            Block      Text       symref 25
 22. ( 3)(0x450) POINTER_REAL_1D
                           Static     Bss        [13] struct(extended file 0,
                                                      index 2)
 23. ( 3)(0x3c0) ALLOC_INT_2D
                           Static     Data       [26] struct(extended file 0,
                                                      index 10)
 24. ( 2)(0x160)            End        Text       symref 21
 25. ( 1)(0x170) main$arraydescs_
                            End        Text       symref 1
 26. ( 1)(0x170) zowie_     Proc       Text       [29] endref 42, btNil
 27. ( 2)(0x58) $f90$f90_array_desc
                            Block      Info       symref 38
 28. ( 3)(   0) dim         Member     Info       [31] 8-bit int
 29. ( 3)(0x40) element_length
                            Member     Info       [32] 32-bit long
 30. ( 3)(0x80) ptr         Member     Info       [34] Pointer to 32-bit long
 31. ( 3)(0x140) ies1       Member     Info       [35] 32-bit long
 32. ( 3)(0x180) ub1        Member     Info       [36] 32-bit long
 33. ( 3)(0x1c0) lb1        Member     Info       [37] 32-bit long
 34. ( 3)(0x200) ies2       Member     Info       [38] 32-bit long
 35. ( 3)(0x240) ub2        Member     Info       [39] 32-bit long
 36. ( 3)(0x280) lb2        Member     Info       [40] 32-bit long
 37. ( 2)(   0) $f90$f90_array_desc
                            End        Info       symref 27
 38. ( 2)( 0x9) ASSUMED_INT_2D
                            Param      VarRegister [41] struct(extended file 0,
                                                        index 27)
 39. ( 2)(0x34)             Block      Text       symref 41
 40. ( 2)(0x1f4)            End        Text       symref 39
 41. ( 1)(0x220) zowie_     End        Text       symref 26
 42. ( 0)(   0) arraydescs.f End       Text       symref 0
```

9.4 Pascal

9.4.1 Sets

See Section 5.3.8.13 for related information.

Source Listing

```
program sets(input,output);

type digitset=set of 0..9;
```

```
var odds,evens:digitset;

begin

  odds:=[1,3,5,7,9];
  evens:=[0,2,4,6,8];

end.
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) set.p      File       Text       symref 10
  1. ( 1)(0x50) $dat       Static     SBss       indexNil
  2. ( 1)(   0) main       Proc       Text       [ 8] endref 9, btNil
  3. ( 2)( 0x4)            Block      Text       symref 8
  4. ( 3)(   0) digitset   Typdef     Info       [16] set of(extended file 0,
                                                      index 10)
  5. ( 3)(-8)   odds       Local      Abs        [16] set of(extended file 0,
                                                      index 10)
  6. ( 3)(-16)  evens      Local      Abs        [16] set of(extended file 0,
                                                      index 10)
  7. ( 2)(0x1c)            End        Text       symref 3
  8. ( 1)(0x24) main       End        Text       symref 2
  9. ( 0)(   0) set.p      End        Text       symref 0
```

9.4.2 Subranges

See Section 5.3.8.12 for related information.

Source Listing

```
subrange.p:

program years(input,output);

type century=0..99;

var year:century;

begin

readln(year);

end.
```

Symbol Table Contents

```
File 0 Local Symbols:

  0. ( 0)(   0) subrange.p File       Text       symref 9
  1. ( 1)(0xc0) $dat       Static     SBss       indexNil
  2. ( 1)(   0) main       Proc       Text       [ 8] endref 8, btNil
  3. ( 2)(0x10)            Block      Text       symref 7
  4. ( 3)(   0) century    Typdef     Info       [10] range0..99 of(extended
                                                      file 0, index 2): 8
  5. ( 3)(-8)   year       Local      Abs        [10] range0..99 of(extended
                                                      file 0, index 2): 8
  6. ( 2)(0x68)            End        Text       symref 3
  7. ( 1)(0x74) main       End        Text       symref 2
  8. ( 0)(   0) subrange.p End        Text       symref 0
```

9.4.3 Variant Records

See Section 5.3.8.11 for related information.

Source Listing

```
variant.p:

program variant(input,output);

type    employeetype=(h,s,m);
        employeerecord=record
                id:integer;
                case status: employeetype of
```

