SYSTEM V
APPLICATION BINARY INTERFACE

Edition 4.1

ACKNOWLEDGEMENTS

"We acknowledge the contributions of the 88OPEN Consortium Ltd., portions of whose *System V ABI Implementation Guide for the M88000 Processor* and the *System V ABI M88000 Processor Networking Supplement* have been incorporated in this section of the ABI with permission."

Contents

Table of Contents

Figures and Tables

1 INTRODUCTION

System V Application Binary Interface

The *System V Application Binary Interface*, or *ABI*, defines a system interface for compiled application programs and a minimal environment for support of installation scripts. Its purpose is to document a standard binary interface for application programs on systems that implement an operating system that complies with the *X/Open Common Application Environment Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition*.

The *ABI* defines a binary interface for application programs that are compiled and packaged for System V implementations on many different hardware architectures. Since a binary specification must include information specific to the computer processor architecture for which it is intended, it is not possible for a single document to specify the interface for all possible System V implementations. Therefore, the *System V ABI* is a family of specifications, rather than a single one.

The *System V ABI* is composed of two basic parts: A generic part of the specification describes those parts of the interface that remain constant across all hardware implementations of System V, and a processor-specific part of the specification describes the parts of the specification that are specific to a particular processor architecture. Together, the generic *ABI* and the processor-specific supplement for a single hardware architecture provide a complete interface specification for compiled application programs on systems that share a common hardware architecture.

This document is the generic *ABI*. It must be used in conjunction with a supplemental specification containing processor-specific information. Whenever a section of this specification must be supplemented by processor-specific information, the text will reference the processor supplement. The processor supplement may also contain additional information that is not referenced here.

Foundations and Structure of the ABI

The *System V ABI* is based on several reference documents. Because it is a binary interface, it includes the fundamental set of machine instructions for that processor to which the specification applies, and includes many other low-level specifications that may be strongly affected by the characteristics of a given processor's architecture. It also includes higher-level information about System V.

The interfaces specified here were drawn from existing standards for operating systems, user interfaces, programming languages, and networking, including those on the following list and others.

- The architecture manual for the target system's processor.
- The *System V Interface Definition, Fourth Edition*. M
- The IEEE *POSIX 1003.1-1990* standard operating system specification.
- The *X/Open Common Application Environment Specification (CAE), Issue 4.2*. X
- The *International Standard, ISO/IEC 9899:1990 (E), Programming Languages - C,* 12/15/90. E
- The *X Window System™, X Version 11, Release 5,* graphical user interface specification. G

Conformance Rule

NOTE

All interfaces in the *X/Open CAE Specification, Issue 4.2* (excluding those marked Withdrawn) and certain interfaces in the *System V Application Binary Interface, Fourth Edition* are contained in this document, the *System V Application Binary Interface, Edition 4.1 (System V ABI 4.1)*. Note that all interfaces in the System V ABI are conformant to XPG4.2. Those interfaces that were originally in the *System V Application Binary Interface, Fourth Edition* will also be conformant to the *System V Interface Definition, Fourth Edition* in cases where the SVID adds additional functionality over XPG4.2. In all cases where *System V Interface Definition, Fourth Edition* functionality conflicts with *X/Open CAE Specification, Issue 4.2* functionality, *X/Open CAE Specification, Issue 4.2* is followed. For this reason certain libraries presented in Chapter 6 and Chapter 8 will be divided into two tables. The first table presents interfaces which are governed by the specifications in the *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition*. The second tables presents interfaces which were not originally in the *System V Application Binary Interface, Fourth Edition* and are therefore only governed by the specifications in the *X/Open CAE Specification, Issue 4.2*.

X X X X X X X X X X X X X X X

The *ABI* is divided into sections dealing with specific portions of the interface. Some sections include a large amount of detailed information, while others contain lists of interface components and pointers to other documents.

In general, this specification does not duplicate information that is available in other standards documents. For example, the *ABI* section that describes system service routines includes a list of the system routines supported in this interface, formal declarations of the data structures they use that are visible to application programs, and a pointer to the *X/Open CAE Specification, Issue 4.2* and the *System V* M
Interface Definition, Fourth Edition for information about the syntax and semantics of each call. Only those routines not described in *System V Interface Definition,* M
Fourth Edition or elsewhere are described in the *ABI*.

Other sections of the *ABI* are written using this same model. The *ABI* identifies operating system components it includes, provides whatever information about those components that is not available elsewhere, and furnishes a reference to another document for further information. Information referenced in this way is as much a part of the *ABI* specification as is the information explicitly included here.

How to Use the System V ABI

The complete *System V ABI* is composed of this generic *ABI* specification and the supplemental processor-specific specification for a particular processor architecture. These two documents constitute a specification that should be used in conjunction with the publicly-available standards documents it references (some of these are listed above). The *ABI* enumerates the system components it includes, but descriptions of those components may be included entirely in the *ABI*, partly in the *ABI* and partly in other documents, or entirely in other reference documents.

Application programmers who wish to produce binary packages that will install and run on any System V-based computer should follow this procedure:

1. Write programs using the *X/Open CAE Specification, Issue 4.2* and the Level 1 X
   source-level interfaces described in the *System V Interface Definition, Fourth* X
   Edition in the following sections. (See the Conformance Rule above.) Routines guaranteed to be present on all ABI-conforming systems as dynamically-linkable resources are listed below.

   - BA_OS: All Level 1 routines are available as shared resources. E
   - BA_LIB: All Level 1 routines are available as shared resources E
     except for the math routines which may be available as an ABI E
     compliant static archive (see Chapter 11):

`acos`	`acosh`	`asin`	`asinh`	`atan`	
`atan2`	`atanh`	`cbrt`	`ceil`	`cos`	
`cosh`	`erf`	`erfc`	`exp`	`fabs`	
`floor`	`fmod`	`gamma`	`hypot`	`j0`	
`j1`	`jn`	`lgamma`	`log10`	`log`	
`matherr`	`pow`	`remainder`	`sin`	`sinh`	*
`sqrt`	`tan`	`tanh`	`y0`	`y1`	
`yn`					

     ABI-conforming applications should not use the encryption routines `crypt()`, `encrypt()`, or `setkey()`, and should not use the Level 2 routines `advance()`, `compile()`, `step()`, `drand48()`, `erand48()`, `jrand48()`, `lcong48()`, `lrand48()`, `mrand48()`, `nrand48()`, `seed48()`, and `srand48()`, since these routines are not required to be present on an ABI-conforming system.

- KE_OS: All Level 1 routines in this section are guaranteed to be E
  present as shared resources on an ABI-conforming system.
- RS_LIB: All routines in this section are guaranteed to be present as shared resources on ABI-conforming systems that include networking facilities.
- MT_LIB: All routines in this section are guaranteed to be present M
  as shared resources on ABI-conforming systems.

The routines listed in the ''System Library'', ''Threads Library'', ''Network E
Services Library'', and the ''X Window System Library'' (see: Chapter 6, 10) E
must be accessed as dynamically-linked resources. Other routines may be dynamically linked, or may be statically bound into the application from an archive library.

2. Use only the system utilities and environment information described in Chapters 8 and 9 of the *ABI*.
3. Compile programs so that the resulting executable programs use the specified interface to all system routines and services, and have the format
   described in the *ABI* specification. The commands available on a system E
   that also supports an ABI development environment is defined in Chapter E
   11.
4. Package the application in the format and on the media described in the *ABI*, and install or create files only in the specified locations provided for
   this purpose when the application is installed. The packaging tools avail- E
   able on a system that also supports an ABI development environment is E
   defined in Chapter 11.

The manufacturers of System V-based computer systems who wish to provide the system interface described in this specification must satisfy a complementary set of requirements:

1. Their system must implement fully the architecture described in the hardware manual for their target processor architecture.
2. The system must be capable of executing compiled programs having the format described in this specification.
3. The system must provide libraries containing the routines specified by the *ABI*, and must provide a dynamic linking mechanism that allows these routines to be attached to application programs at run time. All the system
   routines must behave as specified by the *X/Open CAE Specification, Issue 4.2* X
   and the *System V Interface Definition, Fourth Edition* (see the Conformance X
   Rule above).

4. The system's map of virtual memory must conform to the requirements of the *ABI*.
5. The system's low-level behavior with respect to function call linkage, system traps, signals, and other such activities must conform to the formats documented in the *ABI*.
6. The system's compilation system, if present, must compile source code into executable files having the formats and characteristics specified in the *ABI*.
7. The system must provide all files and utilities specified as part of the *ABI*, in the format defined here and in other referenced documents. All commands
   and utilities must behave as documented in the *X/Open CAE Specification,* X
   Issue 4.2 and the *System V Interface Definition, Fourth Edition* (see the Confor- X
   mance Rule above). The system must also provide all other components of an application's run-time environment that are included or referenced in the *ABI* specification.
8. The system must install packages using the formats and procedure described in the *ABI*, and must be capable of accepting installable software packages, either through physical media or through a network interface.

