4 Object File Format

4.1 Introduction

The Extended Common Object File Format (XCOFF) is based on ideas taken from the Common Object File Format (COFF), which has been a standard for many Unix offerings. XCOFF defines an object file format that will accommodate facilities like dynamic linking, incremental binding, binding with replacement and reordering, and fast program loading and initiation.

XCOFF is the formal definition of the machine image object and executable files for the PowerOpen ABI. These files are produced by language processors (assemblers and compilers) and binders, and are used primarily by binders and the system loaders.

Note: Bits are listed in Big-Endian order.

The following topics are covered:

4.1.1 Description

The assemblers and compilers produce XCOFF object files as output. The binder combines individual object files into an XCOFF executable file. The system loader reads an XCOFF executable file to create an executable memory image of a program. The symbolic debugger reads an XCOFF executable file to provide symbolic access to functions and variables of an executable memory image of a program that is being debugged.

An XCOFF file contains the following parts:

- A composite header consisting of
  - A file header
  - An optional auxiliary header
  - Section headers, one for each of the file's raw-data sections
- Raw-data sections, at most one per section header
- Optional relocation information for individual raw-data sections
- Optional line number information for individual raw-data sections
- An optional symbol table
- An optional string table for symbol names longer than eight bytes

Not every XCOFF file contains every part. A minimal XCOFF file contains nothing but the file header.

An executable XCOFF file must contain a loader raw-data section in order to dynamically load the executable program into memory for execution. Loading an executable XCOFF file into memory creates the following logical segments:

- A text segment (initialized from the .text section of the XCOFF file).
- A data segment, consisting of initialized data (initialized from the .data section of the XCOFF file) followed by uninitialized data (initialized by the system loader to zero). The length of uninitialized data is specified in the .bss section header of the XCOFF file.

The following Figures (4−1 and 4−2) show the XCOFF object file organization and an XCOFF executable file structure example.

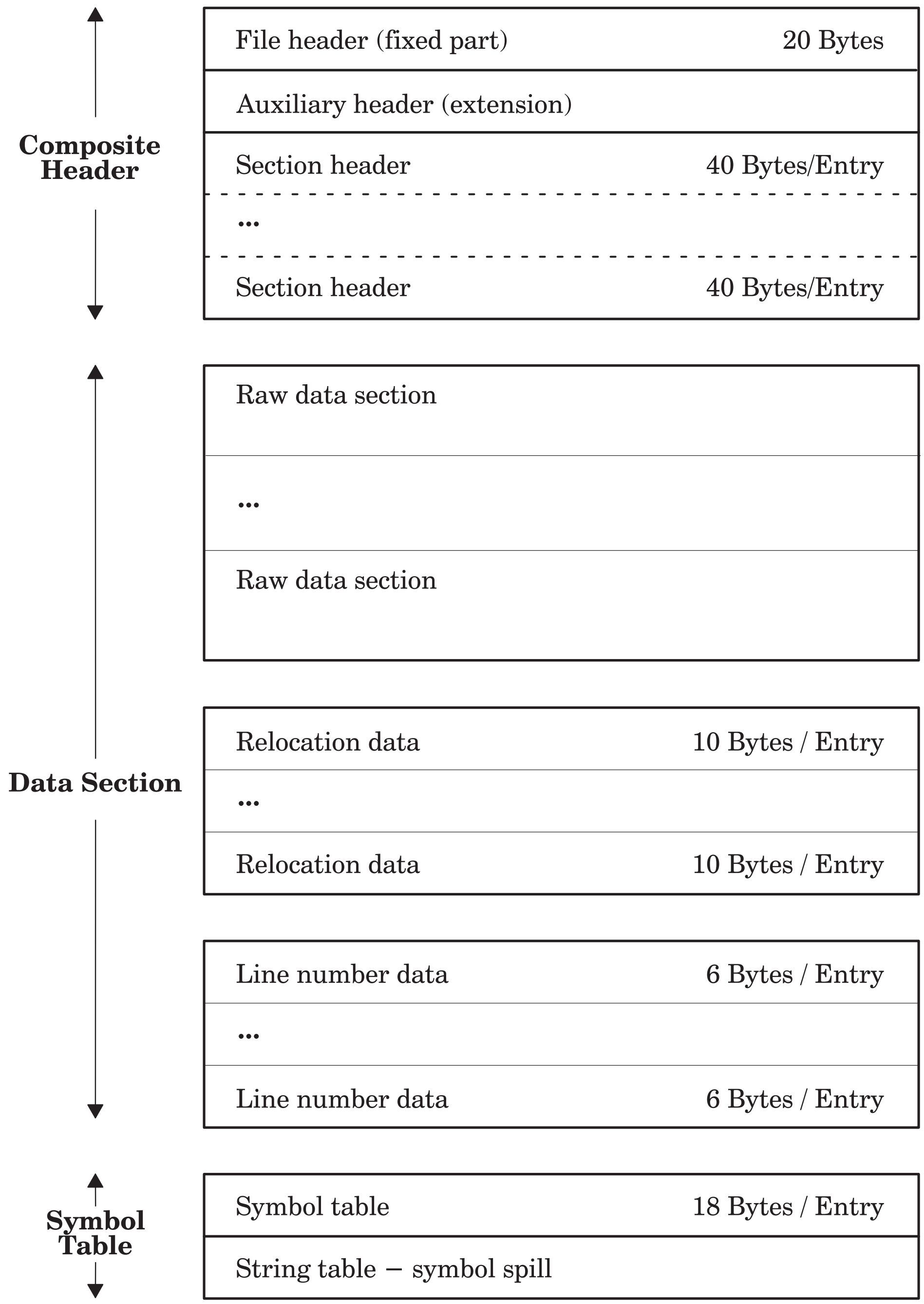

Figure 4–1. Sample XCOFF Object File Organization.

Composite Header	**File header (fixed)**
	Auxiliary header (extension)
	Section Headers .text, .data, .bss section headers .pad section header(s) .loader section header .debug, .typchk, .except, .info section headers
Data Sections	**Raw Data** Executable section: .text, .data, and .pad sections Loader section: .loader – import file IDs – import and export symbols – loader relocation data Compiler comments Symbolic debugger stabstring section Binder type-check section Traceback exception section
	Relocation Data Binder relocation data for .text and .data section
	Line Number Data Symbolic debugger line number data for the .text section
Symbol Table	**Master Symbol Table** Binder and symbolic debugger symbols. This section also contains definitions for file, external, and local symbols.
	String Table Long symbol names.

Figure 4–2. Sample XCOFF **a.out** File Structure.

4.1.2 XCOFF Include Header Files

The include header file **<xcoff.h>** defines the structure of an XCOFF file. The **<xcoff.h>** header file includes the following include header files:

<filehdr.h> defines the file header

<aouthdr.h> defines the auxiliary header

<scnhdr.h> defines the section headers

<loader.h> defines the format of raw data in the .loader section

<typchk.h> defines the format of raw data in the .typchk section

<exceptab.h>
defines the format of raw data in the .except section

<debug.h> defines the format of raw data in the .debug section

<reloc.h> defines the relocation information

<linenum.h> defines the line number information

<syms.h> defines the symbol table format

<storclass.h> defines ordinary storage classes

<dbxstclass.h>
defines storage classes used by the symbolic debuggers

There is an **<a.out.h>** include header file defined that itself includes the **<xcoff.h>** include header file.

4.2 Composite File Header

The following sections describe the XCOFF file header components:

- File Header (**<filehdr.h>**)
- Auxiliary Header (**<aouthdr.h>**)
- Section Headers (**<scnhdr.h>**)

4.2.1 File Header (<filehdr.h>)

The include header file **<filehdr.h>** defines the file header of an XCOFF file. The file header is 20 bytes long and contains the fields shown in Table 4−1.

Table 4−1. File Header **<filehdr.h>** structure.

Offset	Length in Bytes	Sign	Name	Description
0	2	Unsigned	**f_magic**	Target machine
2	2	Unsigned	**f_nscns**	Number of sections
4	4	Signed	**f_timdat**	Time and date of file creation
8	4	Signed	**f_symptr**	Byte offset to symbol table start
12	4	Signed	**f_nsyms**	Number of entries in symbol table
16	2	Unsigned	**f_opthdr**	Number of bytes in optional header
18	2	Unsigned	**f_flags**	Flags (See the Field definitions)

Field definitions:

f_magic Magic number. Specifies the target machine and environment of the object file for PowerOpen ABI. The only valid value is

0x01DF (0737 Octal)

f_nscns The number of section headers contained in the file. The first section header is section header number one; i.e., all references to a section are one-based.

f_timdat The time-and-date stamp indicating when the file was created (number of elapsed seconds since 00:00:00 Universal Coordinated Time (UCT), January 1, 1970). It should be either the actual time or set to 0.

f_symptr A file pointer (byte offset from the beginning of the file) to the start of the symbol table. If **f_nsyms** is zero, then this value is undefined.

f_nsyms The number of entries in the symbol table. Each entry is 18 bytes long.

f_opthdr The length (in bytes) of the auxiliary header. For an XCOFF to be executable, the auxiliary header must exist, and it must be 72 bytes long.

f_flags A bit mask of flags that describe the type of the XCOFF file. The following information defines the flags:

Bit Mask **Flag**

0x0001 **F_RELFLG**
The relocation information for binding has been removed from the file. This flag is used to indicate that relocation information has been stripped from the file. This flag must not be set by compilers if relocation information was not required.

0x0002	**F_EXEC** The file is executable. (There are no unresolved external references.)
0x0004	**F_LNNO** The line numbers have been removed from the file. This flag is used to indicate that line number information has been stripped from the file by a utility program. This flag is not set by language processors, even if no line number information has been generated.
0x0008	**F_LSYMS** This flag indicates that local symbol information has been stripped from the file by a utility program. This flag is not set by the language processor.
0x0010	(Reserved)
0x0020	(Reserved)
0x0040	(Reserved)
0x0080	(Reserved)
0x0100	**F_AR32WR** The byte ordering of a 32-bit word Little-Endian architecture; e.g., Intel 386, 486. Defined for historical accuracy only; it is not to be set or used.
0x0200	**F_AR32W** The byte ordering of a 32-bit word Big-Endian architecture; e.g., POWER.
0x0400	(Reserved)
0x1000	**F_DYNLOAD** The file is dynamically loadable and executable. (If set, F_EXEC must also be set). External references are resolved by way of imports and might contain exports and loader relocation.
0x2000	**F_SHROBJ** The file is a shared object (shared library). The file is separately loadable; it is not normally bound with other objects, and its loader exports symbols are used as automatic imports symbols for other object files.
0x4000	**F_LOADONLY** If the file is a member of an archive, it can be loaded by the system loader. It is ignored by the binder.