```
                        h: (rate:real;
                            hours:integer;);
                        s: (salary:real);
                        m: (profit:real);
        end; { record }

var     employees:array[1..100] of employeerecord;

begin

        employees[1].id:=1;
        employees[1].profit:=0.06;

end.
```

Symbol Table Contents

```
File 0 Local Symbols

 0. (0)(   0) variant.p  File       Text       symref 28
 1. (1)(   0) VARIANT    StaticProc Text       [2] endref 27, btNil
 2. (2)(   0) EMPLOYEETYPE
                         Block      Info       symref 7
 3. (3)(   0) H          Member     Info       [0] btNil
 4. (3)( 0x1) S          Member     Info       [0] btNil
 5. (3)( 0x2) M          Member     Info       [0] btNil
 6. (2)(   0) EMPLOYEETYPE
                         End        Info       symref 2
 7  (2)(0x10) EMPLOYEERECORD
                         Block      Info       symref 23
 8  (3)(   0) ID         Member     Info       [1] int
 9  (3)(0x20) STATUS     Member     Info       [5] enum(extended file 1, index
                                                 2)
10. (3)( 0x9)            Block      Variant    symref 22
11. (4)( 0xc)            Block      Info       symref 15
12. (5)(0x40) RATE       Member     Info       [11] float
13. (5)(0x60) HOURS      Member     Info       [1] int
14  (4)(   0)            End        Info       symref 11
15. (4)(0x11)            Block      Info       symref 18
16. (5)(0x40) SALARY     Member     Info       [11] float
17. (4)(   0)            End        Info       symref 15
18. (4)(0x16)            Block      Info       symref 21
19. (5)(0x40) PROFIT     Member     Info       [11] float
20. (4)(   0)            End        Info       symref 18
21. (3)( 0x9)            End        Variant    symref 10
22. (2)(   0) EMPLOYEERECORD
                         End        Info       symref 7
23. (2)(0x18)            Block      Text       symref 26
24. (3)(-1600) EMPLOYEES Local      Abs        [32] Array [(extended file 1,
                                                 aux 27)1-100:128] of struct
                                                 (extended file 1, index 7)
25. (2)(0x30)            End        Text       symref 23
26. (1)(0x40) VARIANT    End        Text       symref 1
27. (0)(   0) variant.p  End        Text       symref 0
```

10

Programming Examples

This chapter provides complete examples of programs that access object file and symbol table structures. These examples are meant to reinforce the descriptions of these structures and their use. In many cases APIs exist that could be used to simplify these examples. Use of these APIs is strongly encouraged, but they are not employed in these programming examples, because they would hide the details of the structure access and data interpretation.

10.1 Packed Line Numbers

This example illustrates the use of structures described in Section 5.3.2.2.1. The following program will read packed line numbers and display them in expanded form.

Source Listing

readline.c:

```
/* Expand packed line numbers and display ranges of addresses
 * and line numbers.  For simplicity, file and procedure names are
 * omitted.
 */

#include <filehdr.h>
#include <scnhdr.h>
#include <sym.h>
#include <stdio.h>

main(int argc, char **argv){
    FILE                *fd;    /* fopen handle */
    FILHDR              fhead;  /* object file header */
    HDRR                hdrr;   /* symbol table header */
    unsigned char       *pline; /* buffer for packed lines */
    FDR                 *fdr;   /* buffer for FDRs */
    PDR                 *pdr;   /* buffer for PDRs */

    if (argc < 2) {
        printf("Usage: readline <OBJECT>\n");
        exit(1);
    }

    /* Open file argument */

    if ((fd = fopen(argv[1], "r")) == (FILE *)NULL) {
        printf("Bad file %s!\n", argv[1]);
        exit(1);
    }

    /* Read file header and test magic id */

    if (fread(&fhead, FILHSZ, 1, fd) != 1) {
        printf("fread filheader!\n");
        exit(1);
    } else if (fhead.f_magic != ALPHAMAGIC) {
        if (fhead.f_magic == ALPHAUMAGIC)
            printf("Compressed object not supported\n");
        else
            printf("%s is not an object file\n", argv[1]);
        exit(1);
    }

    /* Read symbolic header */

    if (fhead.f_symptr == 0) {
        printf("no syms!\n");
        exit(1);
    }
```

```
    fseek(fd, fhead.f_symptr, 0);
    if (fread(&hdrr, sizeof(HDRR), 1, fd) != 1) {
        printf("symheader read failed!\n");
        exit(1);
    }