Base and Optional Components of the ABI

The *ABI* provides two levels of interface specification: Base and Optional. Base components of the *ABI* are required to be present in all *ABI*-conforming systems. Optional components may be absent on an ABI-conforming system, but, when they are present, they must conform to the specification given in the *ABI*. All components of the *ABI* are to be considered Base components unless they are explicitly described as Optional like the one that follows.

NOTE THE FACILITIES AND INTERFACES DESCRIBED IN THIS SECTION ARE OPTIONAL COMPONENTS OF THE *System V Application Binary Interface*.

This distinction is necessary because some *ABI* capabilities depend on the presence of hardware or other facilities that may not be present, such as integral graphics displays or network connections and hardware. The absence of such facilities does not prevent a System V-based system from conforming with the *ABI* specification, though it may prevent applications that need these facilities from running on those systems.

Evolution of the ABI Specification

The *System V Application Binary Interface* will evolve over time to address new technology and market requirements, and will be reissued at intervals of approximately three years. Each new edition of the specification is likely to contain extensions and additions that will increase the potential capabilities of applications that are written to conform to the *ABI*.

The *System V Application Binary Interface, Edition 4.1* includes certain elements X
marked as DEPRECATED. An interface, header, or command has been marked as X
DEPRECATED if it is specified as To Be Withdrawn in the *X/Open CAE* X
Specification, Issue 4.2, or if it is not present in the *X/Open CAE Specification, Issue* X
4.2 and is designated as Level 2 in the *System V Interface Definition, Fourth Edition*. X
Long term support for such functions can not be presumed.

Definitions of Terms

The following terms are used throughout this document.

- *ABI* or *System V ABI*: Refers to the specification that is the subject of this document, the *System V Application Binary Interface*. The *System V ABI* for a particular system is composed of the generic *ABI* and the *Processor-specific Supplement* for the processor used in the system.
- generic *System V ABI* or generic *ABI*: Consists of the processor-independent portions of the *System V Application Binary Interface*.
- *Processor-specific ABI* or *Processor-specific Supplement*: Consists of those portions of the *System V ABI* that are specific to a particular processor architecture. Together, the generic *ABI* and the appropriate *Processor-specific Supplement* comprise the *System V ABI* for systems employing a particular processor architecture.
- ABI-conforming system: A computer system that provides the binary system interface for application programs described in the *System V ABI*.
- ABI-conforming program: A program written to include only the system routines, commands, and other resources included in the *ABI*, and a program compiled into an executable file that has the formats and characteristics specified for such files in the *ABI*, and a program whose behavior complies with the rules given in the *ABI*.
- ABI-nonconforming program: A program which has been written to include system routines, commands, or other resources not included in the *ABI*, or a program which has been compiled into a format different from those specified in the *ABI*, or a program which does not behave as specified in the *ABI*.
- undefined behavior: Behavior that may vary from instance to instance or may change at some time in the future. Some undesirable programming practices are marked in the *ABI* as yielding undefined behavior.
- unspecified property: A property of an entity that is not explicitly included or referenced in this specification, and may change at some time in the future. In general, it is not good practice to make a program depend on an unspecified property.

DRAFT COPY
March 18, 1997
File: chap1
386:adm.book:sum

NOTE

Diffmarkings have been retained in the text of this book to indicate in which revisions of System V certain modifications were made to the *ABI*.

A "|" character in the right hand margin indicates a corrective change in the *ABI* made just after the Release 4 *ABI* was published.

An "E" character in the right hand margin indicates a change in the *ABI* made in UNIX System V Release 4.1.

A "G" character in the right hand margin indicates a change in the *ABI* made in UNIX System V Release 4.2.

A "M" character in the right hand margin indicates a change in the *ABI* made X
in UnixWare® 2.0. A "X" character in the right hand margin indicates a change in the *ABI* made to merge in XPG4.2 source API requirements.

2 SOFTWARE INSTALLATION

Software Installation and Packaging

Installation Media

This section of the *ABI* describes the media from which application software can be installed on all ABI-conforming systems. It includes the following characteristics of supported media:

- Physical Distribution Media and Formats: Specification of the physical media that may be used to distribute ABI-conforming application software.
- Media Format: Format of the software on the installation medium.
- Software Structure of the Physical Media: A functional description of the files contained on the physical media and their layout on the media.
- File Formats: The format and interpretation of installation data files.

Physical Distribution Media and Formats

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Media Format

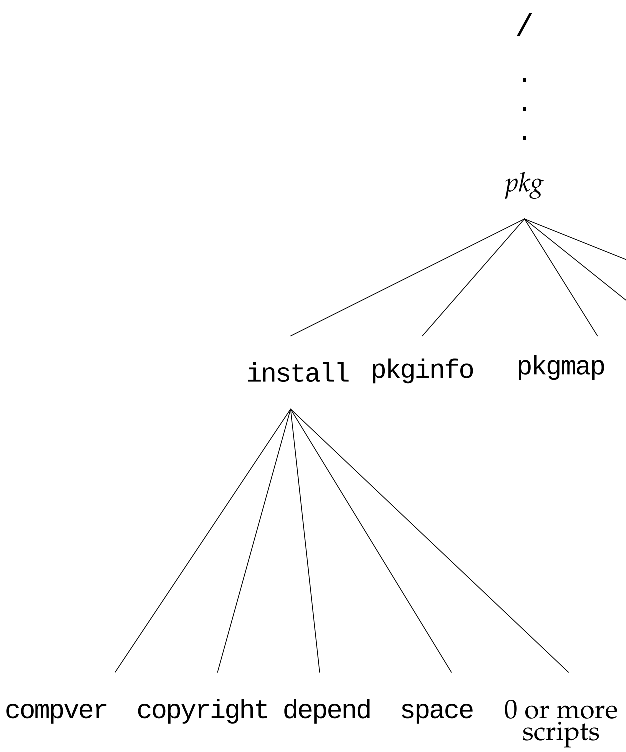
Packages are stored as a continuous data stream on the distribution media. The continuous data stream is valid for all media. The data stream can be created using the `dd` and `cpio` utilities.

Software Structure of the Physical Media

Add-on application software is bundled and installed in units called packages.
Multiple packages can be delivered on a single volume of media or a package can
span multiple volumes of media. A package that spans multiple volumes of
media must be the only package in the distribution. `cpio` always pads headers to E
a 512 byte boundary, and archives to an integral block size, set by the `-C` or `-B` E
option. Each archive on the media is extended to the block size specified by null E
padding.

Software to be bundled as a package must be organized as a file tree subdirectory as shown in Figure 2-1. The data stream from the distribution media is read onto disk into a file subtree of this format before the actual installation begins. Figure 2-2 shows the sequence of files in the data stream stored on distribution media.

Figure 2-1: Package File Tree Organization



The following is a brief description of files and directories shown in Figure 2-1. Entries in `constant width` type are standard file or directory names; the names of the other items are package-specific.

The following files are required:

- `pkginfo`: describes the package.
- `pkgmap`: describes each package object (files, directories, and so on).

The following files and directories are optional:

- `compver`: describes previous versions of the package with which this version is backward compatible.
- `copyright`: copyright notice for the package.
- `depend`: describes dependencies and incompatibilities across packages.
- `install`: contains optional package information files.
- package objects: executables, data files, and so on, belonging to the package.
- `reloc`: contains relocatable package objects (files whose pathnames on the target system are determined at the time of installation rather than at the time of package creation).
- `root`: contains non-relocatable package objects.
- `space`: describes space requirements beyond package objects.
- scripts: zero or more scripts to handle package-specific installation and removal requirements.

Figure 2-2: Data Stream File Layout for Distribution Media

header:	# PaCkAgE DaTaStReAm[: type] pkgA num_parts max_part_size [N ... N] pkgB num_parts max_part_size [N ... N] . . . # end of header
special information files (cpio archive):	pkgA/pkginfo pkgA/pkgmap pkgB/pkginfo pkgB/pkgmap . . .
package part (cpio archive):	pkgA/pkginfo pkgA/install/* pkgA/root.1/* pkgA/reloc.1/*
package part (cpio archive):	pkgA/pkginfo pkgA/root.2/* pkgA/reloc.2/*
	. . .
package part (cpio archive):	pkgB/pkginfo pkgB/install/* pkgB/root.1/* pkgB/reloc.1/*
	. . .

DRAFT COPY
March 18, 1997
File: chap2
386:adm.book:sum

The data stream format begins with a header containing a series of new-line terminated ASCII character strings. A special character string on the first line identifies the start of the header. It should be in the following format: E

 # PaCkAgE DaTaStReAm[: type]

When no *type* is defined, the data stream is continuous. Additional types of data streams may be specified in a processor-specific ABI supplement. Such additional types would deal with media-specific physical storage attributes, for example, record size, blocking factors, or alignment of package parts on the media.

