

Constraint	Indicates a constraint resource.
SubResource	This value is currently not recognized.

A resource definition can be modified with the following:

Type	Indicates the data type of the resource. The data type specified must be listed in the data type definition.
ResourceLiteral	Indicates the keyword used in the UIL file to reference the resource. In AIXwindows, the resource name is the same as the resource literal name (ResourceLiteral).
InternalLiteral	Forces the value of the internal symbol table literal definition of the resource name. This modifier is used only to circumvent symbol table definitions hard-coded into the UIL compiler and should be used sparingly.
Alias	Indicates alternate names for the resources used in a UIL specification.
Related	Special purpose field that allows resources that act as a counter for the current resources to be related to the resource. UIL automatically sets the value of this related resource to the number of items in the compiled instance of the <i>ResourceName</i> type.
Default	Indicates the default value of the resource.
DocName	Defines an arbitrary string for use in the documentation. This value is currently not recognized.

An example of the usage of data type, control list, and class definitions is shown:

```
Resource
  XtbNfirstResource : Argument
  { Type = OddNumber;
    Default = "XtbOLD_VALUE";};
  XtbNsecondResource : Argument
  { Type = NewString;
    Default = "XtbNEW_STRING";};
  XtbNnewResource : Argument
  { Type = OddNumber;
    Default = "XtbODD_NUMBER";};
```

Related Information

The **UIL** file format.

XCOFF Object File Format

Purpose

The extended common object file format (XCOFF) is the object file format for the operating system. XCOFF combines the standard common object file format (COFF) with the TOC module format concept, which provides for dynamic linking and replacement of units within an object file. In AIX 4.3, XCOFF has been extended to provide for 64-bit object files and executable files.

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

The default name for an XCOFF executable file is **a.out**.

Note: This information lists bits in big-endian order.

Read the following information to learn more about XCOFF object files:

- Composite File Header
- Sections and Section Headers

- Relocation Information for XCOFF File (reloc.h)
- Line Number Information for XCOFF File (linenum.h)
- Symbol Table Information
- dbx Stabstrings

Writing Applications that Use XCOFF Declarations

Programs can be written to understand 32-bit XCOFF files, 64-bit XCOFF files, or both. The programs themselves may be compiled in 32-bit mode or 64-bit mode to create 32-bit or 64-bit programs. By defining preprocessor macros, applications can select the proper structure definitions from the XCOFF header files.

Note: This document uses "XCOFF32" and "XCOFF64" as shorthand for "32-bit XCOFF" and "64-bit XCOFF", respectively.

Selecting XCOFF32 Declarations

To select the XCOFF32 definitions, an application merely needs to include the appropriate header files. Only XCOFF32 structures, fields, and preprocessor defines will be included. Structure names and field names will match those in previous versions of the operating system, so existing programs can be recompiled without change.

Note: Existing uses of shorthand type notation (e.g., UINT, ULONG) have been removed.

Selecting XCOFF64 Declarations

To select the XCOFF64 definitions, an application should define the preprocessor macro `__XCOFF64__`. When XCOFF header files are included, the structures, fields, and preprocessor defines for XCOFF64 will be included. Where possible, the structure names and field names are identical to the XCOFF32 names, but field sizes and offsets may differ.

Selecting Both XCOFF32 and XCOFF64 Declarations

To select structure definitions for both XCOFF32 and XCOFF64, an application should define both the preprocessor macros `__XCOFF32__` and `__XCOFF64__`. This will define structures for both kinds of XCOFF files. Structures and typedef names for XCOFF64 files will have the suffix "_64" added to them. (Consult the header files for details.)

Selecting Hybrid XCOFF Declarations

An application may choose to select single structures that contain field definitions for both XCOFF32 and XCOFF64 files. For fields that have the same size and offset in both XCOFF32 and XCOFF64 definitions, the field names are retained. For fields whose size or offset differ between XCOFF32 and XCOFF64 definitions, the XCOFF32 fields have a "32" suffix, while the XCOFF64 fields have a "64" suffix. To select hybrid structure definitions, an application should define the preprocessor macro `__XCOFF_HYBRID__`. For example, the symbol table definition (in `/usr/include/syms.h`) will have the names `n_offset32` and `n_offset64`, which should be used for the 32-bit XCOFF and 64-bit XCOFF respectively.

Understanding XCOFF

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.

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, which is used for all symbol names in XCOFF64 and for symbol names longer than 8 bytes in XCOFF32.

Not every XCOFF file contains every part. A minimal XCOFF file contains only the file header.

Object and Executable Files

XCOFF object files and executable files are similar in structure. An XCOFF executable file (or "module") must contain an auxiliary header, a loader section header, and a loader section.

The loader raw-data section contains information needed to dynamically load a module into memory for execution. Loading an XCOFF executable 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 0). The length of uninitialized data is specified in the `.bss` section header of the XCOFF file.

The XCOFF file Organization illustrates the structure of the XCOFF object file.

XCOFF Header Files

The **xcoff.h** file defines the structure of the XCOFF file. The **xcoff.h** file includes the following 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 <code>.loader</code> section.
typchk.h	Defines the format of raw data in the <code>.typchk</code> section.
exceptab.h	Defines the format of raw data in the <code>.except</code> section.
debug.h	Defines the format of raw data in the <code>.debug</code> 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.

The **a.out.h** file includes the **xcoff.h** file. All of the XCOFF include files include the **xcoff32_64.h** file.

For more information on sections of the XCOFF object file, see "Sections and Section Headers." For more information on the symbol table, see "Symbol Table Information." For more information on the string table, see "String Table." For more information on the Debug section, see "Debug Section."

Composite File Header

The following sections describe the XCOFF composite file header components:

- File Header (`filehdr.h`)
- Auxiliary Header (`aouthdr.h`)
- Section Headers (`scnhdr.h`)

File Header (`filehdr.h`)

The **filehdr.h** file defines the file header of an XCOFF file. The file header is 20 bytes long in XCOFF32 and 24 bytes long in XCOFF64. The structure contains the fields shown in the following table.

Table 13. File Header Structure (Defined in filehdr.h)

Field Name and Description	XCOFF32	XCOFF64
f_magic Target machine	<ul style="list-style-type: none"> • Offset: 0 • Length: 2 	<ul style="list-style-type: none"> • Offset: 0 • Length: 2
f_nscns Number of sections	<ul style="list-style-type: none"> • Offset: 2 • Length: 2 	<ul style="list-style-type: none"> • Offset: 2 • Length: 2
f_timdat Time and date of file creation	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 4 • Length: 4
f_symptr⁺ Byte offset to symbol table start	<ul style="list-style-type: none"> • Offset: 8 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 8
f_nsyms⁺ Number of entries in symbol table	<ul style="list-style-type: none"> • Offset: 12 • Length: 4 	<ul style="list-style-type: none"> • Offset: 20 • Length: 4
f_opthdr Number of bytes in optional header	<ul style="list-style-type: none"> • Offset: 16 • Length: 2 	<ul style="list-style-type: none"> • Offset: 16 • Length: 2
f_flags Flags (see "Field Definitions")	<ul style="list-style-type: none"> • Offset: 18 • Length: 2 	<ul style="list-style-type: none"> • Offset: 18 • Length: 2

+ Use "32" or "64" suffix when `__XCOFF_HYBRID__` is defined.

Field Definitions:

<code>f_magic</code>	Specifies an integer known as the <i>magic number</i> , which specifies the target machine and environment of the object file. For XCOFF32, the only valid value is 0x01DF (0737 Octal). For XCOFF64 on AIX 4.3 and earlier, the only valid value is 0x01EF (0757 Octal). For XCOFF64 on AIX 5.1 and later, the only valid value is 0x01F7 (0767 Octal). Symbolic names for these values are found in the file, <code>/usr/include/filehdr.h</code> .
<code>f_nscns</code>	Specifies the number of section headers contained in the file. The first section header is section header number one; all references to a section are one-based.
<code>f_timdat</code>	Specifies when the file was created (number of elapsed seconds since 00:00:00 Universal Coordinated Time (UCT), January 1, 1970). This field should specify either the actual time or be set to a value of 0.
<code>f_symptr</code>	Specifies a file pointer (byte offset from the beginning of the file) to the start of the symbol table. If the value of the <code>f_nsyms</code> field is 0, then this value is undefined.
<code>f_nsyms</code>	Specifies the number of entries in the symbol table. Each symbol table entry is 18 bytes long.
<code>f_opthdr</code>	Specifies the length, in bytes, of the auxiliary header. For an XCOFF file to be executable, the auxiliary header must exist and be <code>_AOUTHSZ_EXEC</code> bytes long. (<code>_AOUTHSZ_EXEC</code> is defined in <code>outhdr.h</code> .)

f_flags

Specifies a bit mask of flags that describe the type of the object file. The following information defines the flags:

Bit Mask

Flag

0x0001 F_RELFLG

Indicates that the relocation information for binding has been removed from the file. This flag must not be set by compilers, even if relocation information was not required.

0x0002 F_EXEC

Indicates that the file is executable. No unresolved external references exist.

0x0004 F_LNNO

Indicates that line numbers have been stripped from the file by a utility program. This flag is not set by compilers, even if no line-number information has been generated.

0x0008 Reserved.

0x0010 F_FDPR_PROF

Indicates that the file was profiled with the **fdpr** command.

0x0020 F_FDPR_OPTI

Indicates that the file was reordered with the **fdpr** command.

0x0040 F_DSA

Indicates that the file uses Very Large Program Support.

0x0080 Reserved.

0x0100 Reserved.

0x0200 Reserved.

0x0400 Reserved.

0x0800 Reserved.

0x1000 F_DYNLOAD

Indicates the file is dynamically loadable and executable. External references are resolved by way of imports, and the file might contain exports and loader relocation.

0x2000 F_SHROBJ

Indicates the file is a shared object (shared library). The file is separately loadable. That is, it is not normally bound with other objects, and its loader exports symbols are used as automatic import symbols for other object files.

0x4000 F_LOADONLY

If the object file is a member of an archive, it can be loaded by the system loader, but the member is ignored by the binder. If the object file is not in an archive, this flag has no effect.

0x8000 Reserved.

Auxiliary Header (aouthdr.h)

The auxiliary header contains system-dependent and implementation-dependent information, which is used for loading and executing a module. Information in the auxiliary header minimizes 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 are not required for an object file that is not to be loaded. When auxiliary headers are generated by compilers and assemblers, the headers are ignored by the binder.

The auxiliary header immediately follows the file header.

Note: If the value of the `f_opthdr` field is 0, the auxiliary header does not exist.

The C language structure for the auxiliary header is defined in the `authdr.h` file. The auxiliary header contains the fields shown in the following table.

Table 14. Auxiliary Header Structure (Defined in `authdr.h`)

Field Name and Description	XCOFF32	XCOFF64
o_mflag Flags, how to execute	<ul style="list-style-type: none"> • Offset: 0 • Length: 2 	<ul style="list-style-type: none"> • Offset: 0 • Length: 2
o_vstamp Version	<ul style="list-style-type: none"> • Offset: 2 • Length: 2 	<ul style="list-style-type: none"> • Offset: 2 • Length: 2
o_tsize⁺ Text size in bytes	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 56 • Length: 8
o_dsize⁺ Initialized data size in bytes	<ul style="list-style-type: none"> • Offset: 8 • Length: 4 	<ul style="list-style-type: none"> • Offset: 64 • Length: 8
o_bsize⁺ Uninitialized data size in bytes	<ul style="list-style-type: none"> • Offset: 12 • Length: 4 	<ul style="list-style-type: none"> • Offset: 72 • Length: 8
o_entry⁺ Entry point descriptor (virtual address)	<ul style="list-style-type: none"> • Offset: 16 • Length: 4 	<ul style="list-style-type: none"> • Offset: 80 • Length: 8
o_text_start⁺ Base address of text (virtual address)	<ul style="list-style-type: none"> • Offset: 20 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 8
o_data_start⁺ Base address of data (virtual address)	<ul style="list-style-type: none"> • Offset: 24 • Length: 4 	<ul style="list-style-type: none"> • Offset: 16 • Length: 8
o_toc⁺ Address of TOC anchor	<ul style="list-style-type: none"> • Offset: 28 • Length: 4 	<ul style="list-style-type: none"> • Offset: 24 • Length: 8
o_sentry Section number for entry point	<ul style="list-style-type: none"> • Offset: 32 • Length: 2 	<ul style="list-style-type: none"> • Offset: 32 • Length: 2
o_sntext Section number for <code>.text</code>	<ul style="list-style-type: none"> • Offset: 34 • Length: 2 	<ul style="list-style-type: none"> • Offset: 34 • Length: 2
o_sndata Section number for <code>.data</code>	<ul style="list-style-type: none"> • Offset: 36 • Length: 2 	<ul style="list-style-type: none"> • Offset: 36 • Length: 2
o_sntoc Section number for TOC	<ul style="list-style-type: none"> • Offset: 38 • Length: 2 	<ul style="list-style-type: none"> • Offset: 38 • Length: 2
o_snloader Section number for loader data	<ul style="list-style-type: none"> • Offset: 40 • Length: 2 	<ul style="list-style-type: none"> • Offset: 40 • Length: 2
o_snbss Section number for <code>.bss</code>	<ul style="list-style-type: none"> • Offset: 42 • Length: 2 	<ul style="list-style-type: none"> • Offset: 42 • Length: 2