    /* Test for FDRs, PDRs, and packed line numbers */

    if (!hdrr.ifdMax) {
        printf("No file descriptors!\n");
        exit(1);
    } else if (!hdrr.ipdMax) {
        printf("No procedure descriptors!\n");
        exit(1);
    } else if (hdrr.cbLine == 0) {
        printf("No lines!\n");
        exit(1);
    }

    /* Read FDRs */

    fseek(fd, hdrr.cbFdOffset, 0);
    if (!(fdr = (FDR *)malloc(hdrr.ifdMax * sizeof(FDR)))) {
        printf("FDR malloc failed\n");
        exit(1);
    }
    if (fread(fdr, sizeof(FDR), hdrr.ifdMax, fd) != hdrr.ifdMax) {
        printf("FDR read failed\n");
        exit(1);
    }

    /* Read PDRs */

    fseek(fd, hdrr.cbPdOffset, 0);
    if (!(pdr = (PDR *)malloc(hdrr.ipdMax * sizeof(PDR)))) {
        printf("PDR malloc failed\n");
        exit(1);
    }
    if (fread(pdr, sizeof(PDR), hdrr.ipdMax, fd) != hdrr.ipdMax) {
        printf("PDR read failed\n");
        exit(1);
    }

    /* Read packed lines */

    fseek(fd, hdrr.cbLineOffset, 0);
    if (!(pline = (unsigned char *)malloc(hdrr.cbLine))) {
        printf("pline malloc failed\n");
        exit(1);
    }
    if (fread(pline, 1, hdrr.cbLine, fd) != hdrr.cbLine) {
        printf("pline read failed\n");
        exit(1);
    }

    /* Dump expanded packed lines */

    expand_lines(fdr, hdrr.ifdMax, pdr, pline);
}

expand_lines(FDR *fdr, int ifdmax,    /* FDRs and count */
             PDR *pdr,                /* PDRs */
             unsigned char *pline) {  /* Packed lines */

    int ifd;

    /* Iterate through FDRs */

    for (ifd = 0; ifd < ifdmax; ifd++) {

        /* Ignore FDRs without line numbers */

        if (fdr[ifd].cbLine == 0)
            continue;

        printf("File %d:\n", ifd);

        /* Dump expanded lines for this FDR */

        expand_file_lines(&fdr[ifd],
                          &pdr[fdr[ifd].ipdFirst],
                          fdr[ifd].cpd,
```

```
                        &pline[fdr[ifd].cbLineOffset],
                        fdr[ifd].cbLine);
    }
}


proc_pline_count(FDR *fdr,              /* FDR */
                 PDR *pdr,              /* First PDR for FDR */
                 int ipd) {             /* Index of current PDR */

    int nextipd;   /* Index of next PDR with line numbers */
    int i;         /* Index to iterate through PDRs */

    /* Return the number of packed line entries for a PDR.
     * To simplify processing, a procedure with alternate
     * entries is treated as a set of contiguous procedures.
     * In this program the calling procedure does not need
     * to know that the packed lines associated with the
     * alternate entry actually belong to the containing
     * procedure.
     */


    /* Test for no lines */

    if (pdr[ipd].iline == ilineNil)
        return(0);

    nextipd = -1;       /* Next PDR not found yet. */

    /* Iterate through all PDRs for this FDR */

    for (i=0; i < fdr->cpd; i++) {

        /* Find PDRs with packed line offsets the same or
         * greater than the current PDR's.
         */

        if (i != ipd &&
            pdr[i].iline != ilineNil &&
            pdr[i].cbLineOffset >= pdr[ipd].cbLineOffset) {

            /* Save PDR index of closest offset found so far.
             * Do not assume the PDRs are arranged with
             * ascending packed line offsets.
             */

            if (nextipd == -1 ||
                pdr[i].cbLineOffset < pdr[nextipd].cbLineOffset)
                nextipd = i;
        }
    }

    if (nextipd == -1)

        /* Current PDR is the last one in the file with line
         * numbers.  Use the file's packed line count to compute
         * the PDRs packed line count.
         */

        return (fdr->cbLine - pdr[ipd].cbLineOffset);

    else

        return (pdr[nextipd].cbLineOffset - pdr[ipd].cbLineOffset);
}


expand_file_lines(FDR *fdr,             /* FDR */
                  PDR *pdr,             /* First PDR for FDR */
                  int npdr,             /* PDR count for FDR */
                  unsigned char *pline, /* First packed line for FDR */
                  int numline) {        /* Packed line count for FDR */

    int  ipd;                   /* PDR index */
    int  pli, next_pli;         /* Packed line index */
    int  plcount;               /* Packed line count for PDR */
    long curline;               /* Current source line number */
    long start_address;         /* First address for curline */
    long end_address;           /* First address of next source line */