Next are one or more special lines, one for each package in the distribution. Each line has the following format:

pkginstance num_parts max_part_size [N . . . N]

where:

- *pkginstance* is the package identifier made up of the package abbreviation (described later as the `PKG` parameter in the `pkginfo` file) and an optional suffix.
- *num_parts* is the number of parts into which the package is divided. As shown in Figure 2-2, a part is a collection of files contained in a `cpio` archive. A part is the atomic unit by which a package is processed. Each part must fit entirely on a distribution media volume, that is, a part cannot cross volumes. A developer chooses the criteria for grouping files into a part.
- *max_part_size* is the maximum number of 512-byte blocks consumed by a single part of the package.
- *N . . . N* are optional fields to indicate the number of parts stored on the sequential volumes of media which contain the package. For example,

  pkgA 6 2048 2 3 1

  indicates that *pkgA* consists of six parts. The largest part is 2048 512-byte blocks in size. The first two parts exist on the first volume, the next three parts exist on the second volume, and the last part exists on the third, and last, volume. These fields only apply where multiple volumes are needed to distribute a package. Otherwise, these fields should not appear and any process reading the header should assume that all parts of the package reside on the current volume. When these fields are used, there can only be one package in the data stream.

A special new-line terminated character string on a separate line identifies the end of the header. It should be in the following format: E

 # end of header

The header must be padded to a 512-byte boundary.

Following the header is a `cpio` archive containing special information files for each package. The rest of the data stream consists of package parts. Note that a given package may consist of one or more parts, that is, `cpio` archives. A packages which requires multiple parts must meet the following conditions:

- Each part must contain the entire `pkginfo` file.
- Each part includes its own `root` and `reloc` directories and these directories are numbered. For example, a package that requires *n* parts has `root.1` and `reloc.1` through `root.`*n* and `reloc.`*n*. *n* is limited to eight digits.
- The `install` directory and its contents are only provided in the first part.

File Formats

pkginfo File

The `pkginfo` file describes the package, as a whole. Each line is of the form *parameter=value*. The list below describes defined parameters. These parameters can be retrieved via `pkginfo`(AS_CMD) and/or have a specific meaning to the package installation and removal commands (`pkgadd`(AS_CMD) and `pkgrm`(AS_CMD)). No specific ordering of parameters is required. Lines beginning with # are treated as comments.

- `PKG`: Package abbreviation, limited to 9 characters. The first character must be alphabetic; remaining characters can be alphabetic, numeric, or the characters `+` and `-`. `install`, `new`, and `all` are not valid package abbreviations.
- `NAME`: Package name, limited to 256 ASCII characters.
- `ARCH`: A comma-separated list of identifiers that specify the architecture(s) on which the package can run. Each architecture identifier is limited to 16 ASCII characters; the character `','` is invalid.
- `VERSION`: Version identifier, limited to 256 ASCII characters. The first character can not be a left parenthesis due to the syntax of the `depend` file. This identifier is vendor-specific information.
- `DESC`: Descriptive text, limited to 256 ASCII characters.
- `VENDOR`: Vendor identifier, limited to 256 ASCII characters.
- `HOTLINE`: Phone number or mailing address where further information may be requested or bugs reported. The value of this parameter is limited to 256 ASCII characters.
- `EMAIL`: An electronic mail address, with the same purpose and length limitation as the `HOTLINE` parameter.
- `VSTOCK`: Vendor stock number, limited to 256 ASCII characters.
- `CATEGORY`: A comma-separated list of categories to which the package belongs. Add-on software packages must state their membership in the `application` category. Users can request information on all packages in specific categories via `pkginfo`(AS_CMD). Category identifiers are case-insensitive and are limited to 16 alphanumeric characters, excluding the space and comma characters.

- PSTAMP: Production stamp, limited to 256 ASCII characters.
- ISTATES: Space-separated list of valid run-levels during which this package can be installed. Run-levels are integers in the range 0 through 6, s and S [see *System V Interface Definition, Fourth Edition*, init(AS_CMD)].
- RSTATES: Same as ISTATES, but applies to package removal.
- ULIMIT: Temporary file size limit to use during installation of this package [see getrlimit].
- INTONLY: A parameter that indicates a package can only be installed interactively by the pkgadd(AS_CMD) command. If this parameter is supplied and set to any non-null value (for example, y or yes), the package can only be installed interactively. Otherwise, the package can be installed in a noninteractive mode using pkgask(AS_CMD) or pkgadd with the -n option.
- MAXINST: An integer that specifies the maximum number of instances of this package that can be installed on a system at one time. If this parameter is not supplied, a default of 1 is used (at most one copy of the package can be installed on a system at any time).
- BASEDIR: The pathname to a default directory where ''relocatable'' files may be installed. If BASEDIR is blank and the basedir variable in the admin file is set to ''default,'' the package is not considered relocatable. In this case, if files have relative pathnames, package installation will fail. An administrator can override BASEDIR by setting the basedir variable in the admin file.
- CLASSES: A space-separated list of package object classes to be installed. Every package object is assigned to one class (in the pkgmap file). Class assignment can be used to control (for example, based on administrator input at the time of installation) which objects are installed and to provide specific actions to be taken to install or remove them. The value of this parameter can be overridden at the time of installation.

PKG, NAME, ARCH, VERSION, CATEGORY and BASEDIR are mandatory parameters, that is, package developers must supply them. The rest are optional. E

Other parameters may be defined in the pkginfo file. If they are used as part of package object pathnames in the pkgmap file, the value supplied in the pkginfo file is used as a default and may be overridden at the time of installation. Other parameters will be made part of the environment in which installation scripts execute.

The pkgmap File

Each line in the `pkgmap` file describes a package object or installation script (with the exception of one `pkgmap` entry, described at the end of this section). The following list describes the space-separated fields in each `pkgmap` entry; their order must be the same as in the list. Lines in the file that begin with # are ignored.

- *part*: A positive integer that indicates the part of a multi-part package in which this pathname resides.
- *ftype*: A one-character file type identifier from the set below:

`f`	a file
`d`	a directory
`i`	an installation or removal script
`l`	a linked file
`s`	a symbolic link
`p`	a named pipe
`b`	a block special device
`c`	a character special device
`e`	a file installed or removed by editing
`v`	a volatile file, whose contents are expected to change as the package is used on the system
`x`	an exclusive directory, which should contain only files installed as part of this or some other standard format package

- *class*: A package object class identifier, limited to 12 characters. `none` is used to specify no class membership. This field is not specified for files whose *ftype* is `i`.
- *pathname*: The pathname describing the location of the file on the target machine. For files of *ftype*, `l` or `s`, *pathname* must be of the form *path1=path2*, specifying the destination (*path1*) and source (*path2*) files to be linked. Special characters, such as an equal sign (=), are included in pathnames by surrounding the entire pathname in single quotes (as in, for example, `'/usr/lib/~='`).

  The *pathname* may contain variables which support relocation of the file. A $*parameter* may be embedded in the pathname structure. `$BASEDIR` can be used to identify the parent directories of the path hierarchy, making the entire package easily relocatable. Default values for *parameter* and `BASEDIR` must be supplied in the `pkginfo` file and may be overridden at installation.

- *major*: The major device number, applicable to files whose *ftype* is `b` or `c`.
- *minor*: The minor device number, applicable to files whose *ftype* is `b` or `c`.
- *mode*: The octal mode of the file. `?` indicates that no particular mode is required. This field is not provided for files whose *ftype* is `l`, `s` or `i`.
- *owner*: The *uid* of the owner of the file; it must be a legal *uid*. `?` indicates that no particular owner is required. This field is not provided for files whose *ftype* is `l`, `s` or `i`.
- *group*: The group to which the file belongs, limited to 14 characters. `?` indicates that no particular group is required. This field is not provided for files whose *ftype* is `l`, `s` or `i`.
- *size*: The file size in bytes. This field is not provided for files whose *ftype* is `d`, `x`, `p`, `b`, `c`, `s` or `l`.
- *cksum*: The checksum of the file contents, as calculated by `sum`. This field is not provided for files whose *ftype* is `d`, `x`, `p`, `b`, `c`, `s` or `l`.
- *modtime*: The time of last modification as reported by the function `stat`. This field is not provided for files whose *ftype* is `d`, `x`, `p`, `b`, `c` or `l`.

An additional line in the `pkgmap` file, beginning with a colon, provides information about the media on which the package is distributed. Its format is:

`:`*number_of_parts maximum_part_size*

number_of_parts specifies the number of parts which compose this package. *maximum_part_size* specifies the size, in 512-byte blocks, of the largest part.

The copyright File

The contents of the `copyright` file will be displayed on *stdout* at the time of installation; there are no format requirements.

The space File

The `space` file describes disk block and inode requirements for the package beyond files listed in the `pkgmap` file and provided on the media. Each line in the file contains the following three space-separated fields:

- *pathname*: A directory name. Naming conventions (with respect to indicating relocatability) are the same as for the *pathname* field in the `pkgmap` file.
- *blocks*: The number of 512-byte blocks required.

- *inodes*: The number of distinct files required.