4.2.2 Auxiliary Header (<aouthdr.h>)

The auxiliary header contains system and implementation-dependent information, which is usually used for executing the object file. Information in the auxiliary header serves to minimize how much of the file must be processed by the system loader at execution time.

The binder generates an auxiliary header for use by the system loader. Auxiliary headers, if generated by compilers and assemblers, will be ignored by the binder.

The auxiliary header, if it exists, immediately follows the file header.

Note: If **f_opthdr** is 0, the auxiliary header does not exist.

The C language structure for the auxiliary header can be found in the header file **<aouthdr.h>**. The auxiliary header contains the fields shown in Table 4−2.

Table 4−2. Auxiliary Header **<aouthdr.h>** Structure.

Offset	Length in Bytes	Sign	Name	Description
0	2	Signed	**o_mflag**	Flags, how to execute
2	2	Signed	**o_vstamp**	Version
4	4	Signed	**o_tsize**	Text size in bytes
8	4	Signed	**o_dsize**	Initialized data size in bytes
12	4	Signed	**o_bsize**	Uninitialized data size in bytes
16	4	Signed	**o_entry**	Entry point descriptor virtual address
20	4	Unsigned	**o_text_start**	Base address of text (virtual address)
24	4	Unsigned	**o_data_start**	Base address of data (virtual address)
28	4	Unsigned	**o_toc**	Address of TOC
32	2	Signed	**o_snentry**	Section number for entry point
34	2	Signed	**o_sntext**	Section number for .text
36	2	Signed	**o_sndata**	Section number for .data
38	2	Signed	**o_sntoc**	Section number for TOC
40	2	Signed	**o_snloader**	Section number for loader data
42	2	Signed	**o_snbss**	Section number for .bss
44	2	Signed	**o_algntext**	Maximum alignment for .text
46	2	Signed	**o_algndata**	Maximum alignment for .data
48	2	Signed	**o_modtype**	Module type field
50	2	Unsigned	**o_cputype**	CPU-type field
52	4	Unsigned	**o_maxstack**	Maximum stack size allowed (bytes)
56	4	Unsigned	**o_maxdata**	Maximum data size allowed (bytes)
60	12	Unsigned	**o_resv2**	Reserved

The following defines the fields of the auxiliary header. For entries with two labels, the label in parentheses is the original COFF **a.out** name.

Field definitions:

o_mflag (magic) The magic number flag. Tells the operating system the file's execution characteristics. The binder assigns the following:

0x010B Text and data are aligned in the file and may be paged.

o_vstamp (vstamp) The format version identifier for this auxiliary header. 0x0001 is the current value assigned.

o_tsize (tsize) The size (in bytes) of the raw data for the .text section. The .text section typically contains the read-only part of the program. This is the same value as that contained in the **s_size** field of the section header for the .text section. If the section does not exist, the size is set to 0.

o_dsize (dsize) The size (in bytes) of the raw data for the .data section. The .data section contains the initialized data of the program and is writeable. This is the same value as that contained in the **s_size** field of the section header for the .data section. If the section does not exist, the size is set to 0.

o_bsize (bsize) The size (in bytes) of .bss area, which is used for uninitialized variables during execution and is writeable. No raw data exists in the file for the .bss section. This is the same value as that contained in the **s_size** field of the section header for the .bss section. If the section does not exist, the size is set to 0.

o_entry (entry) The virtual address of the entry point function descriptor (see **o_snentry** below). The function descriptor contains the addresses of both the entry point itself and its TOC. The offset of the entry point function descriptor from the beginning of its containing section can be calculated as follows:

Section_offset_value = o_entry − s_paddr [o_snentry − 1],

where **s_paddr** is the virtual address contained in the section header.

o_text_start (text_start) The virtual address of the .text section. This is the address assigned to (used for) the first byte of the .text raw-data section. This is the same value contained in the **s_paddr** field of the section header for the .text section.

o_data_start (data_start) The virtual address of the .data section. This is the address assigned to (used for) the first byte of the .data raw-data section. This is the same value contained in the **s_paddr** field of the section header for the .data section.

For addressing purposes, the .bss section is considered to follow the data section.

The following definitions are used by the system loader. Only a single occurrence of the .text, .data, .bss, and .loader sections may exist in an executable XCOFF file.

o_toc The virtual address of the TOC anchor (see **o_sntoc** below).

o_snentry The number of the file section containing the entry point descriptor.

o_sntext The number of the file .text section.

o_sndata The number of the file .data section.

o_sntoc The number of the file section containing the TOC.

o_snloader The number of the file section containing the system loader information.

o_snbss The number of the file section describing the .bss section.

o_algntext The log (base 2) of the maximum alignment needed for any csect in the .text section.

o_algndata The log (base 2) of the maximum alignment needed for any csect in the .data and .bss sections.

o_modtype Module type field — two-byte ASCII character string. The following module types are defined.

- **1L** Single use — module requires distinct copies of the data section for each load.
- **RE** Reusable — Module requires a private copy of the data area for each process dependent on the module.
- **RO** Read-only — module is read-only and can be used by multiple processes at once.

o_cputype The resultant executable CPU-type identification.
The first byte (high order) is bit encoded to provide CPU-type characteristic information about the resultant executable. The first byte is a bit-mask field, encoded to provide CPU-type characteristics. The second byte is a value. The defined bit values are as follows:

Bit 0 If one then this executable contains only object files of the resultant CPU-type ID. If zero, then this executable contains object files of more than one CPU-type ID.

Bits 1:2 Reserved.

Bit 3 If one then this executable contains object files marked CPU-type Any. If zero, then it does not contain any object file marked Any.

Bits 4:7 Reserved.

The second byte (low order) is a resultant CPU-type ID based on the combination of object files that were bound together. The defined values are as follows:

0 Not used. If both the first and second bytes of **o_cputype** are zero, then a CPU-type of POWER (RISC System/6000) is assumed.

1 PowerPC, 32-bit mode

2 PowerPC, 64-bit mode

3 Common Intersection, PowerPC 32-bit and POWER (RISC System/6000)

4 POWER (RISC System/6000)

5 Any; mix of instructions from different architectures.

6 Specifies mix of POWER and PowerPC instructions (PowerPC 601 RISC microprocessor)

7:127 Reserved (an XCOFF object with these values set is not ABI-conformant)

128:143 Reserved for Apple use (an XCOFF object with these values set is not ABI-conformant)

144:159 Reserved (an XCOFF object with these values set is not ABI-conformant)

160:175 Reserved for Motorola use (an XCOFF object with these values set is not ABI-conformant)

176:223 Reserved (an XCOFF object with these values set is not ABI-conformant)

224 POWER2

225:239 Reserved for IBM use (an XCOFF object with these values set is not ABI-conformant)

240:255 Reserved (an XCOFF object with these values set is not ABI-conformant)

o_maxstack The maximum stack size (in bytes) allowed for this executable. If zero, then the system default maximum stack size is used. The system default depends on the system.

o_maxdata The maximum data size (in bytes) allowed for this executable. If zero, then the system default maximum data size is used. The system default depends on the system.

o_resv2 Twelve bytes reserved for future use. These bytes must be set to 0.

4.2.3 Section Headers (<scnhdr.h>)

Each section of an XCOFF file has a corresponding section header, although some section headers may not have a corresponding raw-data section. A section header provides identification and file accessing information for each section contained within an XCOFF file. Each section header is 40 bytes long. The C language structure for a section header can be found in the header file **<scnhdr.h>**. A section header contains the fields shown in Table 4−3.

Table 4−3. Section Header **<scnhdr.h>** Structure.

Offset	Length in Bytes	Sign	Name	Description
0	8	Signed	**s_name**	Section name
8	4	Unsigned	**s_paddr**	Physical address
12	4	Unsigned	**s_vaddr**	Virtual address (same as physical address)
16	4	Unsigned	**s_size**	Number of bytes in file for this section's raw data
20	4	Signed	**s_scnptr**	Offset in file to raw data for section
24	4	Signed	**s_relptr**	Offset in file to relocation entries for section
28	4	Signed	**s_lnnoptr**	Offset in file to line number entries for section
32	2	Unsigned	**s_nreloc**	Number of relocation entries
34	2	Unsigned	**s_nlnno**	Number of line number entries
36	4	Signed	**s_flags**	Flags to define the section type

Field definitions:

s_name An eight byte, null-padded section name. An eight byte section name will not have a terminating null character. Use the **s_flags** field instead of the **s_name** field to determine a section type. Two sections of the same type may have different names, allowing certain applications to distinguish between them.

s_paddr The physical address of the section. This is the address assigned and used by the language processors and binder for the first byte of the section. For the .text and .data sections, this value also appears in the **o_text_start** and **o_data_start** fields of the auxiliary header, respectively.

s_vaddr The virtual address of the section. This field has the same value as the **s_paddr** field.

s_size The size (in bytes) of this section's raw data.

s_scnptr A file pointer (byte offset from the beginning of the file) to this section's raw data. If this field contains 0, this section has no raw data. Otherwise, the size of the raw data must be contained in the **s_size** field.

s_relptr A file pointer (byte offset from the beginning of the file) to the relocation entries for this section. If this section has no relocation entries, this field must contain 0.

s_lnnoptr A file pointer (byte offset from the beginning of the file) to the line number entries for this section. If this section has no line number entries, this field must contain 0.

s_nreloc The number of relocation entries for this section. If more than 65,534 relocation entries are required, then this field's value will be 65535, and an STYP_OVRFLO section header will contain the actual count of relocation entries in its **s_paddr** field. For more information, refer to the discussion on overflow headers on page 4−16. If this field is set to 65535, **s_nlnno** must also be set to 65535.

s_nlnno The number of line number entries for this section. If more than 65,534 line number entries are required, the field's value will be 65535, and an STYP_OVRFLO section header will contain the actual number of line number entries in its **s_vaddr** field. Refer to the discussion on overflow headers on page 4−16. If this field is set to 65535, **s_nreloc** must also be set to 65535.

s_flags Flags defining the section type. The low-order pair of bytes is used.
A section type identifies both the contents of a section and how the section is to be processed by the binder or the system loader. Only a single bit value may be assigned to the **s_flag**'s field. It must not be the sum or bitwise OR of multiple flags.