    /* Iterate through procedures and alternate entries */
```

```
    for (ipd=0; ipd < npdr; ipd++) {

        /* Ignore procedures without line numbers */

        if (pdr[ipd].iline == ilineNil)
            continue;

        /* Identify Procedure or Alternate entry */

        if (pdr[ipd].lnHigh != -1) {
            printf("  Proc %d:\n", ipd);
        } else {
            printf("  Alt Ent %d:\n", ipd);
        }

        start_address = pdr[ipd].adr;      /* 1st address of proc */
        curline = pdr[ipd].lnLow;          /* 1st line number of proc */

        /* Compute packed line count for this PDR */

        plcount = proc_pline_count(fdr, pdr, ipd);

        pli = pdr[ipd].cbLineOffset;       /* Packed line index */
        next_pli = pli + plcount;          /* End index */

        /* Iterate through packed line numbers */

        for (; pli < next_pli; pli++) {

            long delta;         /* temp for computing line delta */

            /* Use the instruction count to compute the first
             * address of the next line number.
             */

            end_address = start_address +
                              (((pline[pli] & 0xfU) + 1) << 2);
            /* Use the line delta to compute the current
             * line number.  Test for extended deltas that
             * use two additional packed line bytes.
             */

            if ((pline[pli] & 0xf0U) == 0x80U) {

                /* extended delta */

                pli++;
                delta = ((signed char)pline[pli]) << 8;
                pli++;
                delta |= pline[pli];

            } else {
                delta = (signed char)pline[pli] >> 4;
            }

            curline += delta;

            /* Display current address range and source line */

            printf("    0x%lx - 0x%lx :  Line %ld\n",
                   start_address, end_address - 4, curline);

            /* Prepare for next iteration */

            start_address = end_address;
        }
    }
}
```

Sample Output

```
% cc -g -o readline readline.c
% ./readline readline
File 1:
  Proc 0:
    0x120001290 - 0x1200012b4 :  Line 11
    0x1200012b8 - 0x1200012c0 :  Line 19
    0x1200012c4 - 0x1200012d8 :  Line 20
    0x1200012dc - 0x1200012e8 :  Line 21
    0x1200012f0 - 0x120001310 :  Line 26
```

```
    0x120001314 - 0x120001330 :  Line 27
...
```

10.2 Extended Source Location Information

This example illustrates the use of structures described in Section 5.3.2.2.2. The following program will read extended source location information and display the intrepreted line numbers. This example includes a few lines of source from a header file in order to illustrate a typical use of ESLI.

Source Listing

usage.h:

```
if (argc < 2) {
  printf("Usage: readesli <OBJECT>\n");
  exit(1);
}
```

readesli.c:

```
/* readesli.c:  Interpret ESLI and display ranges of addresses with
 * file, line, and column numbers.
 *
 * Omissions for simplification purposes:
 *     - file and procedure names.  These can be found by following
 *       the file or procedure's first local symbol entry.
 *     - alternate entries.  These can be included in the output by
 *       comparing the current PC address (maintained in the ESLI
 *       computation) to the address of the next successive
 *       alternate entry procedure descriptor.
 *     - selecting between ESLI and packed line numbers.  If PDRs
 *       have both, ESLI should be prefered.
 *     - relative file interpretation.  File numbers within ESLI
 *       can be converted to actual FDR indexes using the relative
 *       file descriptor table.
 */

#include <stdio.h>
#include <filehdr.h>
#include <scnhdr.h>
#include <sym.h>
#include <symconst.h>
#include <linenum.h>

main(int argc, char **argv){
  FILE          *fd;            /* fopen handle */
  FILHDR        fhead;          /* object file header */
  HDRR          hdrr;           /* symbol table header */
  char          *optbfr;        /* buffer for optimization symbols */
  FDR           *fdr;           /* buffer for FDRs */
  PDR           *pdr;           /* buffer for PDRs */

#include "usage.h"

  /* Open file argument */

  if ((fd = fopen(argv[1], "r")) == (FILE *)NULL) {
    printf("Bad file %s!\n", argv[1]);
    exit(1);
  }