The depend File

The `depend` file describes dependencies across packages. The format of each entry is as follows:

> *type pkg name*
> (*arch*)*version*

The following field definitions and rules apply:

- *type*: Describes the type of dependency:

  `P` a prerequisite for installation

  `I` an incompatibility

  `R` reverse dependency (the referenced package depends on this package)

- *pkg*: The package abbreviation, as defined in the `pkginfo` file.
- *name*: The package name, as defined in the `pkginfo` file.
- *arch*: The package architecture, as defined in the `pkginfo` file.
- *version*: The package version, as defined in the `pkginfo` file.
- There may be zero or more (*arch*)*version* lines.
- (*arch*)*version* lines must begin with white space.

The compver File

The `compver` file specifies previous versions of the package with which this version is backward compatible. It is used for dependency checking across packages, in conjunction with the `depend` file. It consists of version identifiers, as defined in the `pkginfo` file, one per line.

Installation and Removal Scripts

This section describes installation and removal scripts that may be provided by a package to meet package-specific needs. Scripts are executed by the command `sh` and therefore must be either shell scripts or executable programs. In the case of recovery from an interrupted installation they may be re-executed; they should be written so that multiple invocations produce the same results as a single invocation.

NOTE Since substantive differences exist between the shell traditionally provided by System V systems and the new *X/Open CAE Specification, Issue 4.2* shell, and since many shell scripts are liable to be affected by these differences, it is recommended that installation scripts and any other shell scripts that have stringent compatibility requirements continue to use the traditional System V shell. New shell scripts may wish to take advantage of the capabilities of the *X/Open CAE Specification, Issue 4.2* shell. X X X X X

The request Script

The `request` script, if provided, is the first script executed at the time of package installation. Its purpose is to interact with the user and modify details of the installation process as a result of this interaction. The script writes shell variable assignments to the file named by its only argument. It is executed with *uid* `root` E
and *gid* `sys`. `stdin`, `stdout` and `stderr` are all attached to the controlling terminal. The following variables are defined as part of the installation procedure and may be set by the `request` script: E

- CLASSES: As defined in the `pkginfo` file.

Procedure Scripts

Four scripts may be provided by a package to handle package-specific requirements - `preinstall`, `postinstall`, `preremove` and `postremove`.

The following constraints apply to these procedure scripts:

- When the script is executed, `stdin` is attached to the controlling terminal; `stdout` and `stderr` are attached to the controlling terminal. M
- Each pathname created or modified by a procedure script during installation or removal, and which should be considered part of the package, must be logically added or removed from the package via the `installf`(AS_CMD) or `removef`(AS_CMD) commands.
- Procedure scripts are executed with *uid* `root` and *gid* `sys`. M

Class Scripts

Class scripts provide non-standard installation and removal actions for classes of package objects (the membership of objects in a class is specified in the `pkgmap` file). The following constraints apply:

- Each class included in the value of the CLASSES parameter is installed, in the order in which they appear in that parameter. Objects in class none are installed first.

- If an object belongs to class `none` or no class script is provided for the class, the object is copied from the medium to the target system during installation, and removed during removal.
- Class script names are of the form *operation.class*, where *operation* is either `i` (for install) or `r` (for remove), and *class* is the class name, limited to 12 characters. Class names beginning with `0` are reserved.
- Class scripts will execute as *uid* `root` and *gid* `sys`. M
- During installation, the script is executed with either no arguments or the single argument `ENDOFCLASS`. `stdin` contains a list of filename pairs, of the form *source_pathname destination_pathname*. The *source_pathname* parameter is either a pathname on the medium or `/dev/null` to indicate there is no file to copy from the medium (for example, a directory). *destination_pathname* is the target pathname. Only files that are members of the class and are not identical to files already on the system are provided to the script.
- The script is invoked with the single argument `ENDOFCLASS` to indicate that there are no more files belonging to the class once end of file is reached on `stdin` during the current invocation of the script.
- During removal, the script is executed with no arguments. `stdin` contains a list of filenames, including all members of the class except those shared by other installed packages whose *ftype* in the `pkgmap` file is something other than `e`.
- Two standard classes are defined: `build` and `sed`. The name of the file on the medium is the name of the file on the target system to be modified. |

  For the `sed` class, the file provided on the medium contains the command `sed` instructions. Lines of the format `!install` and `!remove` mark the beginning of instructions which apply to installation and removal, respectively. The file on the target system will be modified by the output of `sed`, using the provided data. |

  A file that belongs to the `build` class is executed with a single argument, `install` or `remove`. Its output (on `stdout`) is written to the file it references on the target system.

Exit Codes Used by Scripts

Scripts shall exit with an additive combination of one of the first four and one of the last two exit codes listed below: E

`0`: successful execution

- `1`: fatal error
- `2`: warning
- `3`: interruption
- `10`: reboot after installation of all packages
- `20`: reboot after installation of this package

For example, the exit code for a script resulting in a warning condition and that requires an immediate reboot is `22`.

File Tree for Add-on Software

`/opt`, `/var/opt` and `/etc/opt` are reserved in the file tree for the installation of application software packages. Each add-on software package should adhere to the following rules:

- Static package objects should be installed in `/opt/`*pkg*, where *pkg* is the package abbreviation or instance.
- Package objects that change in normal operations (for example, log and spool files) should be installed in `/var/opt/`*pkg*.
- Machine-specific configuration files should be installed in `/etc/opt/`*pkg*.
- Executables that are directly invoked by users should be installed in `/opt/`*pkg*/*bin*.
- Only package objects that must reside in specific locations within the system file tree in order to function properly (for example, special files in `/dev`) should be installed in those locations.

Commands That Install, Remove and Access Packages

The following commands and library routines are used to install and remove packages and to retrieve information about installed packages. They will be included in every ABI-conforming system, and are defined in the *System V Interface Definition, Fourth Edition*.

`pkgadd`(AS_CMD):	installs packages
`pkgrm`(AS_CMD):	removes packages
`pkgchk`(AS_CMD):	checks installed packages
`pkginfo`(AS_CMD):	display information about packages
`pkgask`(AS_CMD):	runs the request script and stores output for later use
`installf`(AS_CMD):	associates an installed file with a package
`removef`(AS_CMD):	removes a file's association with a package
`pkgparam`(AS_CMD):	display the values of parameters defined by the package in the `pkginfo` file

3 LOW-LEVEL SYSTEM INFORMATION

Introduction

This chapter defines low-level system information, much of which is processor-specific. It gives the constraints imposed by the system on application programs, and it describes how application programs use operating system services. ANSI C serves as the ABI reference programming language. By defining the implementation of C data types, the ABI can give precise system interface information without resorting to assembly language. Giving C language bindings for system services does *not* preclude bindings for other programming languages. Moreover, the examples given here are not intended to specify the C language available on the processor.

NOTE According to ANSI C, a bit-field may have type `int`, `unsigned int`, or `signed int`. The C language used in this ABI allows bit-fields of type `char`, `short`, `int`, and `long` (plus their `signed` and `unsigned` variants), and of type `enum`.

This chapter's major sections discuss the following topics.

- *Character representations.* This section defines the standard character set used for external files that should be portable among systems.
- *Machine interface.* This section describes the processor architecture available to programs. It also defines the reference language data types, giving the foundation for system interface specifications.
- *Function calling sequence.* The standard function calling sequence accommodates the operating system interface, including system calls, signals, and stack management.
- *Operating system interface.* This section describes the operating system mechanisms that are visible to application programs (such as signals, process initialization, and so on).
- *Coding examples.* Finally, some C code fragments show how programming languages may implement some fundamental operations under the ABI specifications. These examples are not intended to give the only implementations, the optimal implementations, nor requirements for implementations.

Character Representations

Several external file formats represent control information with characters (see ''Archive File'' in Chapter 7, for example). These single-byte characters use the 7-bit ASCII character set. In other words, when the ABI mentions character constants, such as `'/'` or `'\n'`, their numerical values should follow the 7-bit ASCII guidelines. For the previous character constants, the single-byte values would be 47 and 10, respectively.

Character values outside the range of 0 to 127 may occupy one or more bytes, according to the character encoding. Applications can control their own character sets, using different character set extensions for different languages as appropriate. Although ABI-conformance does not restrict the character sets, they generally should follow some simple guidelines.

- Character values between 0 and 127 should correspond to the 7-bit ASCII code. That is, character sets with encodings above 127 should include the 7-bit ASCII code as a subset.
- Multibyte character encodings with values above 127 should contain only bytes with values outside the range of 0 to 127. That is, a character set that uses more than one byte per character should not ''embed'' a byte resembling a 7-bit ASCII character within a multibyte, non-ASCII character.
- Multibyte characters should be self-identifying. That allows, for example, any multibyte character to be inserted between any pair of multibyte characters, without changing the characters' interpretations.

These cautions are particularly relevant for multilingual applications.

Machine Interface (Processor-Specific)

NOTE

This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Function Calling Sequence (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Operating System Interface (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Coding Examples (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

4 OBJECT FILES

DRAFT COPY
March 18, 1997
File: Cchap4
386:adm.book:sum

Introduction

This chapter describes the object file format, called ELF (Executable and Linking Format). There are three main types of object files.