Valid bit values are as follows:

Value	Flag
0x0000	(Reserved)
0x0001	(Reserved)
0x0002	(Reserved)
0x0004	(Reserved)
0x0008	**STYP_PAD** pad section. A section of this type is used to provide alignment padding between sections within an XCOFF executable object file. Padding is allowed in an XCOFF file without a corresponding pad section header.
0x0010	(Reserved)
0x0020	**STYP_TEXT** executable text (code) section. A section of this type contains the executable instructions of a program.
0x0040	**STYP_DATA** initialized data section. A section of this type contains the initialized data and the TOC of a program.
0x0080	**STYP_BSS** uninitialized data section. A section of this type contains the uninitialized data of a program.
0x0100	**STYP_EXCEPT** exception section. A section of this type is used to provide information to identify the reason that a trap or exception occurred within an executable object program.
0x0200	**STYP_INFO** comment section. A section of this type is used to provide comments or data to special processing utility programs.
0x0400	(Reserved)
0x0800	(Reserved)

0x1000	**STYP_LOADER** loader section. A section of this type contains information for the system loader to load an XCOFF executable. The information includes imported symbols, exported symbols, relocation data, and shared object names. A section of this type contains external symbol type-check strings used by the binder and the system loader
0x2000	**STYP_DEBUG** debug section. A section of this type contains stabstring (symbol table string) information used by the symbolic debugger.
0x4000	**STYP_TYPCHK** type-check section. A section of this type contains parameter/argument type-check encodings used by the binder and system loader.
0x8000	**STYP_OVRFLO** relocation or line number field overflow section. A section header of this type contains the count of relocation entries and line number entries for some other section. This section header is required whenever either of the counts exceeds 65,534. (Refer to **s_nreloc** and **s_nlnno** on page 4−16.)

4.3 General Information about Sections and Section Headers

Although section headers are defined to provide a large variety of information about the contents of an XCOFF file, in reality, the number of valid section headers is quite small. Programs that process XCOFF files will only recognize some of the valid sections.

Current applications do not use the **s_name** field to determine the section type. Nevertheless, conventional names are used by system tools, as shown in Table 4−4.

Table 4−4. Conventional Header Names.

Description	Conventional Name	Multiple Allowed?	s_flag
Text section	.text	No	STYP_TEXT
Data section	.data	No	STYP_DATA
BSS section	.bss	No	STYP_BSS
Pad section	.pad	Yes	STYP_PAD
Loader section	.loader	No	STYP_LOADER
Debug section	.debug	No	STYP_DEBUG
Type-check section	.typchk	No	STYP_TYPCHK
Exception section	.except	No	STYP_EXCEPT
Overflow section	.ovrflo	Yes	STYP_OVRFLO
Comment section	.info	Yes	STYP_INFO

Some fields of a section header may not always be used, or may have special usage. This pertains to the following fields:

s_name On input, ignored by the binder and system loader. On output, the conventional names are used. (See the above table for conventional names.)

s_scnptr Ignored for .bss sections.

s_relptr Recognized for the .text and .data sections only. No relocation is performed for other sections. Otherwise, it must be 0.

s_lnnoptr Recognized for the .text section only. Otherwise, it must be 0.

4.3.1 Overflow Headers

s_nreloc, s_nlnno

Handling of relocation or line number field overflows. If a section has more than 65,534 relocation or line number entries, both of these fields are set to 65535 (0xFFFF). In this case, an overflow section header with the **s_flags** equal to STYP_OVRFLO is used to contain the relocation and line number count information. The fields in the overflow section header have meanings as follows:

s_nreloc The file section number of the section header that overflowed; i.e., the section header containing 65535 in its **s_nreloc** and **s_nlnno** fields. This provides a reference back to the primary section header. (There is no reference in the primary section header that identifies the appropriate overflow section header. All the section headers must be searched to locate an overflow section header that contains the correct primary section header reference in this field.) This field must have the same value as **s_nlnno**.

s_nlnno The file section number of the section header that overflowed. This field must have the same value as **s_nreloc**.

s_paddr The number of relocation entries actually required. This field is used instead of the **s_nreloc** field of the section header that overflowed.

s_vaddr The number of line number entries actually required. This field is used instead of the **s_nlnno** field of the section header that overflowed.

The **s_size** and **s_scnptr** fields are zero in an overflow section header. The **s_relptr** and **s_lnnoptr** fields must have the same values as in the corresponding primary section header.

4.3.2 Special Meanings in XCOFF Files

An XCOFF file provides special meaning to the following sections:

- .text, .data, and .bss are the memory image of the program. The relocation parts associated with the .text and .data sections contain the full binder relocation information so it can be used for replacement link editing. Only the .text section is associated with a line number part. The parts associated with the executable code are produced by the compilers and assemblers.
- The .pad section is defined as a null-filled, raw-data section that is used to align a subsequent section in the file on some defined boundary such as a file block boundary or a system page boundary.
- The .loader section is a raw-data section defined to contain the dynamic loader information. This section is generated by the binder and has its own, self-contained symbol table and relocation table. There is no reference to this section from the XCOFF symbol table.
- The .debug section is a raw-data section defined to contain the stabstring or dictionary information required by the symbolic debugger.
- The .typchk section is a raw-data section defined to contain parameter and argument type-checking strings.
- The .except section is a raw-data section defined to contain the exception tables used to identify the reasons for an exception in program execution.
- The .info comment section is a raw-data section defined to contain compiler-generated comments or information.
- The .debug, .typchk, .except, and .info sections are produced by compilers and assemblers. References to these sections or items within these sections are made from the XCOFF symbol table.

4.4 Loader Section (<loader.h>)

The loader section contains information required by the system loader to load and relocate an executable XCOFF object. The loader section is generated by the binder. The loader section has an **s_flags** section type flag of STYP_LOADER in the XCOFF section header. By convention, .loader is the loader section name. The data in this section are not referenced by any entries in the XCOFF symbol table.

The loader section consists of the following parts:

- Header fields
- Symbol table
- Relocation table
- Import file ID strings
- Symbol name string table

The C language structure for the loader section can be found in the header file **<loader.h>**.

The loader section contains the fields shown in the following figures and tables.

Loader Section Header	32 bytes
External Symbol 1	24 bytes
External Symbol 2	24 bytes
...	
External Symbol *m*	24 bytes
Relocation entry 1	12 bytes
Relocation entry 2	12 bytes
...	
Relocation entry *n*	12 bytes
Import file ID 1	variable length
Import file ID 2	variable length
...	
Import file ID *o*	variable length
String table entry 1	variable length
String table entry 2	variable length
...	
String table entry *p*	variable length

Figure 4−3. Loader Section Structure.

4.4.1 Header Field Definitions

The following is a description of the loader section's header field definitions.

Table 4−5. Loader Section Header Structure.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**l_version**	Loader section version number
4	4	Signed	**l_nsyms**	Number of symbol table entries
8	4	Signed	**l_nreloc**	Number of relocation table entries
12	4	Unsigned	**l_istlen**	Length of import file ID string table
16	4	Signed	**l_nimpid**	Number of import file IDs
20	4	Unsigned	**l_impoff**	Offset to start of import file IDs
24	4	Unsigned	**l_stlen**	Length of loader section string table
28	4	Unsigned	**l_stoff**	Offset to start of loader section string table

Field definitions:

l_version The loader section version number. This value must be 1.

l_nsyms The number of symbol table entries in the loader section. This value is the actual count of symbol table entries contained in the loader section and does not include the three implicit entries for the .text, .data, and .bss symbol entries.

l_nreloc The number of relocation table entries in the loader section.

l_istlen The byte length of the loader import file ID string table in the loader section.

l_nimpid The number of import file IDs in the import file ID string table.

l_impoff The byte offset from beginning of the loader section to the first import file ID.

l_stlen The byte length of the loader section string table.

l_stoff The byte offset from beginning of the loader section to the first entry in the loader section string table.

4.4.2 Loader Symbol Table Field Definitions

The loader section symbol table contains the symbol table entries that the system loader needs for its import and export symbol processing and dynamic relocation processing.

The **<loader.h>** file defines the symbol table fields. Each entry is 24 bytes long.

There are three implicit external symbols, one each for the .text section, .data section, and .bss section. These are referenced using symbol table index values 0, 1, and 2 respectively. The first symbol contained in the loader section symbol table is referenced using an index value of 3.

Table 4−6. Loader Section External Symbol Table Structure.

Offset	Length in Bytes	Sign	Name	Description
0 . . . . 0 4	8 . . . . .−or−. 4 4	Signed	**l_name** . . . . **l_zeroes** **l_offset**	Symbol name or byte offset into string table . . . . Zero indicates name is in loader section string table Offset of the name into the loader section string table
8	4	Unsigned	**l_value**	Address field
12	2	Signed	**l_scnum**	Section number containing symbol
14	1	Signed	**l_smtype**	Symbol type, export, import flags
15	1	Signed	**l_smclas**	Symbol storage class
16	4	Signed	**l_ifile**	Import file ID, ordinal of import file IDs
20	4	Signed	**l_parm**	Parameter type-check field

The external symbol table fields are as follows:

l_name Specifies an eight-byte, null-padded symbol name if the name is eight bytes or less in length. Otherwise, the field is treated as the following two four-byte integers for accessing the symbol name:

- **l_zeroes** (overlays first word of **l_name**). A value of 0 indicates that the symbol name is in the loader section string table. This field overlays the first word of the **l_name** field. An **l_name** field that has the first four bytes (first word) of zero is used to indicate that the name string is contained in the string table instead of the **l_name** field.

- **l_offset** (overlays second word of **l_name**). The byte offset from the beginning of the loader section string table to the first character of the symbol name (not its length field).

l_value The virtual address of symbol.

l_scnum The number of the XCOFF section that contains the symbol. If the symbol is undefined or imported, the section number is 0. Otherwise, the section number refers to the .text, .data, or .bss section. The section headers are numbered beginning with one.

l_smtype The symbol type, import flag, export flag, and entry flag.