Table 14. Auxiliary Header Structure (Defined in aouthdr.h) (continued)

Field Name and Description	XCOFF32	XCOFF64
o_algn_{text} Maximum alignment for .text	<ul style="list-style-type: none"> • Offset: 44 • Length: 2 	<ul style="list-style-type: none"> • Offset: 44 • Length: 2
o_algn_{data} Maximum alignment for .data	<ul style="list-style-type: none"> • Offset: 46 • Length: 2 	<ul style="list-style-type: none"> • Offset: 46 • Length: 2
o_mod_{type} Module type field	<ul style="list-style-type: none"> • Offset: 48 • Length: 2 	<ul style="list-style-type: none"> • Offset: 48 • Length: 2
o_cpuf_{lag} Bit flags - cpu types of objects	<ul style="list-style-type: none"> • Offset: 50 • Length: 1 	<ul style="list-style-type: none"> • Offset: 50 • Length: 1
o_cp_{utype} Reserved for CPU type	<ul style="list-style-type: none"> • Offset: 51 • Length: 1 	<ul style="list-style-type: none"> • Offset: 51 • Length: 1
o_max_{stack}⁺ Maximum stack size allowed (bytes)	<ul style="list-style-type: none"> • Offset: 52 • Length: 4 	<ul style="list-style-type: none"> • Offset: 88 • Length: 8
o_max_{data}⁺ Maximum data size allowed (bytes)	<ul style="list-style-type: none"> • Offset: 56 • Length: 4 	<ul style="list-style-type: none"> • Offset: 96 • Length: 8
o__{debugger}⁺ Reserved for debuggers.	<ul style="list-style-type: none"> • Offset: 60 • Length: 4 	<ul style="list-style-type: none"> • Offset: 4 • Length: 4
o__{resv2} Reserved Field must contain 0s.	<ul style="list-style-type: none"> • Offset: 64 • Length: 8 	<ul style="list-style-type: none"> • Offset: 52 • Length: 4
o__{resv3} Reserved. Field must contain 0s.	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A 	<ul style="list-style-type: none"> • Offset: 104 • Length: 116

+Use "32" or "64" suffix when `__XCOFF_HYBRID__` is defined.

Field Definitions: The following information defines the auxiliary header fields. For entries with two labels, the label in parentheses is the alternate original COFF **a.out** file format name.

<code>o_m_{flags}</code> (magic)	Specifies the magic number, which informs the operating system of the file's execution characteristics. The binder assigns the following value: 0x010B Text and data are aligned in the file and may be paged.
<code>o_v_{stamp}</code> (vstamp)	Specifies the format version for this auxiliary header. The only valid value is 1.
<code>o_t_{size}</code> (tsize)	Specifies 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 contained in the <code>s__{size}</code> field of the section header for the .text section.
<code>o_d_{size}</code> (dsize)	Specifies the size (in bytes) of the raw data for the .data section. The .data section contains the initialized data of the program and is writable. This is the same value as contained in the <code>s__{size}</code> field of the section header for the .data section.
<code>o_b_{size}</code> (bsize)	Specifies the size (in bytes) of .bss area, which is used for uninitialized variables during execution and is writable. No raw data exists in the file for the .bss section. This is the same value as contained in the <code>s__{size}</code> field of the section header for the .bss section.

<code>o_entry</code> (<code>entry</code>)	Specifies the virtual address of the entry point. (See the definition of the <code>o_snentry</code> field.) For application programs, this virtual address is the address of the function descriptor. The function descriptor contains the addresses of both the entry point itself and its TOC anchor. The offset of the entry point function descriptor from the beginning of its containing section can be calculated as follows: $\text{Section_offset_value} = \text{o_entry} - \text{s_paddr}[\text{o_snentry} - 1],$ where <code>s_paddr</code> is the virtual address contained in the section header.
<code>o_text_start</code> (<code>text_start</code>)	Specifies the virtual address of the <code>.text</code> section. This is the address assigned to (that is, used for) the first byte of the <code>.text</code> raw-data section. This is the same value as contained in the <code>s_paddr</code> field of the section header for the <code>.text</code> section.
<code>o_data_start</code> (<code>data_start</code>)	Specifies the virtual address of the <code>.data</code> section. This is the address assigned to (that is, used for) the first byte of the <code>.data</code> raw-data section. This is the same value as contained in the <code>s_paddr</code> field of the section header for the <code>.data</code> section.

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

The following definitions are extensions used by the system loader. In general, an object file may contain multiple sections of a given type, but in a module, only a single occurrence of the `.text`, `.data`, `.bss`, and `.loader` sections may exist.

<code>o_toc</code>	Specifies the virtual address of the TOC anchor (see the definition of the <code>o_sntoc</code> field).
<code>o_snentry</code>	Specifies the number of the file section containing the entry-point. (This field contains a file section header sequence number.) The entry point must be in the <code>.text</code> or <code>.data</code> section.
<code>o_sntext</code>	Specifies the number of the file <code>.text</code> section. (This field contains a file section header sequence number.)
<code>o_sndata</code>	Specifies the number of the file <code>.data</code> section. (This field contains a file section header sequence number.)
<code>o_sntoc</code>	Specifies the number of the file section containing the TOC. (This field contains a file section header sequence number.)
<code>o_snloader</code>	Specifies the number of the file section containing the system loader information. (This field contains a file section header sequence number.)
<code>o_snbss</code>	Specifies the number of the file <code>.bss</code> section. (This field contains a file section header sequence number.)
<code>o_algnstext</code>	Specifies the log (base 2) of the maximum alignment needed for any csect in the <code>.text</code> section.
<code>o_algnstata</code>	Specifies the log (base 2) of the maximum alignment needed for any csect in the <code>.data</code> and <code>.bss</code> sections.
<code>o_modtype</code>	Specifies a module type. The value is an ASCII character string. The following module type is recognized by the system loader: RO Specifies a read-only module. If a shared object with this module type has no BSS section and no dependents, the data section of the module will be mapped read-only and shared by all processes using the object.
<code>o_cpuflag</code>	Bit flags - <code>cputypes</code> of objects.
<code>o_cputype</code>	Reserved. This byte must be set to 0.
<code>o_maxstack</code>	Specifies the maximum stack size (in bytes) allowed for this executable. If the value is 0, the system default maximum stack size is used.
<code>o_maxdata</code>	Specifies the maximum data size (in bytes) allowed for this executable. If the value is 0, the system default maximum data size is used.
<code>o_debugger</code>	This field should contain 0. When a loaded program is being debugged, the memory image of this field may be modified by a debugger to insert a trap instruction.

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 in an XCOFF32 file is 40 bytes long, while XCOFF64 section headers are 72 bytes long. The C language structure for a section

header can be found in the **scnhdr.h** file. A section header contains the fields shown in the following table.

Table 15. Section Header Structure (Defined in *scnhdr.h*)

Field Name and Description	XCOFF32	XCOFF64
s_name Section name	<ul style="list-style-type: none"> • Offset: 0 • Length: 8 	<ul style="list-style-type: none"> • Offset: 0 • Length: 8
s_paddr⁺ Physical address	<ul style="list-style-type: none"> • Offset: 8 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 8
s_vaddr⁺ Virtual address (same as physical address)	<ul style="list-style-type: none"> • Offset: 12 • Length: 4 	<ul style="list-style-type: none"> • Offset: 16 • Length: 8
s_size⁺ Section size	<ul style="list-style-type: none"> • Offset: 16 • Length: 4 	<ul style="list-style-type: none"> • Offset: 24 • Length: 8
s_scnptr⁺ Offset in file to raw data for section	<ul style="list-style-type: none"> • Offset: 20 • Length: 4 	<ul style="list-style-type: none"> • Offset: 32 • Length: 8
s_relptr⁺ Offset in file to relocation entries for section	<ul style="list-style-type: none"> • Offset: 24 • Length: 4 	<ul style="list-style-type: none"> • Offset: 40 • Length: 8
s_lnnoptr⁺ Offset in file to line number entries for section	<ul style="list-style-type: none"> • Offset: 28 • Length: 4 	<ul style="list-style-type: none"> • Offset: 48 • Length: 8
s_nreloc⁺ Number of relocation entries	<ul style="list-style-type: none"> • Offset: 32 • Length: 2 	<ul style="list-style-type: none"> • Offset: 56 • Length: 4
s_nlnno⁺ Number of line number entries	<ul style="list-style-type: none"> • Offset: 34 • Length: 2 	<ul style="list-style-type: none"> • Offset: 60 • Length: 4
s_flags⁺ Flags to define the section type	<ul style="list-style-type: none"> • Offset: 36 • Length: 2 	<ul style="list-style-type: none"> • Offset: 64 • Length: 4
+Use "32" or "64" suffix when <code>__XCOFF_HYBRID__</code> is defined.		

Field Definitions: The following information defines the section header fields:

s_name	Specifies an 8-byte, null-padded section name. An 8-byte section name will not have a terminating null character. Use the <code>s_flags</code> field instead of the <code>s_name</code> 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	Specifies the physical address of the section. This is the address assigned and used by the compilers and the binder for the first byte of the section. This field should contain 0 for all sections except the <code>.text</code> , <code>.data</code> , and <code>.bss</code> sections.
s_vaddr	Specifies the virtual address of the section. This field has the same value as the <code>s_paddr</code> field.
s_size	Specifies the size (in bytes) of this section.
s_scnptr	Specifies 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 <code>s_size</code> field.

<code>s_relptr</code>	Specifies 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.
<code>s_lnnoptr</code>	Specifies 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.
<code>s_nreloc</code>	Specifies the number of relocation entries for this section. In an XCOFF32 file, if more than 65,534 relocation entries are required, the field value will be 65535, and an STYP_OVRFLO section header will contain the actual count of relocation entries in the <code>s_paddr</code> field. Refer to the discussion of overflow headers in "Sections and Section Headers" . If this field is set to 65535, the <code>s_nlnno</code> field must also be set to 65535.
<code>s_nlnno</code>	Specifies the number of line number entries for this section. In an XCOFF32 file, if more than 65,534 line number entries are required, the field value will be 65535, and an STYP_OVRFLO section header will contain the actual number of line number entries in the <code>s_vaddr</code> field. Refer to the discussion of overflow headers in "Sections and Section Headers" . If this field is set to 65535, the <code>s_nreloc</code> field must also be set to 65535.
<code>s_flags</code>	Specifies flags defining the section type. The low-order pair of bytes is used. A section type identifies the contents of a section and specifies how the section is to be processed by the binder or the system loader. Only a single bit value may be assigned to the <code>s_flags</code> field. This value must not be the sum or bitwise OR of multiple flags. The two high-order bytes should contain 0.

Valid bit values are:

Value	Flag
-------	------

0x0000	Reserved.
---------------	-----------

0x0001	Reserved.
---------------	-----------

0x0002	Reserved.
---------------	-----------

0x0004	Reserved.
---------------	-----------

0x0008	STYP_PAD
---------------	-----------------

Specifies a pad section. A section of this type is used to provide alignment padding between sections within an XCOFF executable object file. This section header type is obsolete since padding is allowed in an XCOFF file without a corresponding pad section header.

0x0010	Reserved.
---------------	-----------

0x0020	STYP_TEXT
---------------	------------------

Specifies an executable text (code) section. A section of this type contains the executable instructions of a program.

0x0040	STYP_DATA
---------------	------------------

Specifies an initialized data section. A section of this type contains the initialized data and the TOC of a program.

0x0080	STYP_BSS
---------------	-----------------

Specifies an uninitialized data section. A section header of this type defines the uninitialized data of a program.

0x0100	STYP_EXCEPT
---------------	--------------------

Specifies an exception section. A section of this type provides information to identify the reason that a trap or exception occurred within an executable object program.

0x0200	STYP_INFO
---------------	------------------

Specifies a comment section. A section of this type provides comments or data to special processing utility programs.

0x0400	Reserved.
---------------	-----------

0x0800	Reserved.
---------------	-----------

s_flags
continued

Valid bit values are:

Value **Flag**

0x1000 STYP_LOADER

Specifies a loader section. A section of this type contains object file information for the system loader to load an XCOFF executable. The information includes imported symbols, exported symbols, relocation data, type-check information, and shared object names.

0x2000 STYP_DEBUG

Specifies a debug section. A section of this type contains stabstring information used by the symbolic debugger.

0x4000 STYP_TYPCHK

Specifies a type-check section. A section of this type contains parameter/argument type-check strings used by the binder.

0x8000 STYP_OVRFLO

Note: An XCOFF64 file may not contain an overflow section header.

Specifies a 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 when either of the counts exceeds 65,534. See the s_nreloc and s_nlnno fields in "Sections and Section Headers" for more information on overflow headers.

For general information on the XCOFF file format, see "XCOFF Object File Format."

Sections and Section Headers

Section headers are defined to provide a variety of information about the contents of an XCOFF file. Programs that process XCOFF files will recognize only some of the valid sections.

See the following information to learn more about XCOFF file sections:

- Loader Section (loader.h)
- Debug Section
- Type-Check Section
- Exception Section
- Comment Section

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 the following table.