- A *relocatable file* holds code and data suitable for linking with other object files to create an executable or a shared object file.
- An *executable file* holds a program suitable for execution; the file specifies how the function `exec` creates a program's process image. X
- A *shared object file* holds code and data suitable for linking in two contexts. First, the link editor [see `ld`(SD_CMD)] may process it with other relocatable and shared object files to create another object file. Second, the dynamic linker combines it with an executable file and other shared objects to create a process image.

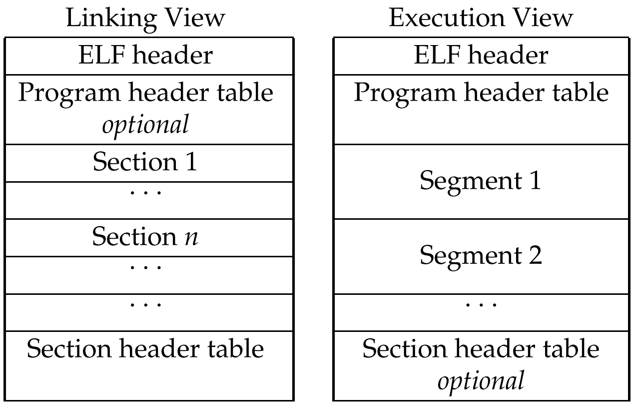
Created by the assembler and link editor, object files are binary representations of programs intended to execute directly on a processor. Programs that require other abstract machines, such as shell scripts, are excluded.

After the introductory material, this chapter focuses on the file format and how it pertains to building programs. Chapter 5 also describes parts of the object file, concentrating on the information necessary to execute a program.

File Format

Object files participate in program linking (building a program) and program execution (running a program). For convenience and efficiency, the object file format provides parallel views of a file's contents, reflecting the differing needs of these activities. Figure 4-1 shows an object file's organization.

Figure 4-1: Object File Format



An *ELF header* resides at the beginning and holds a ''road map'' describing the file's organization. *Sections* hold the bulk of object file information for the linking view: instructions, data, symbol table, relocation information, and so on. Descriptions of special sections appear later in the chapter. Chapter 5 discusses *segments* and the program execution view of the file.

A *program header table*, if present, tells the system how to create a process image. Files used to build a process image (execute a program) must have a program header table; relocatable files do not need one. A *section header table* contains information describing the file's sections. Every section has an entry in the table; each entry gives information such as the section name, the section size, and so on. Files used during linking must have a section header table; other object files may or may not have one.

NOTE Although the figure shows the program header table immediately after the ELF header, and the section header table following the sections, actual files may differ. Moreover, sections and segments have no specified order. Only the ELF header has a fixed position in the file.

Data Representation

As described here, the object file *format* supports various processors with 8-bit bytes and 32-bit architectures. Nevertheless, it is intended to be extensible to larger (or smaller) architectures. Object files therefore represent some control data with a machine-independent format, making it possible to identify object files and interpret their contents in a common way. Remaining data in an object file use the encoding of the target processor, regardless of the machine on which the file was created.

Figure 4-2: 32-Bit Data Types

Name	Size	Alignment	Purpose
`Elf32_Addr`	4	4	Unsigned program address
`Elf32_Half`	2	2	Unsigned medium integer
`Elf32_Off`	4	4	Unsigned file offset
`Elf32_Sword`	4	4	Signed large integer
`Elf32_Word`	4	4	Unsigned large integer
`unsigned char`	1	1	Unsigned small integer

All data structures that the object file format defines follow the ''natural'' size and alignment guidelines for the relevant class. If necessary, data structures contain explicit padding to ensure 4-byte alignment for 4-byte objects, to force structure sizes to a multiple of 4, and so on. Data also have suitable alignment from the beginning of the file. Thus, for example, a structure containing an `Elf32_Addr` member will be aligned on a 4-byte boundary within the file.

For portability reasons, ELF uses no bit-fields.

ELF Header

Some object file control structures can grow, because the ELF header contains their actual sizes. If the object file format changes, a program may encounter control structures that are larger or smaller than expected. Programs might therefore ignore ''extra'' information. The treatment of ''missing'' information depends on context and will be specified when and if extensions are defined.

Figure 4-3: ELF Header

```
#define EI_NIDENT  16

typedef struct {
         unsigned char      e_ident[EI_NIDENT];
         Elf32_Half         e_type;
         Elf32_Half         e_machine;
         Elf32_Word         e_version;
         Elf32_Addr         e_entry;
         Elf32_Off          e_phoff;
         Elf32_Off          e_shoff;
         Elf32_Word         e_flags;
         Elf32_Half         e_ehsize;
         Elf32_Half         e_phentsize;
         Elf32_Half         e_phnum;
         Elf32_Half         e_shentsize;
         Elf32_Half         e_shnum;
         Elf32_Half         e_shstrndx;
} Elf32_Ehdr;
```

`e_ident` The initial bytes mark the file as an object file and provide machine-independent data with which to decode and interpret the file's contents. Complete descriptions appear below, in ''ELF Identification.''

`e_type` This member identifies the object file type.

Name	Value	Meaning
`ET_NONE`	`0`	No file type
`ET_REL`	`1`	Relocatable file
`ET_EXEC`	`2`	Executable file
`ET_DYN`	`3`	Shared object file
`ET_CORE`	`4`	Core file
`ET_LOPROC`	`0xff00`	Processor-specific
`ET_HIPROC`	`0xffff`	Processor-specific

Although the core file contents are unspecified, type `ET_CORE` is reserved to mark the file. Values from `ET_LOPROC` through `ET_HIPROC` (inclusive) are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them. Other values are reserved and will be assigned to new object file types as necessary.

`e_machine` This member's value specifies the required architecture for an individual file.

Name	Value	Meaning	
`EM_NONE`	`0`	No machine	
`EM_M32`	`1`	AT&T WE 32100	
`EM_SPARC`	`2`	SPARC	
`EM_386`	`3`	Intel 80386	
`EM_68K`	`4`	Motorola 68000	
`EM_88K`	`5`	Motorola 88000	
`EM_860`	`7`	Intel 80860	
`EM_MIPS`	`8`	MIPS RS3000 Big-Endian	E
`EM_MIPS_RS4_BE`	`10`	MIPS RS4000 Big-Endian	E
`RESERVED`	`11-16`	Reserved for future use	E

Other values are reserved and will be assigned to new machines as necessary. Processor-specific ELF names use the machine name to distinguish them. For example, the flags mentioned below use the prefix `EF_`; a flag named `WIDGET` for the `EM_XYZ` machine would be called `EF_XYZ_WIDGET`.

`e_version` This member identifies the object file version.

Name	Value	Meaning
EV_NONE	0	Invalid version
EV_CURRENT	1	Current version

The value 1 signifies the original file format; extensions will create new versions with higher numbers. The value of EV_CURRENT, though given as 1 above, will change as necessary to reflect the current version number.

e_entry — This member gives the virtual address to which the system first transfers control, thus starting the process. If the file has no associated entry point, this member holds zero.

e_phoff — This member holds the program header table's file offset in bytes. If the file has no program header table, this member holds zero.

e_shoff — This member holds the section header table's file offset in bytes. If the file has no section header table, this member holds zero.

e_flags — This member holds processor-specific flags associated with the file. Flag names take the form EF_*machine_flag*. See ''Machine Information'' in the processor supplement for flag definitions.

e_ehsize — This member holds the ELF header's size in bytes.

e_phentsize — This member holds the size in bytes of one entry in the file's program header table; all entries are the same size.

e_phnum — This member holds the number of entries in the program header table. Thus the product of e_phentsize and e_phnum gives the table's size in bytes. If a file has no program header table, e_phnum holds the value zero.

e_shentsize — This member holds a section header's size in bytes. A section header is one entry in the section header table; all entries are the same size.

e_shnum — This member holds the number of entries in the section header table. Thus the product of e_shentsize and e_shnum gives the section header table's size in bytes. If a file has no section header table, e_shnum holds the value zero.

e_shstrndx — This member holds the section header table index of the entry associated with the section name string table. If the file has no section name string table, this member holds the value SHN_UNDEF. See ''Sections'' and ''String Table'' below for more information.

ELF Identification

As mentioned above, ELF provides an object file framework to support multiple processors, multiple data encodings, and multiple classes of machines. To support this object file family, the initial bytes of the file specify how to interpret the file, independent of the processor on which the inquiry is made and independent of the file's remaining contents.

The initial bytes of an ELF header (and an object file) correspond to the `e_ident` member.

Figure 4-4: `e_ident[ ]` **Identification Indexes**

Name	Value	Purpose
EI_MAG0	0	File identification
EI_MAG1	1	File identification
EI_MAG2	2	File identification
EI_MAG3	3	File identification
EI_CLASS	4	File class
EI_DATA	5	Data encoding
EI_VERSION	6	File version
EI_PAD	7	Start of padding bytes
EI_NIDENT	16	Size of `e_ident[]`

These indexes access bytes that hold the following values.