  /* Read file header and test magic id */

  if (fread(&fhead, FILHSZ, 1, fd) != 1) {
    printf("fread filheader!\n");
    exit(1);
  } else if (fhead.f_magic != ALPHAMAGIC) {
    if (fhead.f_magic == ALPHAUMAGIC)
      printf("Compressed object not supported\n");
    else
      printf("%s is not an object file\n", argv[1]);
    exit(1);
  }

  /* Read symbolic header */

  if (fhead.f_symptr == 0) {
```

```
    printf("no syms!\n");
    exit(1);
  }
  fseek(fd, fhead.f_symptr, 0);
  if (fread(&hdrr, sizeof(HDRR), 1, fd) != 1) {
    printf("symheader read failed!\n");
    exit(1);
  }

  /* Test for FDRs, PDRs, and optimization symbols */

  if (!hdrr.ifdMax) {
    printf("No file descriptors!\n");
    exit(1);
  } else if (!hdrr.ipdMax) {
    printf("No procedure descriptors!\n");
    exit(1);
  } else if (hdrr.ioptMax == 0) {
    printf("No ESLI!\n");
    exit(1);
  }

  /* Read FDRs */

  fseek(fd, hdrr.cbFdOffset, 0);
  if (!(fdr = (FDR *)malloc(hdrr.ifdMax * sizeof(FDR)))) {
    printf("FDR malloc failed\n");
    exit(1);
  }
  if (fread(fdr, sizeof(FDR), hdrr.ifdMax, fd) != hdrr.ifdMax) {
    printf("FDR read failed\n");
    exit(1);
  }

  /* Read PDRs */

  fseek(fd, hdrr.cbPdOffset, 0);
  if (!(pdr = (PDR *)malloc(hdrr.ipdMax * sizeof(PDR)))) {
    printf("PDR malloc failed\n");
    exit(1);
  }
  if (fread(pdr, sizeof(PDR), hdrr.ipdMax, fd) != hdrr.ipdMax) {
    printf("PDR read failed\n");
    exit(1);
  }

  /* Read optimization symbols */

  fseek(fd, hdrr.cbOptOffset, 0);
  if (!(optbfr = (char *)malloc(hdrr.ioptMax))) {
    printf("opt malloc failed\n");
    exit(1);
  }
  if (fread(optbfr, 1, hdrr.ioptMax, fd) != hdrr.ioptMax) {
    printf("opt read failed\n");
    exit(1);
  }

  /* Dump ESLI for all procedures */

  dump_esli(fdr, hdrr.ifdMax, pdr, optbfr);
}


dump_esli(FDR *fdr, int ifdmax,    /* FDRs and count */
          PDR *pdr,                /* PDRs */
          char *optbfr) {          /* optimization symbols */

  int ifd;

  /* Iterate through FDRs */

  for (ifd = 0; ifd < ifdmax; ifd++) {

    /* Ignore FDRs without optimization symbols */

    if (fdr[ifd].copt == 0)
      continue;

    printf("File %d:\n", ifd);

    /* Dump ESLI for PDRs in this FDR */
```

```
    dump_esli_for_file(&fdr[ifd],
                       &pdr[fdr[ifd].ipdFirst],
                       fdr[ifd].cpd,
                       optbfr + fdr[ifd].ioptBase);
  }
}


dump_esli_for_file(FDR *fdr,            /* FDR */
                   PDR *pdr,            /* First PDR for FDR */
                   int npdr,            /* PDR count for FDR */
                   char *optbfr) {      /* Optimization symbols for FDR */

  int           ipd;            /* PDR index */
  char          *pdr_optbfr;    /* Optimization symbols for PDR */
  PPODHDR       *ppod;          /* PPOD headers */

  /* Iterate through procedures and dump ESLI */

  for (ipd=0; ipd < npdr; ipd++) {

    /* Ignore procedures without optimization symbols */

    if (pdr[ipd].iopt == ioptNil)
      continue;

    /* Set PPOD header pointer and verify content */

    pdr_optbfr = optbfr + pdr[ipd].iopt;
    ppod = (PPODHDR *)pdr_optbfr;

    if (ppod->ppode_tag != PPODE_STAMP ||
        ppod->ppode_val > PPOD_VERSION) {
      continue;
    }