Table 16. Conventional Header Names

Description	Multiple Allowed?	s_flag (and its conventional name)
Text section	Yes	STYP_TEXT (.text)
Data section	Yes	STYP_DATA (.data)
BSS section	Yes	STYP_BSS (.bss)
Pad section	Yes	STYP_PAD (.pad)
Loader section	No	STYP_LOADER (.loader)
Debug section	No	STYP_DEBUG (.debug)
Type-check section	Yes	STYP_TYPCHK (.typchk)
Exception section	No	STYP_EXCEPT (.except)
Overflow section	Yes (one per .text or .data section)	STYP_OVRFLO (.ovrflo)

Table 16. Conventional Header Names (continued)

Description	Multiple Allowed?	s_flag (and its conventional name)
Comment section	Yes	STYP_INFO (.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 (shown in the "Conventional Header Names" table) are used.
s_scnptr	Ignored for .bss sections.
s_relptr	Recognized for the .text and .data sections only. No relocation is performed for other sections, where this value must be 0.
s_lnnoptr	Recognized for the .text section only. Otherwise, it must be 0.
s_nreloc, s_nlnno	Handles relocation or line-number field overflows in an XCOFF32 file. (XCOFF64 files may not have overflow section headers.) If a section has more than 65,534 relocation entries or line number entries, both of these fields are set to a value of 65535. In this case, an overflow section header with the s_flags field equal to STYP_OVRFLO is used to contain the relocation and line-number count information. The fields in the overflow section header are defined as follows:

s_nreloc

Specifies the file section number of the section header that overflowed; that is, the section header containing a value of 65535 in its s_nreloc and s_nlnno fields. This value provides a reference to the primary section header. This field must have the same value as the s_nlnno field.

Note: 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.

s_nlnno

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

s_paddr

Specifies 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

Specifies 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 have a value of 0 in an overflow section header. The s_relptr and s_lnnoptr fields must have the same values as in the corresponding primary section header.

An XCOFF file provides special meaning to the following sections:

- The .text, .data, and .bss sections define 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. Padding is allowed in an XCOFF file without a corresponding section header.
- 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 stab (symbol table) 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 comments or data that are of significance to special processing utility programs.
- The `.debug`, `.except`, `.info`, and `.typchk` sections are produced by compilers and assemblers. References to these sections or to items within these sections are made from the XCOFF Symbol Table.

For more information on XCOFF file sections, see "Loader Section (loader.h)," "Debug Section," "Type-Check Section," "Exception Section," and "Comment Section."

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 is not referenced by 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 **loader.h** file.

Loader Header Field Definitions

The following table describes the loader section's header field definitions.

Table 17. Loader Section Header Structure (Defined in loader.h)

Field Name and Description	XCOFF32	XCOFF64
l_version Loader section version number	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 4
l_nsyms Number of symbol table entries	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 4 • Length: 4
l_nreloc Number of relocation table entries	<ul style="list-style-type: none"> • Offset: 8 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 4
l_istlen Length of import file ID string table	<ul style="list-style-type: none"> • Offset: 12 • Length: 4 	<ul style="list-style-type: none"> • Offset: 12 • Length: 4
l_nimpid Number of import file IDs	<ul style="list-style-type: none"> • Offset: 16 • Length: 4 	<ul style="list-style-type: none"> • Offset: 16 • Length: 4

Table 17. Loader Section Header Structure (Defined in loader.h) (continued)

Field Name and Description	XCOFF32	XCOFF64
l_impoff⁺ Offset to start of import file IDs	<ul style="list-style-type: none"> • Offset: 20 • Length: 4 	<ul style="list-style-type: none"> • Offset: 24 • Length: 8
l_stlen⁺ Length of string table	<ul style="list-style-type: none"> • Offset: 24 • Length: 4 	<ul style="list-style-type: none"> • Offset: 20 • Length: 4
l_stoff⁺ Offset to start of string table	<ul style="list-style-type: none"> • Offset: 28 • Length: 4 	<ul style="list-style-type: none"> • Offset: 32 • Length: 8
l_symoff Offset to start of symbol table	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A 	<ul style="list-style-type: none"> • Offset: 40 • Length: 8
l_rldoff Offset to start of relocation entries	<ul style="list-style-type: none"> • Offset: 36 • Length: 2 	<ul style="list-style-type: none"> • Offset: 64 • Length: 4
+Use "32" or "64" suffix when <code>__XCOFF_HYBRID__</code> is defined.		

The following information defines the loader section's header fields:

<code>l_version</code>	Specifies the loader section version number. This value must be 1 for XCOFF32, 2 for XCOFF64.
<code>l_nsyms</code>	Specifies 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 <code>.text</code> , <code>.data</code> , and <code>.bss</code> symbol entries.
<code>l_nreloc</code>	Specifies the number of relocation table entries in the loader section.
<code>l_istlen</code>	Specifies the byte length of the import file ID string table in the loader section.
<code>l_nimpid</code>	Specifies the number of import file IDs in the import file ID string table.
<code>l_impoff</code>	Specifies the byte offset from beginning of the loader section to the first import file ID.
<code>l_stlen</code>	Specifies the length of the loader section string table.
<code>l_stoff</code>	Specifies the byte offset from beginning of the loader section to the first entry in the string table.
<code>l_symoff</code>	Specifies the byte offset from beginning of the loader section to the start of the loader symbol table (in XCOFF64 only).
<code>l_rldoff</code>	Specifies the byte offset from beginning of the loader section to the start of the loader section relocation entries (in XCOFF64 only).

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`, `.data`, and `.bss` sections. These symbols 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 18. Loader Section Symbol Table Entry Structure

Field Name and Description	XCOFF32	XCOFF64
l_name⁺ Symbol name or byte offset into string table	<ul style="list-style-type: none"> • Offset: 0 • Length: 8 	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A

Table 18. Loader Section Symbol Table Entry Structure (continued)

Field Name and Description	XCOFF32	XCOFF64
l_zeroes⁺ Zero indicates symbol name is referenced from l_offset	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A
l_offset⁺ Byte offset into string table of symbol name	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 4
l_value⁺ Address field	<ul style="list-style-type: none"> • Offset: 8 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 8
l_scnm Section number containing symbol	<ul style="list-style-type: none"> • Offset: 12 • Length: 2 	<ul style="list-style-type: none"> • Offset: 12 • Length: 2
l_smtpe Symbol type, export, import flags	<ul style="list-style-type: none"> • Offset: 14 • Length: 1 	<ul style="list-style-type: none"> • Offset: 14 • Length: 1
l_smclas Symbol storage class	<ul style="list-style-type: none"> • Offset: 15 • Length: 1 	<ul style="list-style-type: none"> • Offset: 15 • Length: 1
l_ifile Import file ID; ordinal of import file IDs	<ul style="list-style-type: none"> • Offset: 16 • Length: 4 	<ul style="list-style-type: none"> • Offset: 16 • Length: 4
l_parm Parameter type-check field	<ul style="list-style-type: none"> • Offset: 20 • Length: 4 	<ul style="list-style-type: none"> • Offset: 20 • Length: 4
+Use "32" or "64" suffix when <code>__XCOFF_HYBRID__</code> is defined.		

The symbol table fields are:

- l_name** (XCOFF32 only) Specifies an 8-byte, null-padded symbol name if it is 8 bytes or less in length. Otherwise, the field is treated as the following two 4-byte integers for accessing the symbol name:
- l_zeroes** (XCOFF32 only) 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 having the first 4 bytes (first word) equal to 0 is used to indicate that the name string is contained in the string table instead of the l_name field.
 - l_offset** (XCOFF32 only) This field overlays the second word of the l_name field. The value of this field is the byte offset from the beginning of the loader section string table to the first byte of the symbol name (not its length field).
- l_offset** (XCOFF64 only) This field has the same use as the l_offset field in XCOFF32.
- l_value** Specifies the virtual address of the symbol
- l_scnm** Specifies 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. Section headers are numbered beginning with 1.

`l_smtype` Specifies the symbol type, import flag, export flag, and entry flag.

Bits 0-4 are flag bits defined as follows:

- Bit 0** 0x80 Reserved.
- Bit 1** 0x40 Specifies an imported symbol.
- Bit 2** 0x20 Specifies an entry point descriptor symbol.
- Bit 3** 0x10 Specifies an exported symbol.
- Bit 4** 0x08 Specifies a weak symbol.
- Bits 5-7** 0x07 Symbol type--see below.

Bits 5-7 constitute a 3-bit symbol type field with the following definitions:

- 0** XTY_ER
Specifies an external reference providing a symbol table entry for an external (global) symbol contained in another XCOFF object file.
- 1** XTY_SD
Specifies the csect section definition, providing the definition of the smallest initialized unit within an XCOFF object file.
- 2** XTY_LD
Specifies the label definition, providing the 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, providing the definition of the smallest uninitialized unit within an XCOFF object file.
- 4-7** Reserved.

`l_smc1as` Specifies the storage mapping class of the symbol, as defined in **syms.h** for the `x_smc1as` field of the csect auxiliary symbol table entry. Values have the symbolic form `XMC_xx`, where `xx` is PR, RO, GL, XO, SV, SV64, SV3264, RW, TC, TD, DS, UA, BS, or UC. See "csect Auxiliary Entry for the **C_EXT**, **C_WEAKEXT**, and **C_HIDEXT** Symbols" for more information.

`l_ifile` Specifies the import file ID string. This integer is the ordinal value of the position of the import file ID string in the import file ID name string table of the loader section. For an imported symbol, the value of 0 in this field identifies the symbol as a deferred import to the system loader. A deferred import is a symbol whose address can remain unresolved following the processing of the loader. If the symbol was not imported, this field must have a value of 0.

`l_parm` Specifies the 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 byte of the parameter type-check string (not to its length field). For more information on the parameter type-check string, see "Type-Check Section". 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.

Loader Relocation Table Field Definitions

The Loader Section Relocation Table Structure 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 in XCOFF32 and 16 bytes long in XCOFF64. The `l_vaddr`, `l_symndx`, and `l_rtype` fields have the same meaning as the corresponding fields of the regular relocation entries, which are defined in the **reloc.h** file. See "Relocation Information for XCOFF File (reloc.h)" for more information.

Table 19. Loader Section Relocation Table Entry Structure

Field Name and Description	XCOFF32	XCOFF64
<code>l_vaddr*</code> Address field	<ul style="list-style-type: none">• Offset: 0• Length: 4	<ul style="list-style-type: none">• Offset: 0• Length: 8

Table 19. Loader Section Relocation Table Entry Structure (continued)

Field Name and Description	XCOFF32	XCOFF64
l_symndx⁺ Loader section symbol table index of referenced item	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 12 • Length: 4
l_rtype Relocation type	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 4
l_value⁺ Address field	<ul style="list-style-type: none"> • Offset: 8 • Length: 2 	<ul style="list-style-type: none"> • Offset: 8 • Length: 2
l_rsecnm File section number being relocated	<ul style="list-style-type: none"> • Offset: 10 • Length: 2 	<ul style="list-style-type: none"> • Offset: 10 • Length: 2

+Use "32" or "64" suffix when `__XCOFF_HYBRID__` is defined.

The `loader.h` file defines the following fields:

Name	Description
<code>l_vaddr</code>	Specifies the virtual address of the relocatable reference.
<code>l_symndx</code>	Specifies the loader section symbol table index (<i>n</i> -th entry) of the symbol that is being referenced. Values 0, 1, and 2 are implicit references to the <code>.text</code> , <code>.data</code> , and <code>.bss</code> sections, respectively. Symbol index 3 is the index for the first symbol actually contained in the loader section symbol table. Note: A reference to an exported symbol can be made using the symbol's section number (symbol number 0, 1, or 2) or using the actual number of the exported symbol.
<code>l_rtype</code>	Specifies the relocation size and type. (This field has the same interpretation as the <code>r_type</code> field in the <code>reloc.h</code> file.) See "Relocation Information for XCOFF File (<code>reloc.h</code>)" for more information.
<code>l_rsecnm</code>	Specifies the section number of the <code>.text</code> , <code>.data</code> , or <code>.bss</code> section being relocated (associated with <code>l_vaddr</code> field). This is a one-based index into the section headers.

Loader Import File ID Name Table Definition

The loader section import file ID name strings of a module provide a list of dependent modules that the system loader must load in order for the module to load successfully. However, this list does not contain the names of modules that the named modules themselves depend on.

Table 20. Loader Section Import File IDs - Contains Variable Length Strings

Offset	Length in Bytes	Name and Description
0	<i>n1</i>	l_impidpath Import file ID path string, null-delimited
<i>n1</i> + 1	<i>n2</i>	l_impidbase Import file ID base string, null-delimited
<i>n1</i> + <i>n2</i> + 2	<i>n3</i>	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.

The first import file ID is a default **LIBPATH** value to be used by the system loader. 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:

- Import file ID path name
- Null delimiter (ASCII Null Character)
- Import file ID base name
- Null delimiter
- Import file ID archive-file-member name
- Null delimiter

For example:

```
/usr/lib\0mylib.a\0shr.o\0
```

Loader String Table Definition

The loader section string table contains the parameter type-checking strings, all symbols names for an XCOFF64 file, and the names of symbols longer than 8 bytes for an XCOFF32 file. Each string consists of a 2-byte length field followed by the string.

Table 21. Loader Section String Table

Offset	Length in Bytes	Description
0	2	Length of string.
2	n	Symbol name string (null-delimited) or parameter type string (not null-delimited).
		Fields repeat for each string.

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

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 but does not include the length of the length field itself.

The symbol table entries of the loader section contain a byte offset value that points to the first byte of the string instead of to the length field.