Bits 0:4 are flag bits defined as follows:

Bit 0 Reserved

Bit 1 Imported symbol

Bit 2 Entry point descriptor symbol

Bit 3 Exported symbol

Bit 4 Reserved

Bits 5:7 constitute a three-bit symbol type field with the following definitions:

0 **XTY_ER**
External Reference. Provides a symbol table entry for an

		external (global) symbol that is contained in another XCOFF object file.
	1	**XTY_SD** csect Section Definition. Provides the definition of the smallest initialized unit within an XCOFF object file.
	2	**XTY_LD** label Definition. Provides the definition of the global entry points within initialized csects. An uninitialized csect of type XTY_CM may not contain a label definition.
	3	**XTY_CM** Common (BSS uninitialized data) csect definition. Provides the definition of the smallest uninitialized unit within an XCOFF object file.
	4:7	Reserved.

l_smclas The storage mapping class of the symbol as defined in **<syms.h>** of **x_smclas** of the csect auxiliary symbol table entry #defines. Allowed values are as follows: XMC_PR, XMC_RO, XMC_GL, XMC_XO, XMC_SV, XMC_RW, XMC_TC, XMC_TD, XMC_DS, XMC_UA, XMC_BS, or XMC_UC. For more information, see page 4−44.

l_ifile Identifies the import file ID string. It is the ordinal value of the position of the import file ID string in the loader section import file ID name string table. For an imported symbol, the value of 0 in this field identifies the symbol as a deferred resolve symbol to the system loader. A deferred resolve symbol is one whose address can remain unresolved following the processing of the system loader. If the symbol was not imported, this field must have a value of 0. (For more information, see Chapter 5, Program Loading and Dynamic Linking.)

l_parm Offset to the parameter type-check string. The byte offset is from the beginning of the loader section string table. The byte offset points to the first character of the parameter type-check string (not to its length field). The parameter type-check string information is described in the type-check section of this document on page 4−27. A value of 0 in the **l_parm** field indicates that the parameter type-checking string is not present for this symbol, and the symbol will be treated as having a universal hash. (For more information on type-checking, refer to Chapter 3−62.)

4.4.3 Loader Relocation Table Field Definitions

The loader section relocation table contains all the relocation information that the system loader needs to properly relocate an executable XCOFF file when it is loaded. The **<loader.h>** file defines the relocation table fields. Each entry in the loader section relocation table is 12 bytes long. The structure for the first 10 bytes of each relocation entry is the same as the regular XCOFF relocation entry, which is defined in the header file **<reloc.h>**. (See page 4−32, Relocation Entry Structure Table.)

Table 4–7. Loader Section Relocation Table.

Offset	Length in Bytes	Sign	Name	Description
0	4	Unsigned	**l_vaddr**	Address field
4	4	Signed	**l_symndx**	Loader section symbol table index of item that is being referenced
8	2	Signed	**l_rtype**	Relocation type
10	2	Signed	**l_rsecnm**	File section number being relocated

Name	Description
l_vaddr	The virtual address of the relocatable reference.
l_symndx	The loader section symbol table index (*n*th entry) of the symbol that is being referenced. The values of 0, 1, and 2 are implicit references to the .text, .data, and .bss sections, respectively. Symbol index 3 is the index for the first symbol actually contained in the loader section symbol table.
l_rtype	The relocation size and type. (The value of this field is the same as the **r_type** field in the **<reloc.h>** header file. ((**l_rtype** >> 8) & 0x1F) + 1 is the relocation size in bits. Note that #bits / 8 is the number of bytes affected, and the bits affected are right justified within the bytes affected.
l_rsecnm	The section number of the .data section being relocated (associated with **l_vaddr** field). The sections are numbered, starting with one. In a future release of the PowerOpen ABI `l_rsecnm` may be enhanced to support other sections than .data.

4.4.4 Loader Import File ID Name Table Definition

The loader section import file ID name strings provides a list of executable XCOFF files that the system must properly load before this executable XCOFF file can execute. This list does not contain the names of the executable XCOFF files that the named files themselves may also require to be loaded.

Table 4–8. Loader Section Import File IDs — Contains Variable Length Strings.

Offset	Length in Bytes	Name	Description
0	$n1$	**l_impidpath**	Import file ID path string, null delimited
$n1+1$	$n2$	**l_impidbase**	Import file ID base string, null delimited
$n1+n2+2$	$n3$	**l_impidmem**	Import file ID member string, null delimited
			Fields repeat for each import file ID

Each import file ID name consists of three null-delimited strings — a file path name, a base name, and an archive member name.

The first import file ID is a default LIBPATH value to be used by the system. The LIBPATH information consists of file paths separated by colons. There is no base name or archive member name, so the file path is followed by three null bytes.

Each entry in the import file ID name table consists of the following:

- The import file ID path name

- Null delimiter (ASCII null character)
- The import file ID basename
- Null delimiter (ASCII null character)
- The import file ID archive file member name
- Null delimiter (ASCII null character)

For example, **/usr/lib/NULL mylib.aNULL shr.oNULL**

Note: NULL = ASCII null character, value 0.

4.4.5 Loader String Table Definition

The loader section string table contains the names of symbols longer than eight characters as well as the parameter type-checking strings. Each string consists of a two-byte length field followed by the string.

Table 4–9. Loader Section String Table — Contains Variable Length Strings.

Offset	Length in Bytes	Name	Description
0	2	**l_parmstlen**	Length of string
2	*n*	**l_parmst**	Symbol name string (null-delimited) or parameter type string (not null-delimited)
Two fields repeat			

Symbol names are null-terminated. The value in the length field includes the length of the string plus the length of the null terminator. The length field value does not include its own length field size.

The parameter type-checking strings contain binary values and are not null-terminated. The value in the length field includes the length of the string only. The length field value does not include the length of its own length field size. The loader section symbol table entries contain a byte offset value that points to the first character of the string. It does not point to the length field.

4.4.6 Loader Section Header Contents

The following is the contents of the section header fields for the loader section. Refer to the section on Section Headers on page 4–13 for the description of the fields.

Name	Contents
s_name	.loader
s_paddr	Zero
s_vaddr	Zero
s_size	The size (in bytes) of the loader section
s_scnptr	Offset from the beginning of the XCOFF file to the first byte of the loader section data
s_relptr	Zero
s_lnnoptr	Zero
s_nreloc	Zero
s_nlnno	Zero

s_flags STYP_LOADER

4.5 Debug Section

The debug section contains the symbolic debugger stabstrings (symbol table strings). It is generated by the compilers and assemblers. It provides symbol attribute information for use by the symbolic debugger. The debug section has a section type flag of STYP_DEBUG in the XCOFF section header. By convention, .debug is the debug section name. The data in this section is referenced from entries in the XCOFF symbol table. A stabstring is a null-terminated character string. Each string is preceded by a two-byte length field.

For information on stabstring grammar, refer to Appendix C, page C−3.

4.5.1 Field Definitions

The following two fields are repeated for each symbolic debugger stabstring:

- A two-byte length field containing the length of the string. The value contained in the length field includes the length of the terminating null character but it does not include the length of the length field itself.
- The symbolic debugger stabstring.

4.5.2 Debug Section Header Contents

The following is the contents of the section header fields for the debug section. Refer to the section on Section Headers for the description of the fields.

Name	Contents
s_name	.debug
s_paddr	Zero
s_vaddr	Zero
s_size	The size (in bytes) of the debug section.
s_scnptr	Offset from the beginning of the XCOFF file to the first byte of the debug section data.
s_relptr	Zero
s_lnnoptr	Zero
s_nreloc	Zero
s_nlnno	Zero
s_flags	STYP_DEBUG

4.6 Type-Check Section

The type-check section contains information encodings about external symbols (data and functions) and function arguments. These type encodings are produced by language processors, and they are used by the binder and the system loader to detect variable mismatches and argument interface errors prior to the execution of a program. Type encodings are designed to allow the maximum checking required by the semantics of each particular language supported, as well as to provide protection to applications written in more than one language. The type-check section has a section type flag of STYP_TYPCHK in the XCOFF section header. By convention, .typchk is the type-check section name. The encodings in this section are referenced from entries in the XCOFF symbol table.

Refer to Chapter 3−62 for more information on type encoding.

4.6.1 Field Definitions

The following two fields are repeated for each parameter type-checking string:

- A two-byte field containing the length of the type-checking string. The value contained in the length field does not include the length of the length field itself.
- The parameter type-checking hash string.

4.6.2 Type Encoding and Checking Format for Data

The type encoding and checking mechanism features four-part hash encoding that provides some flexibility in checking. The mechanism also incorporates a unique value called UNIVERSAL that matches any code. The UNIVERSAL hash can be used as an escape mechanism for assembly programs or for programs where type information or function interfaces might not be known. The UNIVERSAL hash is four blank ASCII characters having a value of 0x20202020.

Following is a description of the contents of each of the four fields associated with the 12-byte type encoding:

Hash Length Field
A two-byte field containing the length of the subsequent three fields, which contain the actual hash code. This field has a value of 10 (2+4+4).

Language ID Field
The language identifier is a two-byte value representing each language. These values are the same as those defined for the traceback table. For more information, refer to Mandatory Traceback Table Fields in Chapter 3, page 3−55.

General Hash Field
The general hash is a four-byte field representing the most general form by which a data symbol or function can be described. It is the greatest common denominator among all supported languages. The universal hash must be generated when the information about a symbol is not available or is incomplete. The general hash is language independent in that if the same symbols are used in two different languages, then the hashes produced by the two language processors must match each other in order for the binding to succeed, if binder-checking is turned on.

Language Hash Field
The language hash is a four-byte field containing a more detailed language-specific representation of the contents of the general hash. This field enables the strictest type checking desired for a given language. Intra-language binding uses the language hash field and checks the field only when both symbols have the same language identifier.

The language hash is language-dependent in that if the same symbols are used in the same languages, then the hashes produced by the two language processors must match each other in order for the binding to succeed, if binder-checking is turned on.

4.6.3 Section Header Contents

The following is the contents of the section header fields for the type-check section. Refer to the section on Section Headers on page 4−13 for the description of the following fields:

Name	Contents
s_name	.typchk
s_paddr	Zero
s_vaddr	Zero
s_size	The size (in bytes) of the type-check section
s_scnptr	Offset from the beginning of the XCOFF file to the first byte of the type-check section data
s_relptr	Zero
s_lnnoptr	Zero
s_nreloc	Zero
s_nlnno	Zero
s_flags	STYP_TYPCHK

4.7 Comment Section

The comment section contains miscellaneous comment information. It is generated by compilers and assemblers and provides a means for the compilers and assemblers to record special ASCII textual information within an XCOFF object file. The comment section has a section type flag of STYP_INFO in the XCOFF section header. By convention, .info is the comment section name.

4.7.1 Field Definitions

The following two fields are repeated for each set of comment data:

- A four-byte length field containing the length of the comment data. The value contained in the length field does not include the length of the length field.
- The comment data.

4.7.2 Section Header Contents

The following is the contents of the section header fields for the comment section. Refer to the section on Section Headers on page 4−13 for the description of the fields.