    /* Search for ESLI PPOD in optimization symbols */

    for (ppod++; ppod->ppode_tag != PPODE_END; ppod++) {
      if (ppod->ppode_tag == PPODE_EXT_SRC) {

        char    *esli_data;     /* ESLI data for procedure */
        int     esli_count;     /* Number of bytes of data */

        if (ppod->ppode_len == 0) {
            /* Immediate data */
            esli_data = (char *)&ppod->ppode_val;
            esli_count = 8;
        } else {
            esli_data = pdr_optbfr + ppod->ppode_val;
            esli_count = ppod->ppode_len;
        }

        printf("  Proc %d:\n", ipd);

        dump_esli_for_proc(esli_data,
                           esli_count,
                           pdr[ipd].adr,
                           pdr[ipd].lnLow);
        break;
      }
    }
  }
}


unsigned long
read_uleb(unsigned char **uleb) {  /* Pointer to LEB pointer */

  /* Read an unsigned LEB value and advance the
   * LEB pointer past the LEB bytes.
   */

  unsigned char *byte;          /* ULEB byte pointer */
  unsigned long value;          /* Return value */
  int           shift;          /* Accumulated bit shift */
  int           morebits;       /* Loop control */

  value = 0;
  shift = 0;
```

```
  byte = *uleb;

  for (morebits=1; morebits; byte++) {

    /* Get 7 bits */
    value |= ((*byte) & 0x7f) << shift;

    /* Increment shift count */
    shift += 7;

    /* Test continue bit */
    morebits = (*byte) & 0x80;
  }

  /* Advance data pointer past ULEB bytes */
  *uleb = byte;

  return(value);
}


long
read_sleb(unsigned char **sleb) {  /* Pointer to SLEB pointer */

  /* Read a signed LEB value and advance the
   * LEB pointer past the LEB bytes.
   */

  unsigned char *byte;          /* SLEB byte pointer */
  long          value;          /* Return value */
  int           shift;          /* Accumulated bit shift */
  int           morebits;       /* Loop control */

  value = 0;
  shift = 0;

  byte = *sleb;

  for (morebits=1; morebits; byte++) {

    /* Get 7 bits */
    value |= ((*byte) & 0x7f) << shift;

    /* Increment shift count */
    shift += 7;

    /* Test continue bit */
    morebits = (*byte) & 0x80;
  }

  /* Extend sign bit if set */
  if ((*byte) & 0x40)
    value |= (-1L << shift);

  /* Advance data pointer past SLEB bytes */
  *sleb = byte;

  return(value);
}


dump_esli_for_proc(char *esli_data,     /* Raw ESLI data */
                   int esli_count,      /* Byte size of ESLI data */
                   long pdr_address,    /* Start address from PDR */
                   long pdr_lnLow) {    /* First source line from PDR */

  /* Read ESLI data for a procedure and display address
   * ranges with file, line, and column information.
   */

  unsigned char *edp;           /* ESLI data pointer */
  unsigned char  cmd;           /* ESLI command */
  int  data_mode = 1;           /* Data mode 1 or 2 */
  int  cmd_mode = 0;            /* Command mode flag */
  long cur_file = 0;            /* Current fileno (not fdr index) */
  long cur_column = 0;          /* Current column number */
  long cur_line = pdr_lnLow;    /* Current line number */
  long start_address;           /* Start of PC address range */
  long end_address;             /* End of PC address range */

  /* Just like packed-line data, ESLI assumes a starting
   * address and computes the end of the PC range along
```

```
 * with the source line information that applies to that
 * range.
 */

start_address = pdr_address;
end_address = start_address;

/* Iterate through ESLI data.  Loop pointer is incremented
 * within loop and LEB reading subroutines.
 */

for (edp = (unsigned char *)esli_data;
     edp < ((unsigned char *)esli_data + esli_count); ) {

  /* Data Modes */

  if (!cmd_mode) {

    /* Test for escape to command mode */

    if (( (*edp) & 0xf0U) == 0x80U) {
      cmd_mode = 1;
      edp++;
      continue;
    }

    /* Use the instruction count to compute the first
     * address of the next line number.
     */

    end_address = start_address +
                  ((( (*edp) & 0xfU) + 1) << 2);
    cur_line += (signed char)(*edp) >> 4;

    if (data_mode == 2)
      cur_column = *(++edp);

    /* Display current address range and source line */

    printf("    0x%lx - 0x%lx :  File %ld Line %ld Col %ld\n",
           start_address, end_address - 4,
           cur_file, cur_line, cur_column);