Loader Section Header Contents

The contents of the section header fields for the loader section are:

Name	Contents
s_name	.loader
s_paddr	0
s_vaddr	0
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	0
s_lnnoptr	0
s_nreloc	0
s_nlnno	0
s_flags	STYP_LOADER

For general information on the XCOFF file format, see "XCOFF Object File Format."

For more information on XCOFF file sections, see "Sections and Section Headers," "Debug Section," "Type-Check Section," "Exception Section," and "Comment Section."

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 2-byte length field in XCOFF32 or a 4-byte length field in XCOFF64.

Field Definitions

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

- A 2-byte (XCOFF32) or 4-byte (XCOFF64) length field containing the length of the string. The value contained in the length field includes the length of the terminating null character but does not include the length of the length field itself.
- The symbolic debugger stabstring.

Refer to discussion of symbolic debugger stabstring grammar for the specific format of the stabstrings.

Debug Section Header Contents

The contents of the section header fields for the debug section are:

Name	Contents
<code>s_name</code>	<code>.debug</code>
<code>s_paddr</code>	0
<code>s_vaddr</code>	0
<code>s_size</code>	The size (in bytes) of the debug section
<code>s_scnptr</code>	Offset from the beginning of the XCOFF file to the first byte of the debug section data
<code>s_relptr</code>	0
<code>s_lmnoptr</code>	0
<code>s_nreloc</code>	0
<code>s_nlnno</code>	0
<code>s_flags</code>	STYP_DEBUG

For general information on the **XCOFF** file format, see "XCOFF Object File Format."

For more information on **XCOFF** file sections, see "Sections and Section Headers," "Debug Section," "Type-Check Section," "Exception Section," and "Comment Section."

Type-Check Section

The type-check section contains the type-checking hash strings and is produced by compilers and assemblers. It is used by the binder to detect variable mismatches and argument interface errors when linking separately compiled object files. (The type-checking hash strings in the loader section are used to detect these errors prior to running a program.) 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 strings in this section are referenced from entries in the **XCOFF** symbol table.

Field Definitions

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

- A 2-byte length 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.

Type Encoding and Checking Format for Data

The type-checking hash strings are used to detect errors prior to execution of a program. Information about all external symbols (data and functions) is encoded by the compilers and then checked for

consistency at bind time and load time. The type-checking strings are designed to enforce the maximum checking required by the semantics of each particular language supported, as well as provide protection to applications written in more than one language.

The type encoding and checking mechanism features 4-part hash encoding that provides some flexibility in checking. The mechanism also uses a unique value, UNIVERSAL, that matches any code. The UNIVERSAL hash can be used as an escape mechanism for assembly programs or for programs in which type information or subroutine interfaces might not be known. The UNIVERSAL hash is four blank ASCII characters (0x20202020) or four null characters (0x00000000).

The following fields are associated with the type encoding and checking mechanism:

code length	A 2-byte field containing the length of the hash. This field has a value of 10.
language identifier	A 2-byte code representing each language. These codes are the same as those defined for the <code>e_lang</code> field in the "Exception Section" information .
general hash	A 4-byte field representing the most general form by which a data symbol or function can be described. This form is the most common to languages supported by . If the information is incomplete or unavailable, a universal hash should be generated. The general hash is language-independent and must match for the binding to succeed.
language hash	A 4-byte field containing a more detailed, language-specific representation of what is in the general hash. It allows for the strictest type-checking required by a given language. This part is used in intra-language binding and is not checked unless both symbols have the same language identifier.

Section Header Contents

The contents of the section header fields for the type-check section are:

Name	Contents
<code>s_name</code>	.typchk
<code>s_paddr</code>	0
<code>s_vaddr</code>	0
<code>s_size</code>	The size (in bytes) of the type-check section
<code>s_scnptr</code>	Offset from the beginning of the XCOFF file to the first byte of the type-check section data
<code>s_relptr</code>	0
<code>s_lnopttr</code>	0
<code>s_nreloc</code>	0
<code>s_nlnno</code>	0
<code>s_flags</code>	STYP_TYPCHK.

For general information on the **XCOFF** file format, see "XCOFF Object File Format."

For more information on **XCOFF** file sections, see "Sections and Section Headers," "Debug Section," "Type-Check Section," "Exception Section," and "Comment Section."

Exception Section

The exception section contains addresses of trap instructions, source language identification codes, and trap reason codes. This section is produced by compilers and assemblers, and used during or after run time to identify the reason that a specific trap or exception occurred. 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 value of 0 in the `e_reason` field contains the symbol table index to a function's **C_EXT**, **C_WEAKEXT**, or **C_HIDEXT** symbol table entry. Reference from the symbol table to an entry in the exception table is via the function auxiliary symbol table entry. For more information on this entry, see "csect Auxiliary Entry for **C_EXT**, **C_WEAKEXT** and **C_HIDEXT** Symbols."

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

The exception section entries contain the fields shown in the following tables.

Table 22. Initial Entry: Exception Section Structure

Field Name and Description	XCOFF32	XCOFF64
e_addr.e_symndx⁺ Symbol table index for function	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 4
e_lang⁺ Compiler language ID code	<ul style="list-style-type: none"> • Offset: 4 • Length: 1 	<ul style="list-style-type: none"> • Offset: 8 • Length: 1
e_reason⁺ Value 0 (exception reason code 0)	<ul style="list-style-type: none"> • Offset: 5 • Length: 1 	<ul style="list-style-type: none"> • Offset: 9 • Length: 1
+Use "32" or "64" suffix when __XCOFF_HYBRID__ is defined. With e_addr.e_symndx, the suffix is added to e_addr (i.e. e_addr32.e_symndx).		

Table 23. Subsequent Entry: Exception Section Structure

Field Name and Description	XCOFF32	XCOFF64
e_addr.e_paddr⁺ Address of the trap instruction	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 8
e_lang⁺ Compiler language ID code	<ul style="list-style-type: none"> • Offset: 4 • Length: 1 	<ul style="list-style-type: none"> • Offset: 8 • Length: 1
e_reason⁺ Trap exception reason code	<ul style="list-style-type: none"> • Offset: 5 • Length: 1 	<ul style="list-style-type: none"> • Offset: 9 • Length: 1
+Use "32" or "64" suffix when __XCOFF_HYBRID__ is defined. With e_addr.e_paddr, the suffix is added to e_addr (i.e. e_addr32.e_paddr).		

Field Definitions

The following defines the fields listed of the exception section:

- e_symndx Contains an integer (overlays the e_paddr field). When the e_reason field is 0, this field is the symbol table index of the function.
- e_paddr Contains a virtual address (overlays the e_symndx field). When the e_reason field is nonzero, this field is the virtual address of the trap instruction.

e_lang Specifies the source language. The following list defines the possible values of the e_lang field.

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 Specifies an 8-bit, compiler-dependent trap exception reason code. Zero is not a valid trap exception reason code because it indicates the start of exception table entries for a new function.

Section Header Contents

The following fields are the contents of the section header fields for the exception section.

Name	Contents
s_name	.except
s_paddr	0
s_vaddr	0
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	0
s_lnnoptr	0
s_nreloc	0
s_nlnno	0
s_flags	STYP_EXCEPT

For general information on the **XCOFF** file format, see "XCOFF Object File Format."

For more information on **XCOFF** file sections, see "Sections and Section Headers," "Debug Section," "Type-Check Section," "Exception Section," and "Comment Section."

Comment Section

The comment section contains information of special processing significance to an application. This section can be produced by compilers and assemblers and used during or after run time to fulfill a special processing need of an application. The comment section has a section type flag of **STYP_INFO** in the **XCOFF** section header. By convention, `.info` is the comment section name. Data in the comment section is referenced from **C_INFO** entries in the **XCOFF** symbol table.

The contents of a comment section consists of repeated instances of a 4-byte length field followed by a string of bytes (containing any binary value). The length of each string is stored in its preceding 4-byte length field. The string of bytes need not be terminated by a null character nor by any other special character. The specified length does not include the length of the length field itself. A length of 0 is allowed. The format of the string of bytes is not specified.

A comment section string is referenced from an entry in the **XCOFF** symbol table. The storage class of the symbol making a reference is **C_INFO**. See "Symbol Table Field Contents by Storage Class" for more information.

A **C_INFO** symbol is associated with the nearest **C_FILE**, **C_EXT**, **C_WEAKEXT**, or **C_HIDEXT** symbol preceding it.

Section Header Contents

The following fields are the contents of the section header fields for the comment section.

Name	Contents
s_name	.info
s_paddr	0
s_vaddr	0
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	0
s_lnnoptr	0
s_nreloc	0
s_nlnno	0
s_flags	STYP_INFO

For general information on the **XCOFF** file format, see "XCOFF Object File Format."

For more information on XCOFF file sections, see "Sections and Section Headers," "Debug Section," "Type-Check Section," "Exception Section," and "Comment Section."

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 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 (14 for XCOFF64). (A relocation entry in the .loader section is 12 bytes long (16 for XCOFF64) and is explained in the loader section description in this document. See "Relocation Table Field Definitions" for more information.) The C language structure for a relocation entry can be found in the **reloc.h** file. A relocation entry contains the fields shown in the following table.

Table 24. Relocation Entry Structure

Field Name and Description	XCOFF32	XCOFF64
r_vaddr* Virtual address (position) in section to be relocated	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 8

Table 24. Relocation Entry Structure (continued)

Field Name and Description	XCOFF32	XCOFF64
r_symndx ⁺ Symbol table index of item that is referenced	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 4
r_rsize ⁺ Relocation size and information	<ul style="list-style-type: none"> • Offset: 8 • Length: 1 	<ul style="list-style-type: none"> • Offset: 12 • Length: 1
r_rtype ⁺ Relocation type	<ul style="list-style-type: none"> • Offset: 9 • Length: 1 	<ul style="list-style-type: none"> • Offset: 13 • Length: 1
+Use "32" or "64" suffix when <code>__XCOFF_HYBRID__</code> is defined.		

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, so a section number is required within each relocation entry to identify the section that needs to be modified.)

Field Definitions

The following defines the relocation-information fields:

<code>r_vaddr</code>	Specifies the virtual address of the value that requires modification 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: $\text{offset_in_section} = \text{r_vaddr} - \text{s_paddr}$
<code>r_symndx</code>	Specifies a zero-based index into the XCOFF symbol table for locating the referenced symbol. The symbol table entry contains an address used to calculate a modification value to be applied at the <code>r_vaddr</code> relocation address.
<code>r_rsize</code>	Specifies the relocation size and sign. Its contents are detailed in the following list: 0x80 (1 bit) Indicates whether the relocation reference is signed (1) or unsigned (0). 0x40 (1 bit) If this field is one, it indicates that the binder replaced the original instruction by a branch instruction to a special fixup instruction sequence. 0x3F(6 bits) Specifies the bit length of the relocatable reference minus one. The current architecture allows for fields of up to 32 bits (XCOFF32) or 64 bits (XCOFF64) to be relocated.

r_rtype

Specifies an 8-bit relocation type field that indicates to the binder which relocation algorithm to use for calculating the modification value. This value is applied at the relocatable reference location specified by the r_vaddr field. The following relocation types are defined:

0x00 R_POS

Specifies positive relocation. Provides the address of the symbol specified by the r_symndx field.

0x01 R_NEG

Specifies negative relocation. Provides the negative of the address of the symbol specified by the r_symndx field.

0x02 R_REL

Specifies relative-to-self relocation. Provides a displacement value between the address of the symbol specified by the r_symndx field and the address of the csect to be modified.

0x03 R_TOC

Specifies relative-to-TOC relocation. Provides a displacement value that is the difference between the address value in the symbol specified by the r_symndx field and the address of the TOC anchor csect. The TOC anchor csect has a symbol table csect auxiliary entry with an x_smc|ass (storage mapping class) value of **XMC_TOC**. The TOC anchor csect must be of zero length. There may be only one TOC anchor csect per XCOFF section.

0x04 R_TRL

Specifies TOC Relative Indirect Load (modifiable) relocation. Provides a displacement value that is the difference between the address value in the symbol specified by the r_symndx field 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 information concerning the instruction being relocated: The instruction that is referenced by the r_vaddr field is a load instruction. That load instruction is permitted to be modified by the binder to become a compute address instruction. Changing an instruction from a load instruction to a compute address instruction avoids a storage reference during execution. A compute address instruction can be used if the address contained at the address specified by the r_symndx field has a value that itself references a r_symndx field that can be accessed with a valid in-range displacement relative to the TOC anchor address. That is, the target of the TOC entry is from -32,768 to 32,767, inclusive, from the TOC anchor address. If a compute address instruction is generated by the binder, the **R_TRL** relocation type is changed to become a **R_TRLA** type. This allows the reverse transformation, if required. Compilers are permitted to generate this relocation type.