Name	Contents
s_name	.info
s_paddr	Zero
s_vaddr	Zero
s_size	The size (in bytes) of the comment section
s_scnptr	Offset from the beginning of the XCOFF file to the first byte of the comment section data
s_relptr	Zero
s_lnnoptr	Zero
s_nreloc	Zero
s_nlnno	Zero
s_flags	STYP_INFO

4.8 Exception Section

The exception section contains addresses of trap instructions, source language identification codes, and trap reason codes and is produced by compilers and assemblers. It is used at or after run-time to identify the reason that a specific trap or exception occurred. Within the exception section there is one group of exception table entries per function. The exception section has a section type flag of STYP_EXCEPT in the XCOFF section header. By convention, .except is the exception section name. Data in the exception section is referenced from entries in the XCOFF symbol table.

An exception table entry with a reason code value of 0 contains the symbol table index to a function's symbol table entry (symbol table entry **n_sclass** value of C_EXT or C_HIDEXT). Reference from the symbol table to an entry in the exception table is via the function auxiliary symbol table entry. (For more information, see Function Auxiliary Entry for C_EXT and C_HIDEXT on page 4–47.)

The C language structure for the exception section entries can be found in the file **<exceptab.h>**.

The exception section entries contain the fields shown in Tables 4–10 and 4–11.

Table 4–10. Initial Exception Section Entry for Function.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**e_symndx**	Symbol table index for function
4	1	Signed	**e_lang**	Compiler language ID code
5	1	Signed	**e_reason**	Value 0 (exception reason code 0)

Table 4–11. Subsequent Exception Section Entries for Function.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**e_paddr**	Address of the trap instruction
4	1	Signed	**e_lang**	Compiler language ID code
5	1	Signed	**e_reason**	Trap exception reason code

Field Definitions:

e_symndx (overlay with the **e_paddr** field). When the **e_reason** field is 0, then this field is the symbol table index of the function.

e_paddr (overlay with the **e_symndx** field). When the **e_reason** field is non-zero, then this field is the virtual address of the trap instruction.

e_lang An eight-bit field that identifies the source language:

ID	Language
0x00	C
0x01	FORTRAN
0x02	Pascal
0x03	Ada
0x04	PL/I
0x05	Basic
0x06	Lisp

0x07	Cobol
0x08	Modula2
0x09	C++
0x0A	RPG
0x0B	PL8, PLIX
0x0C	Assembly
0x0D:0xFF	Reserved

e_reason An eight-bit compiler-dependent trap exception reason code. Zero is not a valid trap exception reason code as it is used to indicate the start of exception table entries for a new function.

4.8.1 Section Header Contents

The following is the contents of the section header fields for the exception section. Refer to the section on Section Headers on page 4–13 for the description of the fields.

Name	Contents
s_name	.except
s_paddr	Zero
s_vaddr	Zero
s_size	The size (in bytes) of the exception section
s_scnptr	Offset from the beginning of the XCOFF file to the first byte of the exception section data
s_relptr	Zero
s_lnnoptr	Zero
s_nreloc	Zero
s_nlnno	Zero
s_flags	STYP_EXCEPT

4.9 Relocation Information for XCOFF File (<reloc.h>)

The text section and data section may have relocation information. The relocation information is used by the binder to modify the text section and data section contents with address and byte offset information of individual XCOFF object files collected into an XCOFF executable file.

The compilers and assemblers are responsible for generating the required relocation entries for the .text and .data sections.

The binder generates relocation information for the loader section, as required by the system loader.

Each relocation entry of the text and data section is 10 bytes long. A relocation entry in the .loader section is 12 bytes long and is explained in the loader section description in this document. (See page 4−22, Relocation Table Field Definitions.) The C language structure for a relocation entry can be found in the file **<reloc.h>**. A relocation entry contains the fields shown in Table 4−12.

Table 4−12. Relocation Entry Structure.

Offset	Length in Bytes	Sign	Name	Description
0	4	Unsigned	**r_vaddr**	Virtual address (position) in section to be relocated
4	4	Unsigned	**r_symndx**	Symbol table index of item that is referenced
8	1	Unsigned	**r_rsize** **r_sign** **r_fixup** **−−−** **r_len**	Relocation size 1 bit **r_sign** Signed/unsigned 1 bit **r_fixup** Fixup generated 1 bit Reserved 5 bits **r_len** Bit length (minus one) of field to modify
9	1	Unsigned	**r_rtype**	Relocation type

The relocation entries for the .text and .data sections are part of their respective sections. The relocation entry refers to a location to be modified. The relocation entries for a section must be in ascending address order.

The loader section contains a single set of relocation entries used by the system loader. A section number is required within each relocation entry to identify the section that needs to be modified.

4.9.1 Field Definitions

The following defines the relocation-information fields:

r_vaddr The virtual address of the value that requires modification (fix-up) by the binder. The byte offset value to the data that requires modification from the beginning of the section that contains the data can be calculated as follows:

offset_in_section = r_vaddr − s_paddr[section being processed minus one]

r_symndx A zero-based index into the XCOFF symbol table for locating the referenced symbol. The symbol table entry contains an address that is used to calculate a modification value to be applied at the **r_vaddr** relocation address.

r_rsize XCOFF relocation size and sign. Its contents are as follows:

r_sign A one-bit field that indicates whether the relocation reference is signed (1) or unsigned (0).

r_fixup A one-bit field that indicates that the binder replaced the original instruction by a branch instruction to a special fixup instruction sequence.

−−−−− A one-bit field that is reserved. It has an expected value of 0.

r_len A five-bit field that contains the bit length minus one of the relocatable reference. The current architecture allows for fields of up to 32 bits to be relocated.

r_rtype An eight-bit relocation type field that indicates to the binder which relocation algorithm to use for calculating the modification (fix-up) value to be applied at the relocatable reference location; i.e., the location specified by the **r_vaddr** field. ((l_rtype >> 8) & 0x1F) + 1 is the relocation size in bits.
Note that #bits / 8 is the number of bytes affected, and the bits affected are right justified within the bytes affected. The following relocation types are defined:

0x00 **R_POS**
Positive Relocation. Provides the address of the symbol specified by **r_symndx**.

0x01 **R_NEG**
Negative Relocation. Provides the negative of the address of the symbol specified by **r_symndx**.

0x02 **R_REL**
Relative to self. Provides a displacement value between the address of the **r_symndx** specified symbol and the address of the csect to be modified.

0x03 **R_TOC**
Relative to TOC. Provides a displacement value that is the difference between the address value in the **r_symndx** specified symbol and the address of the TOC anchor csect. The TOC anchor csect is the one with a symbol table csect auxiliary entry that has the **x_smclas** (storage mapping class) of XMC_TC0. The TOC anchor csect must be of zero length. There may be only one TOC anchor csect per XCOFF object file or executable.

0x04 **R_TRL**
TOC Relative Indirect Load (modifiable L to LA). Provides a displacement value that is the difference between the address value in the **r_symndx** specified symbol and the address of the TOC anchor csect. This relocation entry is treated the same as an **R_TOC** relocation entry. It provides the following additional capabilities: The instruction that is referenced by the **r_vaddr** is a load instruction which can be modified by the binder to become a compute address instruction. Changing an instruction from a load to a compute address instruction avoids a storage reference during execution. A compute address instruction can be used if the address that is contained at the **r_symndx** referenced address has a value that itself references a **r_symndx** that can be accessed with a valid in-range displacement relative to the TOC anchor address; i.e., the

target of the TOC entry is from −32768 to 32767, inclusive, from the TOC anchor address. If a compute address instruction is generated by the binder, then the **R_TRL** relocation type is changed to become a **R_TRLA**. This is to allow the reverse transformation if required. Compilers and assemblers are permitted to generate this relocation type.

0x13 **R_TRLA**
TOC Relative Load Address (modifiable LA to L). Provides a displacement value that is the difference between the address value in the **r_symndx** specified symbol and the address of the TOC anchor csect. This relocation entry is treated the same as an **R_TOC** relocation entry. It provides the following capabilities: The instruction that is referenced by the **r_vaddr** is a compute address instruction, which would be modified by the binder to become a load instruction whenever the calculated displacement value is outside the valid displacement range relative to the TOC anchor address. This relocation type provides the binder with a means to transform a compute address instruction back to a load instruction whenever required. If a load instruction is generated by the binder, the **R_TRLA** relocation type is changed to become an **R_TRL** type. Compilers and assemblers are *not* permitted to generate this relocation type.

0x05 **R_GL**
Global Linkage — External TOC address. Provides the address of the TOC associated with a defined external symbol. The external symbol whose TOC address is required is specified by the **r_symndx** field of the relocation entry. This relocation entry provides a method of accessing the address of the TOC contained within the same executable as the **r_symndx** external symbol is defined.

0x06 **R_TCL**
Local object TOC address. Provides the address of the TOC associated with a defined external symbol. The external symbol for which the TOC address is required is specified by the **r_symndx** field of the relocation entry. It has been determined that the external symbol is defined locally within the resultant executable. This relocation entry provides a method of accessing the address of the TOC contained within the same executable as the **r_symndx** external symbol is defined.

0x0C **R_RL**
Treated the same as **R_POS**.

0x0D **R_RLA**
Treated the same as **R_POS**.

0x0F **R_REF**
Non-relocating reference to prevent garbage collection (by the binder) of a symbol. This relocation type is intended to provide compilers and assemblers a method to specify that a given csect has a dependency upon another csect without using any space in the actual csect. The reason for making the dependency reference is to prevent the binder from garbage collecting

(eliminating) a csect for which another csect has an implicit dependency.

0x08	**R_BA** Branch Absolute, non-modifiable. Provides the address of the symbol specified by **r_symndx** as the target address of a branch instruction.
0x18	**R_RBA** Branch absolute, modifiable. Permits a branch absolute instruction to be changed into a branch relative instruction with the resultant relocation type being changed to **R_RBR**. An **R_RBA** is treated the same as **R_BA** if the instruction is not changed. The instruction would be changed whenever the target of the branch does not have an **x_smclas** value of **XMC_XO** during the final bind.
0x0A	**R_BR** Branch relative to self, non-modifiable. Provides a displacement value between the address of the **r_symndx** specified symbol and the address of the csect containing the branch instruction to be modified. The **R_BR** relocation type is the standard branch relocation type used by the language processors for the PowerOpen. This along with **glink** code allows an executable object file to have its text section be position independent.
0x1A	**R_RBR** Branch relative to self, modifiable. Permits a branch relative instruction to be changed into a branch absolute instruction with the resultant relocation type being changed to **R_RBA**. An **R_RBR** is treated the same as **R_BR** if the instruction is not changed. The instruction would be changed whenever the target of the branch has an **x_smclas** value of **XMC_XO** during the final bind.