    /* Prepare for next iteration */

    edp++;
    start_address = end_address;

  } else {

    /* Command Mode */

    cmd = *edp++;

    /* Do command (CMD_MASK is 0x3F) */

    switch(cmd & CMD_MASK) {

      case ADD_PC: /* PC delta */
        end_address += read_sleb(&edp) << 2;
        break;

      case ADD_LINE: /* Line delta */
        cur_line += read_sleb(&edp);
        break;

      case SET_COL: /* Column */
        cur_column = read_uleb(&edp);
        break;

      case SET_FILE: /* File number */
        cur_file = read_uleb(&edp);
        break;

      case SET_DATA_MODE: /* Mode */
        data_mode = read_uleb(&edp);
        break;

      case ADD_LINE_PC: /* Line and PC delta */
        cur_line += read_sleb(&edp);
        end_address += read_sleb(&edp) << 2;
        break;
```

```
        case ADD_LINE_PC_COL: /* Line/PC delta, column */
          cur_line += read_sleb(&edp);
          end_address += read_sleb(&edp) << 2;
          cur_column = read_uleb(&edp);
          break;

        case SET_LINE: /* Line */
          cur_line = read_uleb(&edp);
          break;

        case SET_LINE_COL: /* Line and column */
          cur_line = read_uleb(&edp);
          cur_column = read_uleb(&edp);
          break;

        case SEQUENCE_BREAK: /* PC gap */
          end_address += read_sleb(&edp) << 2;
          start_address = end_address;
          break;

        default:
          fprintf(stderr,"Unkown ESLI command\n");
          exit(1);
      }

      /* check mark (0x80) flag */

      if ((cmd & MARKb) && end_address > start_address) {
        printf("    0x%lx - 0x%lx :  File %ld Line %ld Col %ld\n",
               start_address, end_address - 4,
               cur_file, cur_line, cur_column);
      }

      /* Check resume (0x40) flags */

      if (cmd & RESUMEb) {
        cmd_mode = 0;
      }
    }
  }
}
```

Sample Output

```
% cc -g -o readesli readesli.c
% ./readesli readesli
File 1:
  Proc 0:
    0x1200013b0 - 0x1200013d4 :  File 0 Line 25 Col 0
    0x1200013d8 - 0x1200013e0 :  File 13 Line 1 Col 0
    0x1200013e4 - 0x1200013f8 :  File 13 Line 2 Col 0
    0x1200013fc - 0x12000140c :  File 13 Line 3 Col 0
    0x120001410 - 0x120001430 :  File 0 Line 37 Col 0
    0x120001434 - 0x120001450 :  File 0 Line 38 Col 0
...
```

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Reader's Comments

Tru64 UNIX
Object File and Symbol Table Format Specification
ObjSpec

Compaq welcomes your comments and suggestions on this manual. Your input will help us to write documentation that meets your needs. Please send your suggestions using one of the following methods:

- This postage-paid form
- Internet electronic mail: `readers_comment@zk3.dec.com`
- Fax: (603) 884-0120, Attn: UBPG Publications, ZKO3-3/Y32

If you are not using this form, please be sure you include the name of the document, the page number, and the product name and version.

Please rate this manual:

	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	☐	☐	☐	☐
Clarity (easy to understand)	☐	☐	☐	☐
Organization (structure of subject matter)	☐	☐	☐	☐
Figures (useful)	☐	☐	☐	☐
Examples (useful)	☐	☐	☐	☐
Index (ability to find topic)	☐	☐	☐	☐
Usability (ability to access information quickly)	☐	☐	☐	☐

Please list errors you have found in this manual:

Page	Description
_____	_____
_____	_____
_____	_____
_____	_____

Additional comments or suggestions to improve this manual:

_____
_____
_____
_____

What version of the software described by this manual are you using? _____

Name, title, department _____
Mailing address _____
Electronic mail _____
Telephone _____
Date _____

--------- Do Not Cut or Tear - Fold Here and Tape ----------

COMPAQ

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 33 MAYNARD MA

POSTAGE WILL BE PAID BY ADDRESSEE

COMPAQ COMPUTER CORPORATION
UBPG PUBLICATIONS MANAGER
ZKO3-3/Y32
110 SPIT BROOK RD
NASHUA NH 03062-2698

--------- Do Not Cut or Tear - Fold Here ----------

Cut on This Line