`EI_MAG0` to `EI_MAG3`

A file's first 4 bytes hold a ''magic number,'' identifying the file as an ELF object file.

Name	Value	Position
ELFMAG0	0x7f	e_ident[EI_MAG0]
ELFMAG1	'E'	e_ident[EI_MAG1]
ELFMAG2	'L'	e_ident[EI_MAG2]
ELFMAG3	'F'	e_ident[EI_MAG3]

`EI_CLASS` The next byte, `e_ident[EI_CLASS]`, identifies the file's class, or capacity.

Name	Value	Meaning
`ELFCLASSNONE`	`0`	Invalid class
`ELFCLASS32`	`1`	32-bit objects
`ELFCLASS64`	`2`	64-bit objects

The file format is designed to be portable among machines of various sizes, without imposing the sizes of the largest machine on the smallest. Class `ELFCLASS32` supports machines with files and virtual address spaces up to 4 gigabytes; it uses the basic types defined above.

Class `ELFCLASS64` is reserved for 64-bit architectures. Its appearance here shows how the object file may change, but the 64-bit format is otherwise unspecified. Other classes will be defined as necessary, with different basic types and sizes for object file data.

`EI_DATA` Byte `e_ident[EI_DATA]` specifies the data encoding of the processor-specific data in the object file. The following encodings are currently defined.

Name	Value	Meaning
`ELFDATANONE`	`0`	Invalid data encoding
`ELFDATA2LSB`	`1`	See below
`ELFDATA2MSB`	`2`	See below

More information on these encodings appears below. Other values are reserved and will be assigned to new encodings as necessary.

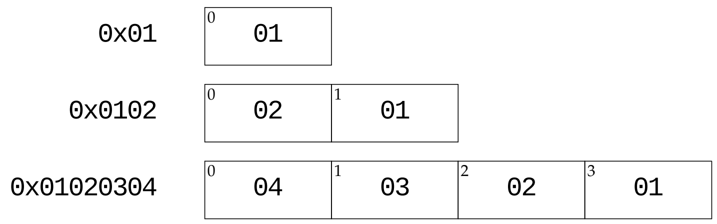
`EI_VERSION` Byte `e_ident[EI_VERSION]` specifies the ELF header version number. Currently, this value must be `EV_CURRENT`, as explained above for `e_version`.

`EI_PAD` This value marks the beginning of the unused bytes in `e_ident`. These bytes are reserved and set to zero; programs that read object files should ignore them. The value of `EI_PAD` will change in the future if currently unused bytes are given meanings.

A file's data encoding specifies how to interpret the basic objects in a file. As described above, class `ELFCLASS32` files use objects that occupy 1, 2, and 4 bytes. Under the defined encodings, objects are represented as shown below. Byte numbers appear in the upper left corners.

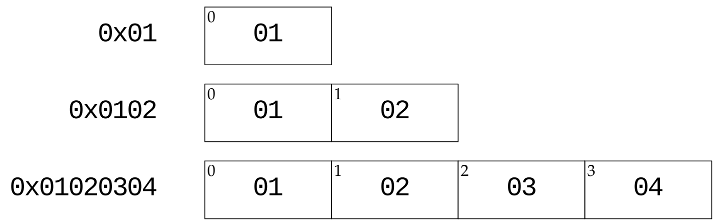
Encoding ELFDATA2LSB specifies 2's complement values, with the least significant byte occupying the lowest address.

Figure 4-5: Data Encoding ELFDATA2LSB



Encoding ELFDATA2MSB specifies 2's complement values, with the most significant byte occupying the lowest address.

Figure 4-6: Data Encoding ELFDATA2MSB



Machine Information (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Sections

An object file's section header table lets one locate all the file's sections. The section header table is an array of Elf32_Shdr structures as described below. A section header table index is a subscript into this array. The ELF header's e_shoff member gives the byte offset from the beginning of the file to the section header table; e_shnum tells how many entries the section header table contains; e_shentsize gives the size in bytes of each entry.

Some section header table indexes are reserved; an object file will not have sections for these special indexes.

Figure 4-7: Special Section Indexes

Name	Value
SHN_UNDEF	0
SHN_LORESERVE	0xff00
SHN_LOPROC	0xff00
SHN_HIPROC	0xff1f
SHN_ABS	0xfff1
SHN_COMMON	0xfff2
SHN_HIRESERVE	0xffff

SHN_UNDEF This value marks an undefined, missing, irrelevant, or otherwise meaningless section reference. For example, a symbol ''defined'' relative to section number SHN_UNDEF is an undefined symbol.

NOTE Although index 0 is reserved as the undefined value, the section header table contains an entry for index 0. That is, if the e_shnum member of the ELF header says a file has 6 entries in the section header table, they have the indexes 0 through 5. The contents of the initial entry are specified later in this section.

SHN_LORESERVE This value specifies the lower bound of the range of reserved indexes.

SHN_LOPROC through SHN_HIPROC
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

SHN_ABS	This value specifies absolute values for the corresponding reference. For example, symbols defined relative to section number SHN_ABS have absolute values and are not affected by relocation.
SHN_COMMON	Symbols defined relative to this section are common symbols, such as FORTRAN COMMON or unallocated C external variables.
SHN_HIRESERVE	This value specifies the upper bound of the range of reserved indexes. The system reserves indexes between SHN_LORESERVE and SHN_HIRESERVE, inclusive; the values do not reference the section header table. That is, the section header table does *not* contain entries for the reserved indexes.

Sections contain all information in an object file, except the ELF header, the program header table, and the section header table. Moreover, object files' sections satisfy several conditions.

- Every section in an object file has exactly one section header describing it. Section headers may exist that do not have a section.
- Each section occupies one contiguous (possibly empty) sequence of bytes within a file.
- Sections in a file may not overlap. No byte in a file resides in more than one section.
- An object file may have inactive space. The various headers and the sections might not ''cover'' every byte in an object file. The contents of the inactive data are unspecified.

A section header has the following structure.

Figure 4-8: Section Header

```
typedef struct {
        Elf32_Word      sh_name;
        Elf32_Word      sh_type;
        Elf32_Word      sh_flags;
        Elf32_Addr      sh_addr;
        Elf32_Off       sh_offset;
        Elf32_Word      sh_size;
        Elf32_Word      sh_link;
        Elf32_Word      sh_info;
        Elf32_Word      sh_addralign;
        Elf32_Word      sh_entsize;
} Elf32_Shdr;
```

`sh_name` This member specifies the name of the section. Its value is an index into the section header string table section [see ''String Table'' below], giving the location of a null-terminated string.

`sh_type` This member categorizes the section's contents and semantics. Section types and their descriptions appear below.

`sh_flags` Sections support 1-bit flags that describe miscellaneous attributes. Flag definitions appear below.

`sh_addr` If the section will appear in the memory image of a process, this member gives the address at which the section's first byte should reside. Otherwise, the member contains 0.

`sh_offset` This member's value gives the byte offset from the beginning of the file to the first byte in the section. One section type, `SHT_NOBITS` described below, occupies no space in the file, and its `sh_offset` member locates the conceptual placement in the file.

`sh_size` This member gives the section's size in bytes. Unless the section type is `SHT_NOBITS`, the section occupies `sh_size` bytes in the file. A section of type `SHT_NOBITS` may have a non-zero size, but it occupies no space in the file.

`sh_link` This member holds a section header table index link, whose interpretation depends on the section type. A table below describes the values.

sh_info — This member holds extra information, whose interpretation depends on the section type. A table below describes the values.

sh_addralign — Some sections have address alignment constraints. For example, if a section holds a doubleword, the system must ensure doubleword alignment for the entire section. That is, the value of sh_addr must be congruent to 0, modulo the value of sh_addralign. Currently, only 0 and positive integral powers of two are allowed. Values 0 and 1 mean the section has no alignment constraints.

sh_entsize — Some sections hold a table of fixed-size entries, such as a symbol table. For such a section, this member gives the size in bytes of each entry. The member contains 0 if the section does not hold a table of fixed-size entries.

A section header's sh_type member specifies the section's semantics.

Figure 4-9: Section Types, sh_type

Name	Value
SHT_NULL	0
SHT_PROGBITS	1
SHT_SYMTAB	2
SHT_STRTAB	3
SHT_RELA	4
SHT_HASH	5
SHT_DYNAMIC	6
SHT_NOTE	7
SHT_NOBITS	8
SHT_REL	9
SHT_SHLIB	10
SHT_DYNSYM	11
SHT_LOPROC	0x70000000
SHT_HIPROC	0x7fffffff
SHT_LOUSER	0x80000000
SHT_HIUSER	0xffffffff

SHT_NULL — This value marks the section header as inactive; it does not have an associated section. Other members of the section header have undefined values.

SHT_PROGBITS — The section holds information defined by the program, whose format and meaning are determined solely by the program.