4.9.2 Additional Relocation Features

Standard Unix practice is to retain relocation information only for unresolved references or references between distinct sections. Once a reference is resolved, the relocation information is discarded. This is sufficient for an incremental bind and a fixed address space model. For the XCOFF file to provide capability for rebinding, the relocation information has the following characteristics:

- Resolved relocation entries are not discarded.
- The relocation information for the text and data sections is used by the binder for linking and relinking. It is ignored by the system loader.
- A special loader section is defined that contains relocation data required by the system loader. The loader section is ignored during relinking but may be regenerated.

4.10 Line Number Information for XCOFF File (<linenum.h>)

Line number entries are used by the symbolic debugger to debug code at the source level. When present, there is one line number entry for every source line that can have a symbolic debugger breakpoint. The line numbers are grouped by function. The beginning of each function is identified by **l_lnno** containing a value of 0. The first field, **l_symndx**, is the symbol table index to the C_EXT or C_HIDEXT symbol table entry for the function.

Each line number entry is six bytes long. The C language structure for a line number entry can be found in the file **<linenum.h>**. A line number entry contains the fields shown in Tables 4−13 and 4−14.

Table 4−13. Initial Line Number Entry for Function.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**l_symndx**	Symbol table index for function
4	2	Unsigned	**l_lnno**	Value 0 (line number zero)

Table 4−14. Subsequent Line Number Entries for Function.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**l_paddr**	Address at which break point can be inserted
4	2	Unsigned	**l_lnno**	Line number relative to start of function

The following list defines the line number entries:

l_symndx (overlay with the **l_paddr** field). The (zero-based) symbol table index to the function name. When the **l_lnno** field is 0, then this interpretation of the field is used.

l_paddr (overlay with the **l_symndx** field). The virtual address of the first instruction of the code that is associated with the line number. When the **l_lnno** field is not 0, then this interpretation of the field is used. For a function, the line number entries are ordered by non-decreasing values of **l_paddr**.

l_lnno The line number relative to start of function, or 0 to indicate the start of the line number entries for a new function.

4.11 Symbol Table Information

One composite symbol table is defined for an XCOFF file. The symbol table contains information required by both the binder (external symbols) and the symbolic debugger (function definitions and internal and external symbols).

The symbol table consists of a list of eight-byte, fixed-length entries. Each symbol represented in the symbol table consists of at least one fixed-length entry, and some are followed by auxiliary entries of the same size.

For each external symbol, one or more auxiliary entries are required that provide additional information concerning the external symbol. There are three major types of external symbols of interest to the binder — those that define replaceable units or csects, those that define the external names for functions or entry points within csects, and those that reference the names of external functions in another XCOFF object. For symbols defining a replaceable unit (csect), a csect auxiliary entry is used to define the length and storage-mapping class of the csect. For symbols defining external names for functions within a csect, the csect auxiliary entry is used to point to the containing csect, point to parameter type-checking information, and point to symbolic debugger information for the function. For symbols referencing the name of an external function, a csect auxiliary entry identifies the symbol as an external reference, and points to parameter type-checking information.

The following describes the general ordering of an XCOFF symbol table. There are C_FILE symbol table entries used to bracket all the symbols contained within a source file. There are the symbolic debugger symbol table entries that are of file scope following the C_FILE entry but before the first csect entry. There are then C_EXT or C_HIDEXT csect definition symbol table entries used to define and bracket all the symbols contained within a csect. All the symbolic debugger symbol table entries following a csect definition symbol table entry are associated with that csect.

The ordering of the symbol table must be arranged by the compilers and assemblers to both accommodate the symbolic debugger requirements and to permit effective management by the binder of the different sections of the object file as a result of such binder actions as garbage collection, incremental bind, and rebinding. This ordering is required by the binder so that if a csect is deleted or replaced, all the symbol table information associated with the csect can also be deleted or replaced. Likewise, if all the csects associated with a source file are deleted or replaced, all the symbol table and related information associated with the file can also be deleted or replaced.

The symbol table can also provide entries to identify special information in a comment section of the XCOFF object file.

4.11.1 Symbol Table Layout

The general (non-mandatory) ordering of the symbol table is shown in Figure 4−4.

```
un_external  Undefined global symbols

.file        Source file 1
    stab             Global Debug symbols of a file
  csect              Replaceable unit definition (code)
    function         Local/External function
      stab           Debug and local symbols of function
    function         Local/External function
      stab           Debug and local symbols of function
   ..............
  csect              Replaceable unit definition (local statics)
    stab             Debug and local statics of file
   ..............
  csect              Relocatable unit definition (global data)
    external         Defined global symbol
      stab           Debug info for global symbol
   ..............
.file        Source file 2
    stab             Global Debug symbols of a file
  csect              Replaceable unit definition (code)
    function         Local/External function
      stab           Debug and local symbols of function
   ..............
  csect              Replaceable unit definition (local statics)
    stab             Debug and Local statics of file
   ..............
  csect              Replaceable unit definition (global data)
    external         Defined global symbol
      stab           Debug info for global symbol
   ..............       ..............
.file        Source file n
   ..............
```

Figure 4–4. Sample Symbol Table Organization.

4.11.2 Symbol Table Entry (<syms.h>)

Each symbol, regardless of storage class and type, has a fixed-format entry in the symbol table. In addition, some symbol types may have additional (auxiliary) symbol table entries immediately following the fixed format entry. Each entry in the symbol table is 18 bytes long. The C language structure for a symbol table entry can be found in the header file **<syms.h>**. The index for the first entry in the symbol table is zero. Table 4–15 shows the structure of the fixed-format part of each symbol in the symbol table.

Table 4–15. Symbol Table Entry Format.

Offset	Length in Bytes	Sign	Name	Description
0	8	Signed	**n_name**	Symbol name
....	.....–or–		....	....
0	4		**n_zeroes**	Zero indicates name in string table or .debug section
4	4		**n_offset**	Offset of the name in string table or .debug section
8	4	Unsigned	**n_value**	Symbol value; storage class dependent
12	2	Signed	**n_scnum**	Section number of symbol
14	2	Unsigned	**n_type**	Basic and derived type specification
16	1	Signed	**n_sclass**	Storage Class of symbol
17	1	Signed	**n_numaux**	Number of auxiliary entries

Field definitions:

n_name The null-padded eight-byte symbol name or symbolic debugger stabstring. The **n_sclass** field is used to determine if the field is a symbol name or symbolic debugger stabstring. By convention, an **n_sclass** value with the high order bit on indicates that this field is a symbolic debugger stabstring.

n_zeroes (overlays first four bytes of **n_name**). A value of 0 indicates that the symbol name is in the string table or .debug section.

n_offset (overlays second four bytes of **n_name**). The byte offset to the symbol name in the string table or .debug section. The byte offset is relative to the start of the string table or .debug section. A byte offset value of 0 is a null or zero-length symbol name.

n_value The symbol value. The definition of the contents of the symbol value field is storage-class dependent.

Content	Storage Class, n_sclass
Relocatable address	C_EXT, C_HIDEXT, C_FCN, C_BLOCK, C_STAT
Zero	C_GSYM, C_BCOMM, C_DECL, C_ENTRY, C_ESTAT, C_ECOMM
Offset in csect	C_FUN, C_STSYM
Offset in file	C_BINCL, C_EINCL
Symbol table index	C_FILE, C_BSTAT
Offset relative to stack frame	C_LSYM, C_PSYM
Register number	C_RPSYM, C_RSYM
Offset within common block	C_ECOML
For more information, refer to page 4–50.	

n_scnum The section number associated with the symbol.

–2 **N_DEBUG**
A symbolic debugging symbol.

−1 **N_ABS**
An absolute symbol. The symbol has a value but is not relocatable.

0 **N_UNDEF**
Undefined or uninitialized external symbol. The symbol is undefined if the **n_value** field is zero. The symbol is defined but uninitialized if the **n_value** field is non-zero. The **n_value** field provides the size in bytes of the storage required by the symbol.

Any other value is the XCOFF section number where the symbol was defined.

n_type Used by C_FILE; not used by other storage classes. For more information, see page 4−41.

n_sclass The storage class of the symbol. The **<storclass.h>** and **<dbxstclass.h>** header files contain the definitions of the storage classes. Refer to page 4−49 for a description of each storage class value and its effect on the other symbol table entry fields.

n_numaux The number of auxiliary entries for the symbol. If more than one auxiliary entry is required for a symbol, then the order of the auxiliary entries is determined by convention. That is, no flag field in the auxiliary entries can be used to distinguish one type of auxiliary entry from another.

4.12 Symbol Table Auxiliary Information

The symbol table contains auxiliary entries to provide supplemental information for a symbol. The auxiliary entries for a symbol follow its symbol table entry. The size of each auxiliary entry is the same as a symbol table entry (18 bytes). The format and quantity of auxiliary entries depend on the storage class (**n_sclass**) and type (**n_type**) of the symbol table entry.

The convention for symbols with a storage class of C_EXT or C_HIDEXT is that those symbols with more than one auxiliary entry must have the csect auxiliary entry as the last auxiliary entry.

4.12.1 File Auxiliary Entry for C_FILE Symbols

The file auxiliary symbol table entry is defined to contain the source file name and compiler-related strings. A file auxiliary entry is optional and is used with a symbol table entry that has a storage class value of C_FILE (see page 4−49). The C language structure for a file auxiliary entry can be found in the **x_file** structure in the header file **<syms.h>**.

The C_FILE symbol provides source file name information, source language ID and CPU version ID information and optionally provides compiler version and time stamp information.

n_type identifies the source language of the source file and the CPU version ID of the compiled object file. **n_type** information is as follows:

- **Source Language ID —** One-byte field. Overlays the high order byte of the **n_type** field. Contains the source language identifier. The values for this field are defined in the **e_lang** field on page 4−30. It can be used by the symbolic debuggers to determine the source language.

  The optional values for this field can also be 248 (TB_OBJECT) for symbols from object files with no C_FILE symbol table entry; or 249 (TB_FRONT) or 250 (TB_BACK) for generated entries used to provide debugging information. If the source language is TB_FRONT or TB_BACK, the eight−character name field will begin with ' ', '\0'. If the source language is TB_FRONT, then the third byte will be the stabstring compaction level for the object file, and the n_offset field will contain the symbol table index of the TB_BACK symbol table entry, if it exists, and 0 otherwise.