0x13 R_TRLA

Specifies TOC Relative Load Address (modifiable LA to L) relocation. Provides a displacement value that is the difference between the address value in the symbol specified by the r_symndx field 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 information concerning the instruction being relocated: The instruction that is referenced by the r_vaddr field is a compute address instruction. The compute address instruction is 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 into 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 are not permitted to generate this relocation type.

r_rtype continued

0x05	R_GL	Specifies Global Linkage-External TOC address relocation. Provides the address of the TOC associated with a defined external symbol. The external symbol with the required TOC address is specified by the <code>r_symndx</code> field of the relocation entry. This relocation entry provides a method of accessing the address of the TOC contained within the same executable where the <code>r_symndx</code> external symbol is defined.
0x06	R_TCL	Specifies local object TOC address relocation. 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 <code>r_symndx</code> field of the relocation entry. 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 where the <code>r_symndx</code> external symbol is defined.
0x0C	R_RL	Treated the same as the R_POS relocation type.
0x0D	R_RLA	Treated the same as the R_POS relocation type.
0x0F	R_REF	Specifies a nonrelocating 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	Treated the same as the R_RBA relocation type.
0x18	R_RBA	Specifies branch absolute relocation. Provides the address of the symbol specified by the <code>r_symndx</code> field as the target address of a branch instruction. The instruction can be modified to a (relative) branch instruction if the target address is relocatable.
0x0A	R_BR	Treated the same as the R_RBR relocation type.
0x1A	R_RBR	Specifies (relative) branch relocation. Provides a displacement value between the address of the symbol specified by the <code>r_symndx</code> field and the address of the csect containing the branch instruction to be modified. The instruction can be modified to an absolute branch instruction if the target address is not relocatable. The R_RBR relocation type is the standard branch relocation type used by compilers and assemblers for the . This relocation type along with glink code allows an executable object file to have a text section that is position-independent.

Additional Relocation Features

Standard 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. To provide the capability for rebinding and handling a relocatable address space model, the relocation information is not discarded from an **XCOFF** file.

For general information on the **XCOFF** file format, see "XCOFF Object File Format."

For more information on relocation field table definitions, see "Relocation Table Field Definitions" in the loader section.

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 a single 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 the `l_1nno` field containing a value of 0. The first field, `l_symndx`, is the symbol table index to the **C_EXT**, **C_WEAKEXT**, 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 **linenum.h** file. A line number entry contains the fields shown in the following tables.

Table 25. Initial Line Number Structure Entry for Function

Field Name and Description	XCOFF32	XCOFF64
<code>l_addr.l_symndx</code> ⁺ Symbol table index for function	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 4
<code>l_1nno</code> ⁺ Value 0 (line number 0)	<ul style="list-style-type: none"> • Offset: 4 • Length: 2 	<ul style="list-style-type: none"> • Offset: 8 • Length: 4
+Use "32" or "64" suffix when __XCOFF_HYBRID__ is defined. With <code>l_addr.l_symndx</code> , the suffix is added to <code>l_addr</code> (i.e. <code>l_addr32.l_symndx</code>).		

Table 26. Subsequent Line Number Entries for Function

Field Name and Description	XCOFF32	XCOFF64
<code>l_paddr</code> ⁺ Address at which break point can be inserted	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 8
<code>l_1nno</code> ⁺ Line number relative to start of function	<ul style="list-style-type: none"> • Offset: 4 • Length: 2 	<ul style="list-style-type: none"> • Offset: 8 • Length: 4
+Use "32" or "64" suffix when __XCOFF_HYBRID__ is defined. With <code>l_addr.l_paddr</code> , the suffix is added to <code>l_addr</code> (i.e. <code>l_addr32.l_paddr</code>).		

Field Definitions

The following list defines the line number entries:

<code>l_symndx</code>	Specifies the symbol table index to the function name (overlays the <code>l_paddr</code> field). When the <code>l_1nno</code> field is 0, this interpretation of the field is used.
<code>l_paddr</code>	Specifies the virtual address of the first instruction of the code associated with the line number (overlays the <code>l_symndx</code> field). When the <code>l_1nno</code> field is not 0, this interpretation of the field is used.
<code>l_1nno</code>	Specifies either the line number relative to the start of a function or 0 to indicate the beginning of a function.

Note: If part of a function other than the beginning comes from an include file, the line numbers are absolute, rather than relative to the beginning of the function. (See the **C_BINCL** and **C_EINCL** symbol types in "Storage Classes by Usage and Symbol Value Classification" for more information.)

For general information on the **XCOFF** file format, see "XCOFF Object File Format."

For information on debugging, see "Debug Section."

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 18-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.

See the following information to learn more about the symbol table:

- Symbol Table Auxiliary Information
- Symbol Table Field Contents by Storage Class
- String Table

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, performing the following functions:

- Define replaceable units or csects.
- Define the external names for functions or entry points within csects.
- Reference the names of external functions in another **XCOFF** object.

For symbols defining a replaceable unit (csect), a csect auxiliary entry defines the length and storage-mapping class of the csect. For symbols defining external names for functions within a csect, the csect auxiliary entry points to the containing csect, the parameter type-checking information, and the 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.

Symbol Table Contents

An XCOFF symbol table has the following general contents and ordering:

- The **C_FILE** symbol table entries used to bracket all the symbol table entries associated with a given source file.
- The **C_INFO** comment section symbol table entries that are of source file scope. These follow the **C_FILE** entry but before the first csect definition symbol table entry.
- The symbolic debugger symbol table entries that are of file scope. These follow the **C_FILE** entry but before the first csect entry.
- csect definition symbol table entries used to define and bracket all the symbols contained with a csect.
- **C_INFO** comment section symbol table entries that follow a csect definition symbol table entry are associated with that csect.
- All symbolic debugger symbol table entries that follow a csect definition symbol table entry or label symbol table entry are associated with that csect or label.

The ordering of the symbol table must be arranged by the compilers and assemblers both to 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 binding, 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.

Symbol Table Layout

The following example shows the general ordering of the symbol table.

```

un_external      Undefined global symbols

.file           Prolog --defines stabstring compaction level
.file           Source file 1
.info           Comment section reference symbol with file scope
stab            Global Debug symbols of a file
csect          Replaceable unit definition (code)
.info           Comment section reference symbol with csect scope
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
.....

```

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 **syms.h** file. The index for the first entry in the symbol table is 0. The following table shows the structure of the fixed-format part of each symbol in the symbol table.

Table 27. Symbol Table Entry Format

Field Name and Description	XCOFF32	XCOFF64
n_name Symbol name (occupies the same 8 bytes as n_zeroes and n_offset)	<ul style="list-style-type: none"> Offset: 0 Length: 8 	<ul style="list-style-type: none"> Offset: N/A Length: N/A
n_zeroes Zero, indicating name in string table or .debug section (overlays first 4 bytes of n_name)	<ul style="list-style-type: none"> Offset: 0 Length: 4 	<ul style="list-style-type: none"> Offset: N/A Length: N/A
n_offset⁺ Offset of the name in string table or .debug section (In XCOFF32: overlays last 4 bytes of n_name)	<ul style="list-style-type: none"> Offset: 4 Length: 4 	<ul style="list-style-type: none"> Offset: 8 Length: 4

Table 27. Symbol Table Entry Format (continued)

Field Name and Description	XCOFF32	XCOFF64
n_value+ Symbol value; storage class-dependent	<ul style="list-style-type: none"> • Offset: 8 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 8
n_scnm Section number of symbol	<ul style="list-style-type: none"> • Offset: 12 • Length: 2 	<ul style="list-style-type: none"> • Offset: 12 • Length: 2
n_type Basic and derived type specification	<ul style="list-style-type: none"> • Offset: 14 • Length: 2 	<ul style="list-style-type: none"> • Offset: 14 • Length: 2
n_lang Source language ID (overlays first byte of n_type)	<ul style="list-style-type: none"> • Offset: 14 • Length: 1 	<ul style="list-style-type: none"> • Offset: 14 • Length: 1
n_cpu CPU Type ID (overlays second byte of n_type)	<ul style="list-style-type: none"> • Offset: 15 • Length: 1 	<ul style="list-style-type: none"> • Offset: 15 • Length: 1
n_sclass Storage class of symbol	<ul style="list-style-type: none"> • Offset: 16 • Length: 1 	<ul style="list-style-type: none"> • Offset: 16 • Length: 1
n_numaux Number of auxiliary entries	<ul style="list-style-type: none"> • Offset: 17 • Length: 1 	<ul style="list-style-type: none"> • Offset: 17 • Length: 1
+Use "32" or "64" suffix when <code>__XCOFF_HYBRID__</code> is defined.		

Field Definitions: The following defines the symbol table entry fields:

n_name Used by XCOFF32 only. Specifies an 8-byte, null-padded symbol name or symbolic debugger stabstring. The storage class field is used to determine if the field is a symbol name or symbolic debugger stabstring. By convention, a storage class value with the high-order bit on indicates that this field is a symbolic debugger stabstring.

If the XCOFF32 symbol name is longer than 8 bytes, the field is interpreted as the following two fields:

n_zeroes

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

n_offset

Specifies the byte offset to the symbol name in the string table or .debug section (overlays last 4 bytes of n_name). 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_offset For XCOFF64: Specifies 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. (For XCOFF32 only, used in conjunction with n_zeroes. See entry immediately above.)

<code>n_value</code>	<p>Specifies the symbol value. The contents of the symbol value field is storage class-dependent, as shown in the following definitions:</p> <p>Content Storage Class</p> <p>Relocatable address <code>C_EXT, C_WEAKEXT, C_HIDEXT, C_FCN, C_BLOCK, C_STAT</code></p> <p>Zero <code>C_GSYM, C_BCOMM, C_DECL, C_ENTRY, C_ESTAT, C_ECOMM</code></p> <p>Offset in csect <code>C_FUN, C_STSYM</code></p> <p>Offset in file <code>C_BINCL, C_EINCL</code></p> <p>Offset in comment section <code>C_INFO</code></p> <p>Symbol table index <code>C_FILE, C_BSTAT</code></p> <p>Offset relative to stack frame <code>C_LSYM, C_PSYM</code></p> <p>Register number <code>C_RPSYM, C_RSYM</code></p> <p>Offset within common block <code>C_ECOML</code></p>
<code>n_sctnum</code>	<p>Specifies a section number associated with one of the following symbols:</p> <ul style="list-style-type: none"> -2 Specifies N_DEBUG, a special symbolic debugging symbol. -1 Specifies N_ABS, an absolute symbol. The symbol has a value but is not relocatable. 0 Specifies N_UNDEF, an undefined external symbol. <p>Any other value Specifies the section number where the symbol was defined.</p>
<code>n_type</code>	<p>Used in COFF for type information. This use is obsolete in XCOFF. For C_EXT and C_HIDEXT symbols, this field should contain 0x0020 for function symbols and 0 otherwise. This field has a special purpose for C_FILE symbols. See "File Auxiliary Entry for the C_FILE Symbol" for more information.</p>
<code>n_sclass</code>	<p>Specifies the storage class of the symbol. The storclass.h and dbxstclass.h files contain the definitions of the storage classes. See "Symbol Table Field Contents by Storage Class" for more information.</p>
<code>n_numaux</code>	<p>Specifies the number of auxiliary entries for the symbol. If more than one auxiliary entry is required for a symbol, 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.</p>

For general information on the XCOFF file format, see "XCOFF Object File Format."

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 length 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.

In XCOFF32, symbols having a storage class of **C_EXT**, **C_WEAKEXT** or **C_HIDEXT** and more than one auxiliary entry must have the csect auxiliary entry as the last auxiliary entry. In XCOFF64, the `x_auxtype` field of each auxiliary symbol table entry differentiates the symbols, but the convention is to generate the csect auxiliary symbol table entry last.

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**. The C language structure for a file auxiliary entry can be found in the **x_file** structure in the **syms.h** file.

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

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

Source Language ID	Overlays the high-order byte of the n_type field. This field contains the source-language identifier. The values for this field are defined in the e_lang field in "Exception Section" . This field can be used by the symbolic debuggers to determine the source language. The optional values for this field are 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 8-character name field begins with ' ' (blank) , '\0'(NULL). If the source language is TB_FRONT, the third byte is the stabstring compaction level for the object file, and the n_offset field contains the symbol table index of the TB_BACK symbol table entry, if it exists, or 0 otherwise.
--------------------	---

Defined as the low-order byte of the `n_type` field. Describes the kind of instructions generated for the file. The following values are defined:

0	Reserved.
1	Specifies , 32-bit mode.
2	Reserved.
3	Specifies the common intersection of 32-bit and Processor.
4	Specifies Processor.
5	Specifies any mix of instructions between different architectures.
6	Specifies a mix of and instructions ().
7-223	Reserved.
224	Specifies instructions.
225-255	Reserved.

If both fields are 0, no information is provided about the source language.

File Name Auxiliary Entry Format

Offset	Length in Bytes	Name	Description
0	14	x_fname	Source file string
0	4	x_zeroes	Zero, indicating file string in string table (overlays first 4 bytes of <code>x_fname</code>)
4	4	x_offset	Offset of file string in string table (overlays 5th-8th bytes of <code>x_fname</code>)
14	1	x_ftype	File string type
15	2		Reserved. Must contain 0.
17	1	x_auxtype	Auxiliary symbol type(XCOFF64 only)

Field Definitions: The following defines the fields listed above:

x_fname	Specifies the source file name or compiler-related string. If the file name or string is longer than 8 bytes, the field is interpreted as the following two fields: x_zeroes A value of 0 indicates that the source file string is in the string table (overlays first 4 bytes of x_fname). x_offset Specifies the offset from the beginning of the string table to the first byte of the source file string (overlays last 4 bytes of x_fname).
x_fstype	Specifies the source-file string type. 0 XFT_FN Specifies the source-file name 1 XFT_CT Specifies the compiler time stamp 2 XFT_CV Specifies the compiler version number 128 XFT_CD Specifies compiler-defined information (no name) Reserved. This field must contain 2 bytes of 0.
x_auxtype	(XCOFF64 only) Specifies the type of auxiliary entry. Contains <code>_AUX_FILE</code> for this auxiliary entry.