SHT_SYMTAB and SHT_DYNSYM — These sections hold a symbol table. Currently, an object file may have only one section of each type, but this restriction may be relaxed in the future. Typically, SHT_SYMTAB provides symbols for link editing, though it may also be used for dynamic linking. As a complete symbol table, it may contain many symbols unnecessary for dynamic linking. Consequently, an object file may also contain a SHT_DYNSYM section, which holds a minimal set of dynamic linking symbols, to save space. See ''Symbol Table'' below for details.

SHT_STRTAB — The section holds a string table. An object file may have multiple string table sections. See ''String Table'' below for details.

SHT_RELA — The section holds relocation entries with explicit addends, such as type Elf32_Rela for the 32-bit class of object files. An object file may have multiple relocation sections. See ''Relocation'' below for details.

SHT_HASH — The section holds a symbol hash table. All objects participating in dynamic linking must contain a symbol hash table. Currently, an object file may have only one hash table, but this restriction may be relaxed in the future. See ''Hash Table'' in Chapter 5 for details.

SHT_DYNAMIC — The section holds information for dynamic linking. Currently, an object file may have only one dynamic section, but this restriction may be relaxed in the future. See ''Dynamic Section'' in Chapter 5 for details.

SHT_NOTE — The section holds information that marks the file in some way. See ''Note Section'' in Chapter 5 for details.

SHT_NOBITS — A section of this type occupies no space in the file but otherwise resembles SHT_PROGBITS. Although this section contains no bytes, the sh_offset member contains the conceptual file offset.

SHT_REL — The section holds relocation entries without explicit addends, such as type Elf32_Rel for the 32-bit class of object files. An object file may have multiple relocation sections. See ''Relocation'' below for details.

SHT_SHLIB — This section type is reserved but has unspecified semantics. Programs that contain a section of this type do not conform to the ABI.

SHT_LOPROC through SHT_HIPROC
: Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

SHT_LOUSER — This value specifies the lower bound of the range of indexes reserved for application programs.

SHT_HIUSER — This value specifies the upper bound of the range of indexes reserved for application programs. Section types between SHT_LOUSER and SHT_HIUSER may be used by the application, without conflicting with current or future system-defined section types.

Other section type values are reserved. As mentioned before, the section header for index 0 (SHN_UNDEF) exists, even though the index marks undefined section references. This entry holds the following.

Figure 4-10: Section Header Table Entry: Index 0

Name	Value	Note
sh_name	0	No name
sh_type	SHT_NULL	Inactive
sh_flags	0	No flags
sh_addr	0	No address
sh_offset	0	No file offset
sh_size	0	No size
sh_link	SHN_UNDEF	No link information
sh_info	0	No auxiliary information
sh_addralign	0	No alignment
sh_entsize	0	No entries

A section header's sh_flags member holds 1-bit flags that describe the section's attributes. Defined values appear below; other values are reserved.

Figure 4-11: Section Attribute Flags, `sh_flags`

Name	Value
SHF_WRITE	0x1
SHF_ALLOC	0x2
SHF_EXECINSTR	0x4
SHF_MASKPROC	0xf0000000

If a flag bit is set in `sh_flags`, the attribute is ''on'' for the section. Otherwise, the attribute is ''off'' or does not apply. Undefined attributes are set to zero.

`SHF_WRITE` The section contains data that should be writable during process execution.

`SHF_ALLOC` The section occupies memory during process execution. Some control sections do not reside in the memory image of an object file; this attribute is off for those sections.

`SHF_EXECINSTR` The section contains executable machine instructions.

`SHF_MASKPROC` All bits included in this mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Two members in the section header, `sh_link` and `sh_info`, hold special information, depending on section type.

Figure 4-12: sh_link and sh_info Interpretation

sh_type	sh_link	sh_info
SHT_DYNAMIC	The section header index of the string table used by entries in the section.	0
SHT_HASH	The section header index of the symbol table to which the hash table applies.	0
SHT_REL SHT_RELA	The section header index of the associated symbol table.	The section header index of the section to which the relocation applies.
SHT_SYMTAB SHT_DYNSYM	The section header index of the associated string table.	One greater than the symbol table index of the last local symbol (binding STB_LOCAL).
other	SHN_UNDEF	0

Special Sections

Various sections hold program and control information. Sections in the list below are used by the system and have the indicated types and attributes.

Figure 4-13: Special Sections

Name	Type	Attributes
.bss	SHT_NOBITS	SHF_ALLOC + SHF_WRITE
.comment	SHT_PROGBITS	none
.data	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.data1	SHT_PROGBITS	SHF_ALLOC + SHF_WRITE
.debug	SHT_PROGBITS	none
.dynamic	SHT_DYNAMIC	see below
.dynstr	SHT_STRTAB	SHF_ALLOC
.dynsym	SHT_DYNSYM	SHF_ALLOC
.fini	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.got	SHT_PROGBITS	see below

Figure 4-13: Special Sections (continued)

.hash	SHT_HASH	SHF_ALLOC
.init	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR
.interp	SHT_PROGBITS	see below
.line	SHT_PROGBITS	none
.note	SHT_NOTE	none
.plt	SHT_PROGBITS	see below
.rel*name*	SHT_REL	see below
.rela*name*	SHT_RELA	see below
.rodata	SHT_PROGBITS	SHF_ALLOC
.rodata1	SHT_PROGBITS	SHF_ALLOC
.shstrtab	SHT_STRTAB	none
.strtab	SHT_STRTAB	see below
.symtab	SHT_SYMTAB	see below
.text	SHT_PROGBITS	SHF_ALLOC + SHF_EXECINSTR

`.bss` This section holds uninitialized data that contribute to the program's memory image. By definition, the system initializes the data with zeros when the program begins to run. The section occupies no file space, as indicated by the section type, `SHT_NOBITS`.

`.comment` This section holds version control information.

`.data` and `.data1`
These sections hold initialized data that contribute to the program's memory image.

`.debug` This section holds information for symbolic debugging. The contents are unspecified. All section names with the prefix `.debug` are reserved for future use in the ABI. E

`.dynamic` This section holds dynamic linking information. The section's attributes will include the `SHF_ALLOC` bit. Whether the `SHF_WRITE` bit is set is processor specific. See Chapter 5 for more information.

`.dynstr` This section holds strings needed for dynamic linking, most commonly the strings that represent the names associated with symbol table entries. See Chapter 5 for more information.

`.dynsym` This section holds the dynamic linking symbol table, as ''Symbol Table'' describes. See Chapter 5 for more information.

`.fini` This section holds executable instructions that contribute to the process termination code. That is, when a program exits normally, the system arranges to execute the code in this section.

`.got` This section holds the global offset table. See ''Coding Examples'' in Chapter 3, ''Special Sections'' in Chapter 4, and ''Global Offset Table'' in Chapter 5 of the processor supplement for more information.

`.hash` This section holds a symbol hash table. See ''Hash Table'' in Chapter 5 for more information.

`.init` This section holds executable instructions that contribute to the process initialization code. That is, when a program starts to run, the system arranges to execute the code in this section before calling the main program entry point (called `main` for C programs).

`.interp` This section holds the path name of a program interpreter. If the file has a loadable segment that includes the section, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. See Chapter 5 for more information.

`.line` This section holds line number information for symbolic debugging, which describes the correspondence between the source program and the machine code. The contents are unspecified.

`.note` This section holds information in the format that ''Note Section'' in Chapter 5 describes.

`.plt` This section holds the procedure linkage table. See ''Special Sections'' in Chapter 4 and ''Procedure Linkage Table'' in Chapter 5 of the processor supplement for more information.

`.rel`*name* and `.rela`*name*

These sections hold relocation information, as ''Relocation'' below describes. If the file has a loadable segment that includes relocation, the sections' attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off. Conventionally, *name* is supplied by the section to which the relocations apply. Thus a relocation section for `.text` normally would have the name `.rel.text` or `.rela.text`.

`.rodata` and `.rodata1`

These sections hold read-only data that typically contribute to a non-writable segment in the process image. See ''Program Header'' in Chapter 5 for more information.

`.shstrtab`	This section holds section names.
`.strtab`	This section holds strings, most commonly the strings that represent the names associated with symbol table entries. If the file has a loadable segment that includes the symbol string table, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.
`.symtab`	This section holds a symbol table, as ''Symbol Table'' in this chapter describes. If the file has a loadable segment that includes the symbol table, the section's attributes will include the `SHF_ALLOC` bit; otherwise, that bit will be off.
`.text`	This section holds the ''text,'' or executable instructions, of a program.

Section names with a dot (`.`) prefix are reserved for the system, although applications may use these sections if their existing meanings are satisfactory. Applications may use names without the prefix to avoid conflicts with system sections. The object file format lets one define sections not in the list above. An object file may have more than one section with the same name.

Section names reserved for a processor architecture are formed by placing an abbreviation of the architecture name ahead of the section name. The name should be taken from the architecture names used for `e_machine`. For instance .FOO.psect is the psect section defined by the FOO architecture. Existing extensions are called by their historical names.