- **CPU Version ID —** One-byte field. Overlays the low-order byte of the **n_type** field. The following values are defined:

  0 Not used. If both the Source Language ID and CPU Version ID are zero, then the binder will assume a CPU Version ID of POWER (RISC System/6000)

  1 PowerPC, 32-bit mode

  2 PowerPC, 64-bit mode

  3 Common Intersection, PowerPC 32-bit and POWER (RISC System/6000)

  4 POWER (RISC System/6000)

  5 Any; mix of instructions from different architectures.

  6 Specifies mix of POWER and PowerPC instructions (PowerPC 601 RISC microprocessor)

  7:127 Reserved (an XCOFF object with these values set is not ABI-conformant)

  128:143 Reserved for Apple use (an XCOFF object with these values set is not ABI-conformant)

  144:159 Reserved (an XCOFF object with these values set is not ABI-conformant)

  160:175 Reserved for Motorola use (an XCOFF object with these values set is not ABI-conformant)

176:223 Reserved (an XCOFF object with these values set is not ABI-conformant)

224 POWER2

225:239 Reserved for IBM use (an XCOFF object with these values set is not ABI-conformant)

240:255 Reserved (an XCOFF object with these values set is not ABI-conformant)

The binder will use this field to produce a resultant CPU-type field in the XCOFF auxiliary header.

Table 4−16. File Name Auxiliary Entry Format — 18 Bytes.

Offset	Length in Bytes	Sign	Name	Description
0	14	Signed	**x_fname**	Source file string
....	... −or−.......		....	....
0	4		**x_zeroes**	Zero means file string in string table
4	4		**x_offset**	Offset of file string in string table
14	1	Unsigned	**x_ftype**	File string type
15	3		**−−−−−**	Reserved; must contain 0

Field definitions:

x_fname Source file name or compiler-related string.

- **x_zeroes** (overlays first word of **x_fname)**. A value of 0 indicates that the source file string is in the string table.
- **x_offset** (overlays second word of **x_fname**). Offset from the beginning of the string table to the first character of the source file string.

x_ftype Source file string type.

0	**XFT_FN**	Source file name
1	**XFT_CT**	Compile time stamp
2	**XFT_CV**	Compiler version number
128	**XFT_CD**	Compiler defined information

−−−−−− Reserved. Three bytes containing 0.

If the file auxiliary entry is not used, then the **n_name** field of the symbol table entry contains the source file name. If the file auxiliary entry is used, then the **n_name** field of the symbol table entry contains .file as its string, and the first file auxiliary entry (by convention) contains the source file name. More than one file auxiliary entry is permitted for a given symbol table entry. The **n_numaux** field contains the number of file auxiliary entries.

4.12.2 csect Auxiliary Entry for C_EXT and C_HIDEXT Symbols

The csect auxiliary entry identifies csects (section definitions), entry points (label definitions), and external references (label declarations). A csect auxiliary entry is required for each symbol table entry that has a storage class value of C_EXT or C_HIDEXT (defined on page 4−49). By convention, the csect auxiliary entry must be the last auxiliary entry for any external symbol that has more than one auxiliary entry. The C language structure for a csect auxiliary entry can be found in the **x_csect** structure in the header file **<syms.h>**.

Table 4−17. csect Auxiliary Entry Format.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**x_scnlen**	Meaning is dependent on **x_smtyp**, see below
4	4	Signed	**x_parmhash**	Offset of parm type-check hash in .typchk section
8	2	Unsigned	**x_snhash**	.typchk section number
10	1	Unsigned	**x_smtyp**	Symbol alignment and type five-bit symbol alignment (log 2) three-bit symbol type, see below
11	1	Unsigned	**x_smclas**	Storage mapping class
12	4	Signed	**x_stab**	Reserved symbolic debugger stab entry index in .debug section
16	2	Unsigned	**x_snstab**	Reserved .debug section number

Field definitions:

x_scnlen Meaning is dependent on **x_smtyp** as follows:

If Then

XTY_SD **x_scnlen** contains the csect length.

XTY_LD **x_scnlen** contains the symbol table index of the containing csect.

XTY_CM **x_scnlen** contains the csect length.

XTY_ER **x_scnlen** contains zero.

x_parmhash Byte offset of the parameter type-check string in the .typchk section. The byte offset is from the beginning of the .typchk section in an XCOFF file. The byte offset points to the first character of the parameter type-check string (not to its length field). The parameter type-check string information is described in Chapter 3−62. A value of 0 in this field (**x_parmhash**) indicates that the parameter type-checking string is not present for this symbol, and the symbol will be treated as having a universal hash. Ignored for C_HIDEXT symbols.

x_snhash The .typchk section number. The XCOFF section number containing the parameter type-checking strings. Sections are numbered, starting with one. For compatibility with object files generated by some compilers, if **x_parmhash** is not equal to zero but **x_snhash** does equal zero, then use the first .typchk section in the file. Ignored for C_HIDEXT symbols.

x_smtyp Symbol alignment and type.

Bits 0:4 A five-bit csect address alignment value (log base 2). The alignment value is used only when the lower three-bit value of **x_smtyp** is XTY_SD, XTY_CM, or XTY_ER. For example, a value of 3 in this field indicates 2^3, or 8, meaning that the csect is to be aligned on an eight-byte address value.

Bits 5:7 Three-bit symbol type with the following definitions:

0 XTY_ER External reference. Provides a symbol table entry for an external (global) symbol that is contained in another XCOFF object file.

1 XTY_SD csect section definition. Provides the definition of the smallest initialized unit within an XCOFF file.

2 XTY_LD Specifies a label definition of the global entry points for initialized csects. An uninitialized csect of type XTY_CM may not contain a label definition.

3 XTY_CM Specifies a common (BSS uninitialized data) csect definition of the smallest uninitialized unit within an XCOFF file.

4:7 Reserved.

x_smclas The csect storage mapping class. This field permits the binder to arrange csects by their storage mapping class.
The storage mapping class field is used only when the lower three-bit value of **x_smtyp** is XTY_SD, XTY_CM, XTY_ER, or XTY_LD and must otherwise be zero.

The following storage mapping classes are defined:

Value Class	Description (read-only classes, normally mapped to .text section)
0 XMC_PR	Program code. The csect contains the executable instructions of the program.
1 XMC_RO	Read-only constant. The csect contains data that is constant and will not change during execution of the program.
2 XMC_DB	Debug dictionary table. The csect contains symbolic debugging data or exception processing data. This storage mapping class was defined to permit compilers with special symbolic debugging or exception processing requirements to place data in csects that are loaded at execution time but that can be collected separate from the executable code of the program.
6 XMC_GL	Global linkage. The csect provides the interface code necessary to handle csect relative calls to a target symbol that can be out of module. This global linkage csect has the same name as the target symbol and becomes the local target of the relative call, thus maintaining position independent code within the .text section of the executable XCOFF object file.
7 XMC_XO	Extended operation. A csect of this type has no dependency on (references through) the TOC. It is intended to reside at a fixed address in memory such that it can be the target of a branch to an absolute address.
12 XMC_TI	Reserved.
13 XMC_TB	Reserved.

Value Class	Description (read/write classes, normally mapped to .data or .bss section)
5 XMC_RW	Read/write data. A csect of this type contains initialized or uninitialized data that is permitted to be modified during

program execution. If the **x_smtyp** is XTY_SD then the csect contains initialized data and is mapped into the .data section. If the **x_smtyp** is XTY_CM then the csect is uninitialized and is mapped into the .bss section. Typically, all the initialized static data from a C source file is contained in a single csect of this type. It would have an **n_sclass** value of C_HIDEXT with a local data name. An initialized definition for a global data scalar or structure from a C source file is contained in its own csect of this type. It would have an **n_sclass** value of C_EXT with the global data name. A csect of this type is accessible by name references from other object files.

15 XMC_TC0 TOC anchor for TOC addressability. This is a zero-length csect whose **n_value** address provides the base address for TOC relative addressability. Only one csect of type XMC_TC0 is permitted per .data section of an XCOFF object file. In implementations that permit compilers and assemblers to generate multiple .data sections, there must be a csect of type XMC_TC0 in each section that contains data that is referenced (by way of a relocation entry) as a TOC relative data item. Some hardware architectures have a limit to the value that a relative displacement field within a load instruction may contain. This limit then becomes the inherent limit that the size of the TOC of an executable XCOFF object file may be. For PowerPC this limit is 65,536 bytes or 16,384 four-byte TOC entries.

3 XMC_TC General TOC entry. Generally a csect that is four bytes in length and contains the address of another csect or global symbol. It is used to provide addressability to other csects or symbols. The symbols may be contained in either the local executable XCOFF object or in another executable XCOFF object. Special processing semantics are used in the binder in an attempt to eliminate duplicate TOC entries as follows:

- Symbols that have an **n_sclass** value of C_EXT are global symbols and must have names (non-null **n_name** field). These symbols require no special TOC processing logic to combine duplicate entries. Duplicate entries with the same **n_name** are combined into a single entry.
- Symbols that have an **n_sclass** value of C_HIDEXT are not global symbols, and duplicate entries are resolved by context. Two of these symbols will be combined as duplicate into a single entry whenever the following conditions are met:
  - The **n_name** fields are the same. That is they have either a null name or the same name string.
  - Each is four bytes long.
  - Each has a relocation symbol table entry that references external symbols with the same name.

To minimize the number of duplicate TOC entries that cannot be combined by the binder, it is recommended that compilers and assemblers adhere to a common naming convention for

TOC entries. By convention, compilers and assemblers produce TOC entries that have an **n_sclass** value of C_HIDEXT and have an **n_name** string that is the same as the **n_name** of the symbol that the TOC entry addresses.

16 XMC_TD Scalar data entry in the TOC. A csect that is a special form of a XMC_RW csect that can be directly accessed from the TOC by compiler generated code. This permits some frequently used global symbols to be accessed directly from the TOC rather than indirectly through an address pointer csect contained within the TOC. A csect of type XMC_TD has the following characteristics:

- The compiler generates code that is TOC relative to directly access the data contained in the csect of type XMC_TD.
- By convention, it is four bytes long or less.
- It has initialized data that is permitted to be modified during program execution.
- If a same named csect of type XMC_RW or XMC_UA exists, it will be replaced by the XMC_TD csect.

For the cases where the TOC scalar cannot reside in the TOC, the binder must be capable of transforming the compiler generated TOC relative instructions into the conventional indirect addressing instruction sequence. This transformation would be necessary if the TOC scalar were contained within a shared object.