If the file auxiliary entry is not used, the symbol name is the name of the source file. If the file auxiliary entry is used, then the symbol name should be `.file`, 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.

csect Auxiliary Entry for C_EXT, C_WEAKEXT, 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**, **C_WEAKEXT**, or **C_HIDEXT**. See "Symbol Table Entry (syms.h)" for more information. By convention, the csect auxiliary entry in an XCOFF32 file 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 `syms.h` file.

Table 28. csect Auxiliary Entry Format

Field Name and Description	XCOFF32	XCOFF64
x_scnlen (See field definition section)	<ul style="list-style-type: none"> Offset: 0 Length: 4 	<ul style="list-style-type: none"> Offset: N/A Length: N/A
x_scnlen_lo (See field definition section) Low 4 bytes of section length	<ul style="list-style-type: none"> Offset: N/A Length: N/A 	<ul style="list-style-type: none"> Offset: 0 Length: 4
x_parmhash Offset of parameter type-check hash in <code>.typchk</code> section	<ul style="list-style-type: none"> Offset: 4 Length: 4 	<ul style="list-style-type: none"> Offset: 4 Length: 4
x_snhash .typchk section number	<ul style="list-style-type: none"> Offset: 8 Length: 2 	<ul style="list-style-type: none"> Offset: 8 Length: 2

Table 28. csect Auxiliary Entry Format (continued)

Field Name and Description	XCOFF32	XCOFF64
x_smtyp Symbol alignment and type 3-bit symbol alignment (log 2) 3-bit symbol type	<ul style="list-style-type: none"> • Offset: 10 • Length: 1 	<ul style="list-style-type: none"> • Offset: 10 • Length: 1
x_smc1as Storage mapping class	<ul style="list-style-type: none"> • Offset: 11 • Length: 1 	<ul style="list-style-type: none"> • Offset: 11 • Length: 1
x_stab Reserved	<ul style="list-style-type: none"> • Offset: 12 • Length: 4 	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A
x_snstab Reserved	<ul style="list-style-type: none"> • Offset: 16 • Length: 2 	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A
x_scnlen_hi (See field definition section) High 4 bytes of section length	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A 	<ul style="list-style-type: none"> • Offset: 12 • Length: 4
(pad) Reserved	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A 	<ul style="list-style-type: none"> • Offset: 16 • Length: 1
x_auxtype Contains <code>_AUX_CSECT</code> ; indicates type of auxiliary entry	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A 	<ul style="list-style-type: none"> • Offset: 17 • Length: 1

Field Definitions: The following defines the fields listed above:

`x_scnlen` Specifies a meaning 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 0.

In the XCOFF64 format, the value of `x_scnlen` is divided into two fields: `x_scnlen_hi`, representing the upper 4 bytes of the value, and `x_scnlen_lo`, representing the lower 4 bytes of the value.

`x_parmhash` Specifies the 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 byte of the parameter type-check string (not to its length field). See "Type-Check Section" for more information. A value of 0 in the `x_parmhash` 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. The value should be 0 for **C_HIDEXT** symbols.

`x_snhash` Specifies the `.typchk` section number. The XCOFF section number containing the parameter type-checking strings. The section numbers are one-based. For compatibility with object files generated by some compilers, if `x_parmhash` is not equal to 0 but `x_snhash` does equal 0, then the first `.typchk` section in the file is used. The value should be 0 for **C_HIDEXT** symbols.

x_smtyp

Specifies symbol alignment and type:

Bits 0-4

Contains a 5-bit csect address alignment value (log base 2). For example, a value of 3 in this field indicates 2³, or 8, meaning the csect is to be aligned on an 8-byte address value. The alignment value is used only when the value of bits 5-7 of the x_smtyp field is either **XTY_SD** or **XTY_CM**.

Bits 5-7

Contains a 3-bit symbol type field. See the definitions for bits 5-7 of the l_smtyp field in "Loader Section" for more information.

x_smc1as

Specifies the csect storage-mapping class. This field permits the binder to arrange csects by their storage-mapping class. The x_smc1as field is used only when the value of bits 5-7 of the x_smtyp field is either **XTY_SD** or **XTY_CM**.

The following storage-mapping classes are read-only and normally mapped to the .text section:

Value Class

Description

0 XMC_PR

Specifies program code. The csect contains the executable instructions of the program.

1 XMC_RO

Specifies a read-only constant. The csect contains data that is constant and will not change during execution of the program.

2 XMC_DB

Specifies the 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 separately from the executable code of the program.

6 XMC_GL

Specifies 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 calls. As a result, the csect maintains position-independent code within the .text section of the executable XCOFF object file.

7 XMC_XO

Specifies 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-absolute instruction.

12 XMC_TI

Reserved.

13 XMC_TB

Reserved.

The following storage-mapping classes are read/write and normally mapped to the .data or .bss section:

Value Class

Description

5 XMC_RW

Specifies 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 value is **XTY_SD**, the csect contains initialized data and is mapped into the .data section. If the x_smtyp value is **XTY_CM**, 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. The csect would have a storage class value of **C_HIDEXT**. An initialized definition for a global data scalar or structure from a C source file is contained in its own csect of this type. The csect would have a storage class value of **C_EXT**. A csect of this type is accessible by name references from other object files.

Value Class

Description

15 XMC_TC0

Specifies 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 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 limit the value that a relative displacement field within a load instruction may contain. This limit then becomes an inherent limit on the size of a TOC for an executable XCOFF object. For RS/6000, this limit is 65,536 bytes, or 16,384 4-byte TOC entries.

3 XMC_TC

Specifies general TOC entry. A csect of this type is usually 4 bytes in length and contains the address of another csect or global symbol. This csect provides 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 by the binder to eliminate duplicate TOC entries as follows:

- Symbols that have a storage class value of **C_EXT** are global symbols and must have names (a non-null `n_name` field). These symbols require no special TOC processing logic to combine duplicate entries. Duplicate entries with the same `n_name` value are combined into a single entry.
- Symbols that have a storage class value of **C_HIDEXT** are not global symbols, and duplicate entries are resolved by context. Any two such symbols will be defined as duplicates and combined 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 4 bytes long.
 - Each has a single RLD entry that references external symbols with the same name.

To minimize the number of duplicate TOC entries that cannot be combined by the binder, compilers and assemblers should adhere to a common naming convention for TOC entries. By convention, compilers and assemblers produce TOC entries that have a storage class value of **C_HIDEXT** and an `n_name` string that is the same as the `n_name` value for the symbol that the TOC entry addresses.

16 XMC_TD

Specifies scalar data entry in the TOC. A csect that is a special form of an **XMC_RW** csect that is directly accessed from the TOC by compiler generated code. This lets some frequently used global symbols be accessed directly from the TOC rather than indirectly through an address pointer csect contained in 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**.
- It is 4-bytes long or less.
- It has initialized data that can be modified as the program runs.
- If a same named csect of type **XMC_RW** or **XMC_UA** exist, it is replaced by the **XMC_TD** csect.

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

x_smc1as
continued

Value Class

Description

10 XMC_DS

Specifies a csect containing a function descriptor, which 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 is contained. The function descriptor has a storage class value of **C_EXT** and has an `n_name` value that is the same as the name of the function in the source file. The addresses of function descriptors are imported to and exported from an executable XCOFF file.

8 XMC_SV

Specifies 32-bit supervisor call descriptor csect. 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. These symbols are only available to 32-bit programs.

17 XMC_SV64

Specifies 64-bit supervisor call descriptor csect. See **XMV_SV** for supervisor call information. These symbols are only available to 64-bit programs.

18 XMC_SV3264

Specifies supervisor call descriptor csect for both 32-bit and 64-bit. See **XMV_SV** for supervisor call information. These symbols are available to both 32-bit and 64-bit programs.

4 XMC_UA

Unclassified. This csect is treated as read/write. This csect is frequently produced by an assembler or object file translator program that cannot determine the true classification of the resultant csect.

9 XMC_BS

Specifies 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

Specifies unnamed FORTRAN common. A csect of this type is intended for an 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 (Unused for 64-bit).

x_snstab Reserved (Unused for 64-bit).

Auxiliary Entries for the C_EXT, C_WEAKEXT, and C_HIDEXT Symbols

Auxiliary symbol table entries are defined in XCOFF to contain reference and size information associated with a defined function. These auxiliary entries are produced by compilers and assembler for use by the symbolic debuggers. In XCOFF32, a function auxiliary symbol table entry contains the required information. In XCOFF64, both a function auxiliary entry and an exception auxiliary entry may be needed. When both auxiliary entries are generated for a single **C_EXT**, **C_WEAKEXT**, or **C_HIDEXT** symbol, the `x_size` and `x_endndx` fields must have the same values.

The function auxiliary symbol table entry is defined in the following table.

Table 29. Function Auxiliary Entry Format

Field Name and Description	XCOFF32	XCOFF64
x_expnr File offset to exception table entry	<ul style="list-style-type: none"> • Offset: 0 • Length: 4 	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A
x_fsize Size of function in bytes	<ul style="list-style-type: none"> • Offset: 4 • Length: 4 	<ul style="list-style-type: none"> • Offset: 8 • Length: 4
x_innoptr File pointer to line number	<ul style="list-style-type: none"> • Offset: 8 • Length: 4 	<ul style="list-style-type: none"> • Offset: 0 • Length: 8
x_endndx Symbol table index of next entry beyond this function	<ul style="list-style-type: none"> • Offset: 12 • Length: 4 	<ul style="list-style-type: none"> • Offset: 12 • Length: 4
(pad) Unused	<ul style="list-style-type: none"> • Offset: 16 • Length: 1 	<ul style="list-style-type: none"> • Offset: 16 • Length: 1
x_auxtype Contains <code>_AUX_FCN</code> ; Type of auxiliary entry	<ul style="list-style-type: none"> • Offset: N/A • Length: N/A 	<ul style="list-style-type: none"> • Offset: 17 • Length: 1

Field Definitions: The following defines the fields listed in the Function Auxiliary Entry Format table:

<code>x_expnr</code>	(XCOFF32 only) This field is a file pointer to an exception table entry. The value is the byte offset from the beginning of the XCOFF object file. In an XCOFF64 file, the exception table offsets are in an exception auxiliary symbol table entry.
<code>x_fsize</code>	Specifies the size of the function in bytes.
<code>x_innoptr</code>	Specifies a file pointer to the line number. The value is the byte offset from the beginning of the XCOFF object file.
<code>x_endndx</code>	Specifies the symbol table index of the next entry beyond this function.

The exception auxiliary symbol table entry, defined in XCOFF64 only, is shown in the following table.

Table 30. Exception Auxiliary Entry Format (XCOFF64 only)

Offset	Length	Name and Description
0	8	x_expnr File offset to exception table entry.
8	4	x_fsize Size of function in bytes
12	4	x_endndx Symbol table index of next entry beyond this function
16	1	(pad) Unused
17	1	x_auxtype Contains <code>_AUX_EXCEPT</code> ; Type of auxiliary entry

Field Definitions: The following defines the fields listed in the Exception Auxiliary Entry Format table:

x_exptr This field is a file pointer to an exception table entry. The value is the byte offset from the beginning of the XCOFF object file.

x_fsize Specifies the size of the function in bytes.

x_endndx Specifies the symbol table index of the next entry beyond this function.

Block Auxiliary Entry for the 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. The section auxiliary symbol table entry is produced by compilers for use by the symbolic debuggers.

Table 31. Table Entry Format

Field Name and Description	XCOFF32	XCOFF64
(no name) Reserved	<ul style="list-style-type: none"> Offset: 0 Length: 4 	<ul style="list-style-type: none"> Offset: N/A Length: N/A
x_1nno Source line number	<ul style="list-style-type: none"> Offset: 4 Length: 2 	<ul style="list-style-type: none"> Offset: 0 Length: 4
(no name) Reserved	<ul style="list-style-type: none"> Offset: 6 Length: 12 	<ul style="list-style-type: none"> Offset: 4 Length: 13
x_auxtype Contains _AUX_SYM; Type of auxiliary entry	<ul style="list-style-type: none"> Offset: N/A Length: N/A 	<ul style="list-style-type: none"> Offset: 17 Length: 1

Field Definitions: The following defines the fields above:

(no name) Reserved.

x_1nno Specifies the line number of a source file. The maximum value of this field is 65535 for XCOFF64 and 2^{32} for XCOFF64.

(no name) Reserved.

Section Auxiliary Entry for the C_STAT Symbol

The section auxiliary symbol table entry ID is defined in XCOFF32 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 and removed by the binder.

Table 32. Section Auxiliary Entry Format (XCOFF32 Only)

Offset	Length in Bytes	Name and Description
0	4	x_scnlen Section length
4	2	x_nreloc Number of relocation entries
6	2	x_n1inno Number of line numbers
8	10	(no name) Reserved

Field Definitions: The following list defines the fields:

x_scnlen	Specifies section length in bytes.
x_nreloc	Specifies the number of relocation entries. The maximum value of this field is 65535.
x_nlinno	Specifies the number of line numbers. The maximum value of this field is 65535.
(no name)	Reserved.

For general information on the XCOFF file format, see "XCOFF Object File Format." For more information on the symbol table, see "Symbol Table Information."

For information on debugging, see "Debug Section."