10 XMC_DS Function descriptor csect. A csect that is a function descriptor. A function descriptor contains the following three values:

- The address of the executable code for a function.
- The address of the TOC anchor (TOC base address) of the module that contains the function.
- The environment pointer (used by languages such as Pascal and PL/I).

There is only one function descriptor csect for a function, and it must be contained within the same executable as the function itself. The function descriptor has a **n_sclass** value of C_EXT and has an **n_name** value that is the same as the name of the function in the source file. It is the function descriptors that are imported and exported to and from an executable XCOFF file.

8 XMC_SV Supervisor call descriptor. A csect that is a supervisor call descriptor. The supervisor call descriptors are contained within the operating system kernel. To an application program, the reference to a supervisor call descriptor is treated the same as a reference to a regular function descriptor. It is through the import/export mechanism that a function descriptor is treated as a supervisor call descriptor.

4 XMC_UA Unclassified, treated as read/write. A csect of this type is considered a read/write csect. It is frequently produced by an assembler or object file translator program that does not know the true classification of the resultant csect.

9 XMC_BS BSS class (uninitialized static internal). A csect of this type is uninitialized, and is intended to be mapped into the .bss section. This type of csect must have a **x_smtyp** value of XTY_CM.

11 XMC_UC Unnamed FORTRAN common. A csect of this type is intended for unnamed and uninitialized FORTRAN common. It is intended to be mapped into the .bss section. This type of csect must have a **x_smtyp** value of XTY_CM.

x_stab (Reserved).

x_snstab (Reserved).

4.12.3 Function Auxiliary Entry for C_EXT and C_HIDEXT Symbols

The function auxiliary symbol table entry is defined in XCOFF to contain reference and size information associated with a defined function. It is produced by the language processors for use by symbolic debuggers.

The function auxiliary symbol table entry format is shown in Table 4−18.

Table 4−18. Function Auxiliary Entry Format.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**x_tagndx**	Tag index
4	4	Signed	**x_fsize**	Size of function (in bytes)
8	4	Signed	**x_lnnoptr**	File pointer to line number
12	4	Signed	**x_endndx**	Symbol table index of next entry beyond this function
16	2	Unsigned	**x_tvndx**	Unused

Field definitions:

x_tagndx Tag index. File pointer to the first exception table entry for the function — this value is the byte offset from the beginning of the XCOFF object file. If no exception table exists, the value is set to zero.

x_fsize Size of the function (in bytes).

x_lnnoptr File pointer to the first line number entry for the function — this value is the byte offset from the beginning of the XCOFF object file. If no line number section exists, the value is set to zero.

x_endndx Symbol table index (zero-based) of the next entry beyond this function. If there is not a next entry, the value is set to zero.

x_tvndx Reserved.

4.12.4 Block Auxiliary Entry for C_BLOCK and C_FCN Symbols

The section auxiliary symbol table entry is defined in XCOFF to provide information associated with the begin and end blocks of functions. It is produced by compilers for use by symbolic debuggers.

Table 4−19. Table Entry Format.

Offset	Length in Bytes	Sign	Name	Description
0	4		−−−−−	Reserved
4	2	Unsigned	**x_lnno**	Source line number
6	12		−−−−−	Reserved

Field definitions:

−−−−− Reserved.

x_lnno Source file line number. The size of this field limits this value to a maximum of 65535.

−−−−− Reserved.

4.12.5 Section Auxiliary Entry for C_STAT Symbols

The section auxiliary symbol table entry ID is defined in XCOFF to provide information in the symbol table concerning the size of sections produced by a compiler or assembler. The generation of this information by a compiler is optional and is ignored.

Table 4−20. Section Auxiliary Entry Format.

Offset	Length in Bytes	Sign	Name	Description
0	4	Signed	**x_scnlen**	Section length
4	2	Unsigned	**x_nreloc**	Number of relocation entries
6	2	Unsigned	**x_nlinno**	Number of line numbers
8	10		−−−−−−	Reserved

Field definitions:

x_scnlen Section length in bytes.

x_nreloc Number of relocation entries. The size of this field limits this value to a maximum of 65535.

x_nlinno Number of line numbers. The size of this field limits this value to a maximum of 65535.

−−−−−− Reserved.

4.13 Additional Symbol Table Entry Information

4.13.1 Symbol Table Field Contents by Storage Class

This section defines the symbol table field contents for each of the defined storage classes that are used in XCOFF. Table 4−21 lists storage class entries in alphabetic order.

Table 4−21. Symbol Table by Storage Class.

Class Definition	n_name	n_value	n_scnum	Aux. Entry
C_BCOMM 135 Beginning of common block	Name of the common block.*	zero, undefined	**N_DEBUG**	
C_BINCL 108 Beginning of include file	Source name of the include file.**	File pointer	**N_DEBUG**	
C_BLOCK 100 Beginning or end, inner block	.bb or .eb	Relocatable address	**N_SCNUM**	BLOCK
C_BSTAT 143 Beginning of static block	.bs	Symbol table index	**N_DEBUG**	
C_DECL 140 Declaration of object (type)	Debugger stabstring*	zero, undefined	**N_SCNUM**	
C_ECOML 136 Local member of common block	Debugger stabstring*	Offset within common block	**N_ABS**	
C_ECOMM 137 End of common block	Debugger stabstring*	zero, undefined	**N_DEBUG**	
C_EINCL 109 End of include file	Source name of the include file.**	File pointer	**N_DEBUG**	
C_ENTRY 141 Alternate entry	Debugger stabstring*	zero, undefined	**N_DEBUG**	
C_ESTAT 144 End of static block	.es	zero, undefined	**N_DEBUG**	
C_EXT 2 External symbol (Defines external symbols for binder processing)	Symbol Name**	Relocatable address	**N_SCNUM** or **N_UNDEF**	FUNCTION CSECT
C_FCN 101 Beginning or end of function	.bf or .ef	Relocatable address	**N_SCNUM**	BLOCK
C_FILE 103 Source file name and compiler information	.file or source file name (if no aux. entries).**	Symbol table index	**N_DEBUG**	FILE
C_FUN 142 Function or procedure	Debugger stabstring*	Offset within containing csect	**N_ABS**	
C_GSYM 128 Global variable	Debugger stabstring*	zero, undefined	**N_DEBUG**	
C_HIDEXT 107 Unnamed external symbol	Symbol Name or null**	Relocatable address	**N_SCNUM**	FUNCTION CSECT
C_LSYM 129 Automatic variable allocated on stack	Debugger stabstring*	Offset relative to stack frame	**N_ABS**	
C_NULL 0 Symbol table entry marked for deletion.		0x00DE1E00		Any

Class Definition	n_name	n_value	n_scnum	Aux. Entry
C_PSYM 130 Argument to function allocated on stack	Debugger stabstring*	Offset relative to stack frame	**N_ABS**	
C_RPSYM 132 Argument to function or procedure stored in register	Debugger stabstring*	Register number	**N_ABS**	
C_RSYM 131 Register variable – – –	Debugger stabstring*	Register number	**N_ABS**	
C_STAT 3 Static symbol (Some compilers generate these symbols in the symbol table to identify size of the .text, .data, and .bss sections. Not used nor preserved by binder.)	Symbol Name**	Relocatable address	**N_SCNUM**	SECTION
C_STSYM 133 Statically allocated symbol	Debugger stabstring*	Offset within csect	**N_ABS**	
C_TCSYM 134 Reserved	Debugger stabstring*			

Note: *For long name, **n_offset** value is an offset into the debug section.

Note: **For long name, **n_offset** value is an offset into the string table.

4.13.1.1 Storage Classes by Usage and Symbol Value Classification

The symbol storage classes (**n_sclass**) that are used and relocated by the binder. (The symbol values — **n_value** — are addresses):

Class	Description
C_EXT	External Symbol
C_HIDEXT	Unnamed external symbol
C_BLOCK	Beginning or end of an inner block (.bb or .eb)
C_FCN	Beginning or end of a function (.bf or .ef)

The symbol storage class (**n_sclass**) that is not used and is discarded by the binder:

C_STAT Static symbol (Contained in statics csect. Some compilers generate these symbols in the symbol table to identify size of the .text, .data, and .bss sections.)

Symbol classes which are used by the binder and symbolic debugger or other utilities for file scope and accessing purposes:

C_FILE The source file name. The **n_value** field holds the symbol index of the next file entry. The **n_name** field is the name of the file.

C_BINCL The beginning of include header file. The **n_value** field is the line number byte offset in the object file to the first line number from the include file.

C_EINCL The end of include header file. The **n_value** field is the line number byte offset in object file to last line number from the include file.

Symbol classes that exist only for symbolic debugging purposes and are not relocated:

C_BCOMM Beginning of common block, value is meaningless, name is name of common block.

C_ECOML Local member of common block, value is byte offset within common block.

C_ECOMM End of common block, value is meaningless.

C_BSTAT Beginning of static block, value is symbol table index of csect containing static symbols, name is .bs.

C_ESTAT End of static block, value is meaningless, name is .es.

C_DECL Declaration of object (type declarations), value is undefined.

C_ENTRY Alternate entry (FORTRAN) and has a corresponding C_EXT symbol. The value field is undefined.

C_FUN Function or procedure. May have corresponding C_EXT symbol. The value field is byte offset within containing csect.

C_GSYM Global variable and has a corresponding C_EXT symbol. The value field is undefined.

C_LSYM Automatic variable allocated on stack, value is byte offset relative to stack frame.

C_PSYM Argument to function allocated on stack, value is byte offset relative to stack frame.

C_RSYM Register variable. The value is register number.

C_RPSYM Argument to function or procedure stored in a register. The value is register number where argument is stored.

C_STSYM Statically allocated symbol. The value is byte offset within csect pointed to by containing C_BSTAT entry.

4.14 String Table

The string table contains the names of symbols that are longer than eight characters. If the string table is present, the first four bytes contain the length (in bytes) of the string table, including the length of this length field. The remainder of the table is a sequence of null-terminated ASCII strings. If the **n_zeroes** field in the symbol table entry is zero, then the **n_offset** field gives the byte offset into the string table of the name of the symbol.

If a string table is not used, it may be omitted entirely, or a string table consisting of only the length field (containing a value of 0 or 4) may be used. Specifying a value of 4 is preferable. Table 4−22 shows string table organization.

Table 4−22. String Table Organization.

Offset	Length in Bytes	Description
0	4	Length of string table
4	*n*	Symbol name string; null-terminated
Field repeats for each symbol name		