Symbol Table Field Contents by Storage Class

This section defines the symbol table field contents for each of the defined storage classes (`n_sclass`) that are used in XCOFF. The following table lists storage class entries in alphabetic order. See "Symbol Table Entry (syms.h)" for more information.

Table 33. Symbol Table by Storage Class

Class Definition	Field Contents
C_BCOMM 135 Beginning of common block	<p>n_name Name of the common block*</p> <p>n_value 0, undefined</p> <p>n_sclass N_DEBUG</p> <p>Aux. Entry</p>
C_BINCL 108 Beginning of include file	<p>n_name Source name of the include file**</p> <p>n_value File pointer</p> <p>n_sclass N_DEBUG</p> <p>Aux. Entry</p>
C_BLOCK 100 Beginning or end of inner block	<p>n_name .bb or .eb</p> <p>n_value Relocatable address</p> <p>n_sclass N_SCNUM</p> <p>Aux. Entry BLOCK</p>

Table 33. Symbol Table by Storage Class (continued)

Class Definition	Field Contents
C_BSTAT 143 Beginning of static block	n_name .bs n_value Symbol table index n_scnm N_DEBUG Aux. Entry
C_DECL 140 Declaration of object (type)	n_name Debugger stabstring* n_value 0, undefined n_scnm N_SCNUM Aux. Entry
C_ECOML 136 Local member of common block	n_name Debugger stabstring* n_value Offset within common block n_scnm N_ABS Aux. Entry
C_ECOMM 137 End of common block	n_name Debugger stabstring* n_value 0, undefined n_scnm N_DEBUG Aux. Entry
C_EINCL 109 End of include file	n_name Source name of the include file** n_value File pointer n_scnm N_DEBUG Aux. Entry

Table 33. Symbol Table by Storage Class (continued)

Class Definition	Field Contents
C_ENTRY 141 Alternate entry	n_name * n_value 0, undefined n_scnm N_DEBUG Aux. Entry
C_ESTAT 144 End of static block	n_name .es n_value 0, undefined n_scnm N_DEBUG Aux. Entry
C_EXT 2 External symbol (defining external symbols for binder processing)	n_name Symbol Name** n_value Relocatable address n_scnm N_SCNUM or N_UNDEF Aux. Entry FUNCTION CSECT
C_FCN 101 Beginning or end of function	n_name .bf or .ef n_value Relocatable address n_scnm N_SCNUM Aux. Entry BLOCK
C_FILE 103 Source file name and compiler information	n_name .file or source file name (if no auxiliary entries)** n_value Symbol table index n_scnm N_DEBUG Aux. Entry FILE

Table 33. Symbol Table by Storage Class (continued)

Class Definition	Field Contents
C_FUN 142 Function or procedure	n_name Debugger stabstring* n_value Offset within containing csect n_scnm N_ABS Aux. Entry
C_GSYM 128 Global variable	n_name Debugger stabstring* n_value 0, undefined n_scnm N_DEBUG Aux. Entry
C_HIDEXT 107 Unnamed external symbol	n_name Symbol Name or null** n_value Relocatable address n_scnm N_SCNUM Aux. Entry FUNCTION CSECT
C_INFO 100 Comment section reference	n_name Info Name Identifier or null** n_value Offset within comment section n_scnm N_SCNUM Aux. Entry
C_LSYM 129 Automatic variable allocated on stack	n_name Debugger stabstring* n_value Offset relative to stack frame n_scnm N_ABS Aux. Entry

Table 33. Symbol Table by Storage Class (continued)

Class Definition	Field Contents
C_NULL 0 Symbol table entry marked for deletion.	n_name n_value 0x00DE1E00 n_scnm Aux. Entry Any
C_PSYM 130 Argument to subroutine allocated on stack	n_name Debugger stabstring* n_value Offset relative to stack frame n_scnm N_ABS Aux. Entry
C_RPSYM 132 Argument to function or procedure stored in register	n_name Debugger stabstring* n_value Register number n_scnm N_ABS Aux. Entry
C_RSYM 131 Register variable	n_name Debugger stabstring* n_value Register number n_scnm N_ABS Aux. Entry
C_STAT 3 Static symbol (Unknown. Some compilers generate these symbols in the symbol table to identify size of the .text , .data , and .bss sections. Not used or preserved by binder.)	n_name Symbol Name** n_value Relocatable address n_scnm N_SCNUM Aux. Entry SECTION

Table 33. Symbol Table by Storage Class (continued)

Class Definition	Field Contents
C_STSYM 133 Statically allocated symbol	n_name Debugger stabstring* n_value Offset within csect n_scnm N_ABS Aux. Entry
C_TCSYM 134 Reserved	n_name Debugger stabstring* n_value n_scnm Aux. Entry
C_WEAKEXT 111 Weak external symbol (defining weak external symbols for binder processing)	n_name Symbol Name** n_value Relocatable address n_scnm N_SCNUM or N_UNDEF Aux. Entry FUNCTION CSECT

Notes:

1. *For long name, the `n_offset` value is an offset into the `.debug` section.
2. **For long name, the `n_offset` value is an offset into the string table.

Storage Classes by Usage and Symbol Value Classification

Following are the storage classes used and relocated by the binder. The symbol values (`n_value`) are addresses.

Class	Description
C_EXT	Specifies an external or global symbol
C_WEAKEXT	Specifies an external or global symbol with weak binding
C_HIDEXT	Specifies an internal symbol
C_BLOCK	Specifies the beginning or end of an inner block (<code>.bb</code> or <code>.eb</code>)
C_FCN	Specifies the beginning or end of a function (<code>.bf</code> or <code>.ef</code> only)
C_STAT	Specifies a static symbol (contained in <code>statics</code> csect)

Following are storage classes used by the binder and symbolic debugger or by other utilities for file scoping and accessing purposes:

C_FILE	Specifies the source file name. The <code>n_value</code> field holds the symbol index of the next file entry. The <code>n_name</code> field is the name of the file.
C_BINCL	Specifies the beginning of include header file. The <code>n_value</code> field is the line number byte offset in the object file to the first line number from the include file.

- C_EINCL** Specifies the end of include header file. The `n_value` field is the line number byte offset in the object file to last line number from the include file.
- C_INFO** Specifies the location of a string in the comment section. The `n_value` field is the offset to a string of bytes in the specified **STYP_INFO** section. The string is preceded by a 4-byte length field. The `n_name` field is preserved by the binder. An application-defined unique name in this field can be used to filter access to only those comment section strings intended for the application.

Following are the storage classes that exist only for symbolic debugging purposes:

- C_BCOMM** Specifies the beginning of a common block. The `n_value` field is meaningless; the name is the name of the common block.
- C_ECOML** Specifies a local member of a common block. The `n_value` field is byte-offset within the common block.
- C_ECOMM** Specifies the end of a common block. The `n_value` field is meaningless.
- C_BSTAT** Specifies the beginning of a static block. The `n_value` field is the symbol table index of the csect containing static symbols; the name is `.bs`.
- C_ESTAT** Specifies the end of a static block. The `n_value` field is meaningless; the name is `.es`.
- C_DECL** Specifies a declaration of object (type declarations). The `n_value` field is undefined.
- C_ENTRY** Specifies an alternate entry (FORTRAN) and has a corresponding **C_EXT** or **C_WEAKEXT** symbol. The `n_value` field is undefined.
- C_FUN** Specifies a function or procedure. May have a corresponding **C_EXT** or **C_WEAKEXT** symbol. The `n_value` field is byte-offset within the containing csect.
- C_GSYM** Specifies a global variable and has a corresponding **C_EXT** or **C_WEAKEXT** symbol. The `n_value` field is undefined.
- C_LSYM** Specifies an automatic variable allocated on the stack. The `n_value` field is byte offset relative to the stack frame (platform dependent).
- C_PSYM** Specifies an argument to a subroutine allocated on the stack. The `n_value` field is byte-offset relative to the stack frame (platform dependent).
- C_RSYM** Specifies a register variable. The `n_value` field is the register number.
- C_RPSYM** Specifies an argument to a function or procedure stored in a register. The `n_value` field is the register number where argument is stored.
- C_STSYM** Specifies a statically allocated symbol. The `n_value` field is byte-offset within csect pointed to by containing **C_BSTAT** entry.

For general information on the XCOFF file format, see "XCOFF Object File Format." For more information on the symbol table, see "Symbol Table Information."

For information on debugging, see "Debug Section."

String Table

IN XCOFF32, the string table contains the names of symbols that are longer than 8 bytes. In XCOFF64, the string table contains the names of all symbols. If the string table is present, the first 4 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 0, 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. A value of 4 is preferable. The following table shows string table organization.

Table 34. String Table Organization

Offset	Length in Bytes	Description
0	4	Length of string table.
4	<i>n</i>	Symbol name string, null-terminated.
		Field repeats for each symbol name.

For general information on the **XCOFF** file format, see "XCOFF Object File Format."

dbx Stabstrings

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.

See "Debug Section" for a general discussion.

Stabstring Terminal Symbols

In the stabstring grammar, there are five types of terminal symbols, which are written in all capital letters. These symbols are described by the regular expressions in the following list:

Note: The [] (brackets) denote one instance, []* (brackets asterisk) denote zero or more instances, []+ (brackets plus sign) denote one or more instances, () (parentheses) denote zero or one instance, .* (dot asterisk) denotes a sequence of zero or more bytes, and | (pipe) denotes alternatives.

Symbol
NAME
STRING

Regular Expression

[^ ; : ' "] (A name consists of any non-empty set of characters, excluding ; : ' or ".)

'.*' | ".*", where \", \', or \\ can be used inside the string

Within a string, the \ (backslash character) may have a special meaning. If the character following the \ is another \, one of the backslashes is ignored. If the next character is the quote character used for the current string, the string is interpreted as containing an embedded quote. Otherwise, the \ is interpreted literally. However, if the closing quote is the last character in the stabstring, and a \ occurs immediately before the quote, the \ is interpreted literally. This use is not recommended.

The \ must be quoted only in the following instances:

- The \ is the last character in the string (to avoid having the closing quote escaped).
- The \ is followed by the current quote character.
- The \ is followed by another \.

An escaped quote is required only when a single string contains both a single quote and a double quote. Otherwise, the string should be quoted with the quote character not contained in the strings.

A string can contain embedded null characters, so utilities that process stabstrings must use the length field to determine the length of a stabstring.

INTEGER
HEXINTEGER

(-)[0-9]+

[0-9A-F]+

The hexadecimal digits **A-F** must be uppercase.

REAL

[+-][0-9]+(.[0-9]*([eEqQ](+-)[0-9]+) | (+-)|INF | QNAN | SNAN

Real numbers are the same strings recognized by the **scanf** subroutine when using the "%lf" pattern. Therefore, white space may occur before a real number.

Stabstring Grammar

REALs may be preceded by white space, and STRINGs may contain any characters, including null and blank characters. Otherwise, there are no null or blank characters in a stabstring.

Long stabstrings can be split across multiple symbol table entries for easier handling. In the stabstring grammar, a # (pound sign) indicates a point at which a stabstring may be continued. A continuation is

indicated by using either the ? (question mark) or \ as the last character in the string. The next part of the stabstring is in the name of the next symbol table entry. If an alternative for a production is empty, the grammar shows the keyword /*EMPTY*/.

The following list contains the stabstring grammar:

Stabstring:

Basic structure of stabstring:

NAME : *Class*

Name of object followed by object classification

:*Class* Unnamed object classification.

Class: Object classifications:

c = *Constant* ;

Constant object

NamedType

User-defined types and tags

Parameter

Argument to subprogram

Procedure

Subprogram declaration

Variable

Variable in program

Label Label object.

Constant:

Constant declarations:

b *OrdValue*

Boolean constant

c *OrdValue*

Character constant

e *TypeID , OrdValue*

Enumeration constant

i **INTEGER**

Integer constant

r **REAL**

Floating point constant

s **STRING**

String constant

C **REAL, REAL**

Complex constant

S *TypeID , NumElements , NumBits , BitPattern*

Set constant.

OrdValue:

Associated numeric value: INTEGER

NumElements:

Number of elements in the set: INTEGER

NumBits:

Number of bits in item: INTEGER

NumBytes:

Number of bytes in item: INTEGER

BitPattern:

Hexadecimal representation, up to 32 bytes: HEXINTEGER

NamedType:

User-defined types and tags:

t *TypeID*

User-defined type (TYPE or typedef), excluding those that are valid for **T** *TypeID*

T *TypeID*

Struct, union, class, or enumeration tag

Parameter:

Argument to procedure or function:

a *TypeID*

Passed by reference in general register

p *TypeID*

Passed by value on stack

v *TypeID*

Passed by reference on stack

C *TypeID*

Constant passed by value on stack

D *TypeID*

Passed by value in floating point register

R *TypeID*

Passed by value in general register

X *TypeID*

Passed by value in vector register

Procedure:

Procedure or function declaration:

Proc Procedure at current scoping level

Proc , **NAME : NAME**

Procedure named 1st NAME, local to 2nd NAME, where 2nd NAME is different from the current scope.

Variable:

Variable in program:

TypeID

Local (automatic) variable of type *TypeID*

d *TypeID*

Floating register variable of type *TypeID*

r *TypeID*

Register variable of type *TypeID*

x *TypeID*

Vector register variable of type *TypeID*

- G** *TypeID*
Global (external) variable of type *TypeID*
- S** *TypeID*
Module variable of type *TypeID* (C static global)
- V** *TypeID*
Own variable of type *TypeID* (C static local)
- Y** FORTRAN pointer variable
- Z** *TypeID* **NAME**
FORTRAN pointee variable

Label: Label:

- L** Label name.

Proc: Different types of functions and procedures:

- f** *TypeID*
Private function of type *TypeID*
- g** *TypeID*
Generic function (FORTRAN)
- m** *TypeID*
Module (Modula-2, ext. Pascal)
- J** *TypeID*
Internal function of type *TypeID*
- F** *TypeID*
External function of type *TypeID*
- I** (capital i) Internal procedure
- P** External procedure
- Q** Private procedure

TypeID:

Type declarations and identifiers:

- INTEGER**
Type number of previously defined type
- INTEGER = TypeDef**
New type number described by *TypeDef*
- INTEGER = TypeAttrs TypeDef**
New type with special type attributes

TypeAttrs:

@ *TypeAttrList* ;

Note: Type attributes (*TypeAttrs*) are extra information associated with a type, such as alignment constraints or pointer-checking semantics. The **dbx** program recognizes only the **size** attribute and the **packed** attribute. The **size** attribute denotes the total size of a padded element within an array. The **packed** attribute indicates that a type is a packed type. Any other attributes are ignored by **dbx**.

TypeAttrList:

List of special type attributes:

TypeAttrList ;
 @ *TypeAttr*
 TypeAttr

TypeAttr:

Special type attributes:

a INTEGER

Align boundary

s INTEGER

Size in bits

p INTEGER

Pointer class (for checking)

P Packed type

Other Anything not covered is skipped entirely

TypeDef:

Basic descriptions of objects:

INTEGER

Type number of a previously defined type

b *TypeID* ; # *NumBytes*

Pascal space type

c *TypeID* ; # *NumBits*

Complex type *TypeID*

d *TypeID*

File of type *TypeID*

e *EnumSpec* ;

Enumerated type (default size, 32 bits)

g *TypeID* ; # *NumBits*

Floating-point type of size *NumBits*

For *i* types, *ModuleName* refers to the Modula-2 module from which it is imported.

i **NAME : NAME** ;

Imported type *ModuleName:Name*

i **NAME : NAME** , *TypeID* ;

Imported type *ModuleName:Name* of type *TypeID*

k *TypeID*

C++ constant type

I ; # Usage-is-index; specific to COBOL

m *OptVBaseSpec* *OptMultiBaseSpec* *TypeID* : *TypeID* : *TypeID* ;

C++ pointer to member type; the first *TypeID* is the member type; the second is the type of the class

n *TypeID* ; # *NumBytes*

String type, with maximum string length indicated by *NumBytes*

o **NAME** ;

Opaque type

o **NAME** , *TypeID*

Opaque type with definition of *TypeID*

w *TypeID*
Wide character

z *TypeID ; # NumBytes*
Pascal gstring type

C *Usage*
COBOL Picture

I *NumBytes ; # PicSize*
(uppercase i) Index is type; specific to COBOL

K *CobolFileDesc;*
COBOL File Descriptor

M *TypeID ; # Bound*
Multiple instance type of *TypeID* with length indicated by Bound

N *Pascal Stringptr*

S *TypeID*
Set of type *TypeID*

***** *TypeID*
Pointer of type *TypeID*

& *TypeID*
C++ reference type

V *TypeID*
C++ volatile type

Z *C++ ellipses parameter type*

Array Subrange ProcedureType
For function types rather than declarations

Record
Record, structure, union, or group types

EnumSpec:
List of enumerated scalars:

EnumList
Enumerated type (C and other languages)

TypeID : EnumList
C++ enumerated type with repeating integer type

EnumList:
Enum EnumList Enum

Enum: Enumerated scalar description:
NAME : *OrdValue* , #

Array: Array descriptions:

a *TypeID ; # TypeID*
Array; *FirstTypeID* is the index type

A *TypeID*
Open array of *TypeID*

D **INTEGER** , *TypeID*
N-dimensional dynamic array of *TypeID*

E **INTEGER** , *TypeID*
N-dimensional dynamic subarray of *TypeID*

O **INTEGER** , *TypeID*
New open array

P *TypeID* ; # *TypeID*
Packed array

Subrange:

Subrange descriptions:

r *TypeID* ; # *Bound* ; # *Bound*
Subrange type (for example, char, int,\,), lower and upper bounds

Bound:

Upper and lower bound descriptions:

INTEGER
Constant bound

Boundtype **INTEGER**
Variable or dynamic bound; value is address of or offset to bound

J Bound is indeterminable (no bounds)

Boundtype:

Adjustable subrange descriptions:

- A** Bound passed by reference on stack
- S** Bound passed by value in static storage
- T** Bound passed by value on stack
- a** Bound passed by reference in register
- t** Bound passed by value in register

ProcedureType:

Function variables (1st type C only; others Modula-2 & Pascal)

f *TypeID* ;
Function returning type *TypeID*

f *TypeID* , *NumParams* ; *TParamList* ;
Function of N parameters returning type *TypeID*

p *NumParams* ; *TParamList* ;
Procedure of N parameters

R *NumParams* ; *NamedTParamList*
Pascal subroutine parameter

F *TypeID* , *NumParams* ; *NamedTParamList* ;
Pascal function parameter

NumParams:

Number of parameters in routine:

INTEGER.

TParamList:

Types of parameters in Modula-2 function variable:

TParam
Type of parameter and passing method

TParam:

Type and passing method

TypeID , *PassBy* ; #

NamedTParamList:

Types of parameters in Pascal-routine variable:

*/*EMPTY*/*

NamedTPList

NamedTPList:

NamedTParam *NamedTPList* *NamedTParam*

NamedTParam:

Named type and passing method:

Name : *TypeID* , *PassBy* *InitBody* ; #

: *TypeID* , *PassBy* *InitBody* ; #

Unnamed parameter

Record:

Types of structure declarations:

- **s** *NumBytes* # *FieldList* ;
- Structure or record definition
- **u** *NumBytes* # *FieldList* ;
- Union
- **v** *NumBytes* # *FieldList* *VariantPart* ;
- Variant Record
- **Y** *NumBytes* *ClassKey* *OptPBV* *OptBaseSpecList* (*ExtendedFieldList* *OptNameResolutionList* ;
- C++ class
- **G** *Redefinition* , **n** *NumBits* # *FieldList* ;
- COBOL group without conditionals
- Gn** *NumBits* *FieldList* ;
- **G** *Redefinition* , **c** *NumBits* # *CondFieldList* ;
- COBOL group with conditionals
- Gc** *NumBits* *CondFieldList* ;

OptVBaseSpec:

v ptr-to-mem class has virtual bases.

*/*EMPTY*/*

Class has no virtual bases.

OptMultiBaseSpec:

m Class is multi-based.

*/*EMPTY*/*

Class is not multi-based.

OptPBV:

V Class is always passed by value.

*/*EMPTY*/*

Class is never passed by value.

ClassKey:

s struct

u union

c class

OptBaseSpecList:
*/*EMPTY*/ BaseSpecList*

BaseSpecList:
BaseSpec
BaseSpecList , BaseSpec

BaseSpec:
VirtualAccessSpec BaseClassOffset : ClassTypeID

BaseClassOffset:
INTEGER
Base record offset in bytes

ClassTypeID:
TypeID
Base class type identifier

VirtualAccessSpec:

v *AccessSpec*
Virtual

v Virtual

AccessSpec

*/*EMPTY*/*

GenSpec:

c Compiler-generated

*/*EMPTY*/*

AccessSpec:

i # Private

o # Protected

u # Public

AnonSpec:

a Anonymous union member

*/*EMPTY*/*

VirtualSpec:

v p Pure virtual

v Virtual

*/*EMPTY*/*

ExtendedFieldList:

ExtendedFieldList ExtendedField

*/*EMPTY*/*

ExtendedField:

GenSpec AccessSpec AnonSpec DataMember

GenSpec VirtualSpec AccessSpec OptVirtualFuncIndex MemberFunction

AccessSpec AnonSpec NestedClass
AnonSpec FriendClass
AnonSpec FriendFunction

DataMember:

MemberAttrs : Field ;

MemberAttrs:

IsStatic IsVtblPtr IsVBasePtr

IsStatic:

*/*EMPTY*/*

s Member is static.

IsVtblPtr:

*/*EMPTY*/*

p **INTEGER NAME**

Member is vtbl pointer; NAME is the external name of v-table.

IsVBasePtr:

*/*EMPTY*/*

b Member is vbase pointer.

r Member is vbase self-pointer.

Member Function:

[FuncType MemberFuncAttrs : NAME : TypeID ; #

MemberFuncAttrs:

IsStatic IsInline IsConst IsVolatile

IsInline:

*/*EMPTY*/*

i Inline function

IsConst:

*/*EMPTY*/*

k const member function

IsVolatile:

*/*EMPTY*/*

V Volatile member function

NestedClass:

N *TypeID ; #*

FriendClass:

(*TypeID ; #*

FriendFunction:

] *NAME : TypeID ; #*

OptVirtualFuncIndex:

*/*EMPTY*/* **INTEGER**

FuncType:

f Member function

c Constructor

d Destructor

InitBody:

STRING
/*EMPTY*/

OptNameResolutionList:

/*EMPTY*/
) *NameResolutionList*

NameResolutionList: *NameResolution*
NameResolution , *NameResolutionList*

NameResolution: *MemberName* : *ClassTypeID*
Name is resolved by compiler.

MemberName:
Name is ambiguous.

MemberName:

NAME

FieldList:

Structure content descriptions:

Field /*EMPTY*/

FieldList *Field*
Member of record or union.

Field: Structure-member type description:

NAME : *TypeID* , *BitOffset* , *NumBits* ; #

VariantPart:

Variant portion of variant record:

[*Vtag* *VFieldList*]
Variant description

VTag: Variant record tag:

(*Field* Member of variant record

(**NAME** : ; #
Variant key name

VFieldList:

Variant record content descriptions:

VList
VFieldList *VList*
Member of variant record

VList: Variant record fields:

VField
VField *VariantPart*
Member of variant record

VField:

Variant record member type description:

(*VRangeList* : *FieldList*
Variant with field list

VRangeList:

List of variant field labels:

VRange
VRangeList , *VRange*
Member of variant record

VRange:

Variant field descriptions:

- b** *OrdValue*
Boolean variant
- c** *OrdValue*
Character variant
- e** *TypeID* , *OrdValue*
Enumeration variant
- i** **INTEGER**
Integer variant
- r** *TypeID* ; *Bound* ; *Bound*
Subrange variant

CondFieldList:

Conditions,#FieldList
FieldList# ;

Conditions:

*/*Empty*/*
Conditions condition

BitOffset:

Offset in bits from beginning of structure: INTEGER

Usage:

Cobol usage description:
PICStorageType NumBits , EditDescription , PicSize ;
Redefinition , PICStorageType NumBits , EditDescription , PicSize ;
PICStorageType NumBits , EditDescription , PicSize , # Condition ;
Redefinition , PICStorageType NumBits , EditDescription , PicSize , # Condition ;

Redefinition:

Cobol redefinition: **r** NAME

PICStorageType:

Cobol PICTURE types:

- a** Alphabetic
- b** Alphabetic, edited
- c** Alphanumeric
- d** Alphanumeric, edited
- e** Numeric, signed, trailing, included
- f** Numeric, signed, trailing, separate
- g** Numeric, signed, leading, included
- h** Numeric, signed, leading, separate
- i** Numeric, signed, default, comp
- j** Numeric, unsigned, default, comp
- k** Numeric, packed, decimal, signed

l	Numeric, packed, decimal, unsigned
m	Numeric, unsigned, comp-x
n	Numeric, unsigned, comp-5
o	Numeric, signed, comp-5
p	Numeric, edited
q	Numeric, unsigned
s	Indexed item
t	Pointer

EditDescription:

Cobol edit description:

STRING

Edit characters in an alpha PIC

INTEGER

Decimal point position in a numeric PIC

PicSize:

Cobol description length:

INTEGER

Number of repeated '9's in numeric clause, or length of edit format for edited numeric

Condition:

Conditional variable descriptions:

NAME : INTEGER = **q** *ConditionType* , *ValueList* ; #

ConditionType:

Condition descriptions:

ConditionPrimitive , *KanjiChar*

ConditionPrimitive:

Primitive type of Condition:

n *Sign DecimalSite*

Numeric conditional

a Alphanumeric conditional

f Figurative conditional

Sign: For types with explicit sign:

+ Positive

- Negative

[^+-] Not specified

DecimalSite:

Number of places from left for implied decimal point:

INTEGER

KanjiChar:

0 only if Kanji character in value: INTEGER

ValueList

Values associated with condition names

Value ValueList Value

Value Values associated with condition names:

INTEGER : *ArbitraryCharacters #*
Integer indicates length of string

CobolFileDesc:

COBOL file description:
Organization AccessMethod NumBytes

Organization:

COBOL file-description organization:

i Indexed
l Line Sequential
r Relative
s Sequential

AccessMethod:

COBOL file description access method:

d Dynamic
o Sort
r Random
s Sequential

PassBy:

Parameter passing method:

INTEGER
0 = passed-by reference; 1 = passed-by value

Related Information

Header Files.

The **as** command, **dbx** command, **dump** command, **ld** command, **size** command, **strip** command, and **what** command.