Pre-existing Extensions	
`.sdata`	`.tdesc`
`.sbss`	`.lit4`
`.lit8`	`.reginfo`
`.gptab`	`.liblist`
`.conflict`	

NOTE For information on processor-specific sections, see the ABI supplement for the desired processor.

String Table

String table sections hold null-terminated character sequences, commonly called strings. The object file uses these strings to represent symbol and section names. One references a string as an index into the string table section. The first byte, which is index zero, is defined to hold a null character. Likewise, a string table's last byte is defined to hold a null character, ensuring null termination for all strings. A string whose index is zero specifies either no name or a null name, depending on the context. An empty string table section is permitted; its section header's `sh_size` member would contain zero. Non-zero indexes are invalid for an empty string table.

A section header's `sh_name` member holds an index into the section header string table section, as designated by the `e_shstrndx` member of the ELF header. The following figures show a string table with 25 bytes and the strings associated with various indexes.

Index	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	`\0`	`n`	`a`	`m`	`e`	`.`	`\0`	`V`	`a`	`r`
10	`i`	`a`	`b`	`l`	`e`	`\0`	`a`	`b`	`l`	`e`
20	`\0`	`\0`	`x`	`x`	`\0`					

Figure 4-14: String Table Indexes

Index	String
0	*none*
1	`name.`
7	`Variable`
11	`able`
16	`able`
24	*null string*

As the example shows, a string table index may refer to any byte in the section. A string may appear more than once; references to substrings may exist; and a single string may be referenced multiple times. Unreferenced strings also are allowed.

Symbol Table

An object file's symbol table holds information needed to locate and relocate a program's symbolic definitions and references. A symbol table index is a subscript into this array. Index 0 both designates the first entry in the table and serves as the undefined symbol index. The contents of the initial entry are specified later in this section.

Name	Value
`STN_UNDEF`	0

A symbol table entry has the following format.

Figure 4-15: Symbol Table Entry

```
typedef struct {
        Elf32_Word        st_name;
        Elf32_Addr        st_value;
        Elf32_Word        st_size;
        unsigned char     st_info;
        unsigned char     st_other;
        Elf32_Half        st_shndx;
} Elf32_Sym;
```

`st_name` This member holds an index into the object file's symbol string table, which holds the character representations of the symbol names. If the value is non-zero, it represents a string table index that gives the symbol name. Otherwise, the symbol table entry has no name.

NOTE External C symbols have the same names in C and object files' symbol tables.

`st_value` This member gives the value of the associated symbol. Depending on the context, this may be an absolute value, an address, and so on; details appear below.

`st_size`	Many symbols have associated sizes. For example, a data object's size is the number of bytes contained in the object. This member holds 0 if the symbol has no size or an unknown size.
`st_info`	This member specifies the symbol's type and binding attributes. A list of the values and meanings appears below. The following code shows how to manipulate the values.

```
#define ELF32_ST_BIND(i)   ((i)>>4)
#define ELF32_ST_TYPE(i)   ((i)&0xf)
#define ELF32_ST_INFO(b,t) (((b)<<4)+((t)&0xf))
```

`st_other`	This member currently holds 0 and has no defined meaning.
`st_shndx`	Every symbol table entry is ''defined'' in relation to some section; this member holds the relevant section header table index. As Figure 4-7 and the related text describe, some section indexes indicate special meanings.

A symbol's binding determines the linkage visibility and behavior.

Figure 4-16: Symbol Binding, `ELF32_ST_BIND`

Name	Value
`STB_LOCAL`	0
`STB_GLOBAL`	1
`STB_WEAK`	2
`STB_LOPROC`	13
`STB_HIPROC`	15

`STB_LOCAL`	Local symbols are not visible outside the object file containing their definition. Local symbols of the same name may exist in multiple files without interfering with each other.
`STB_GLOBAL`	Global symbols are visible to all object files being combined. One file's definition of a global symbol will satisfy another file's undefined reference to the same global symbol.

STB_WEAK — Weak symbols resemble global symbols, but their definitions have lower precedence.

STB_LOPROC through STB_HIPROC
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Global and weak symbols differ in two major ways.

- When the link editor combines several relocatable object files, it does not allow multiple definitions of STB_GLOBAL symbols with the same name. On the other hand, if a defined global symbol exists, the appearance of a weak symbol with the same name will not cause an error. The link editor honors the global definition and ignores the weak ones. Similarly, if a common symbol exists (that is, a symbol whose st_shndx field holds SHN_COMMON), the appearance of a weak symbol with the same name will not cause an error. The link editor honors the common definition and ignores the weak ones.
- When the link editor searches archive libraries [see ''Archive File'' in Chapter 7], it extracts archive members that contain definitions of undefined global symbols. The member's definition may be either a global or a weak symbol. The link editor does *not* extract archive members to resolve undefined weak symbols. Unresolved weak symbols have a zero value.

In each symbol table, all symbols with STB_LOCAL binding precede the weak and global symbols. As ''Sections'' above describes, a symbol table section's sh_info section header member holds the symbol table index for the first non-local symbol.

A symbol's type provides a general classification for the associated entity.

Figure 4-17: Symbol Types, ELF32_ST_TYPE

Name	Value
STT_NOTYPE	0
STT_OBJECT	1
STT_FUNC	2
STT_SECTION	3
STT_FILE	4
STT_LOPROC	13
STT_HIPROC	15

STT_NOTYPE — The symbol's type is not specified.

STT_OBJECT — The symbol is associated with a data object, such as a variable, an array, and so on.

STT_FUNC — The symbol is associated with a function or other executable code.

STT_SECTION — The symbol is associated with a section. Symbol table entries of this type exist primarily for relocation and normally have STB_LOCAL binding.

STT_FILE — Conventionally, the symbol's name gives the name of the source file associated with the object file. A file symbol has STB_LOCAL binding, its section index is SHN_ABS, and it precedes the other STB_LOCAL symbols for the file, if it is present.

STT_LOPROC through STT_HIPROC — Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Function symbols (those with type STT_FUNC) in shared object files have special significance. When another object file references a function from a shared object, the link editor automatically creates a procedure linkage table entry for the referenced symbol. Shared object symbols with types other than STT_FUNC will not be referenced automatically through the procedure linkage table.

If a symbol's value refers to a specific location within a section, its section index member, st_shndx, holds an index into the section header table. As the section moves during relocation, the symbol's value changes as well, and references to the symbol continue to ''point'' to the same location in the program. Some special section index values give other semantics.

SHN_ABS — The symbol has an absolute value that will not change because of relocation.

SHN_COMMON — The symbol labels a common block that has not yet been allocated. The symbol's value gives alignment constraints, similar to a section's sh_addralign member. That is, the link editor will allocate the storage for the symbol at an address that is a multiple of st_value. The symbol's size tells how many bytes are required.

SHN_UNDEF — This section table index means the symbol is undefined. When the link editor combines this object file with another that defines the indicated symbol, this file's references to the symbol will be linked to the actual definition.

As mentioned above, the symbol table entry for index 0 (STN_UNDEF) is reserved; it holds the following.

Figure 4-18: Symbol Table Entry: Index 0

Name	Value	Note
st_name	0	No name
st_value	0	Zero value
st_size	0	No size
st_info	0	No type, local binding
st_other	0	
st_shndx	SHN_UNDEF	No section

Symbol Values

Symbol table entries for different object file types have slightly different interpretations for the st_value member.

- In relocatable files, st_value holds alignment constraints for a symbol whose section index is SHN_COMMON.
- In relocatable files, st_value holds a section offset for a defined symbol. That is, st_value is an offset from the beginning of the section that st_shndx identifies.
- In executable and shared object files, st_value holds a virtual address. To make these files' symbols more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation) for which the section number is irrelevant.

Although the symbol table values have similar meanings for different object files, the data allow efficient access by the appropriate programs.

Relocation

Relocation is the process of connecting symbolic references with symbolic definitions. For example, when a program calls a function, the associated call instruction must transfer control to the proper destination address at execution. In other words, relocatable files must have information that describes how to modify their section contents, thus allowing executable and shared object files to hold the right information for a process's program image. *Relocation entries* are these data.

Figure 4-19: Relocation Entries

```
typedef struct {
        Elf32_Addr        r_offset;
        Elf32_Word        r_info;
} Elf32_Rel;

typedef struct {
        Elf32_Addr        r_offset;
        Elf32_Word        r_info;
        Elf32_Sword       r_addend;
} Elf32_Rela;
```

`r_offset` This member gives the location at which to apply the relocation action. For a relocatable file, the value is the byte offset from the beginning of the section to the storage unit affected by the relocation. For an executable file or a shared object, the value is the virtual address of the storage unit affected by the relocation.

`r_info` This member gives both the symbol table index with respect to which the relocation must be made, and the type of relocation to apply. For example, a call instruction's relocation entry would hold the symbol table index of the function being called. If the index is `STN_UNDEF`, the undefined symbol index, the relocation uses 0 as the ''symbol value.'' Relocation types are processor-specific; descriptions of their behavior appear in the processor supplement. When the text in the processor supplement refers to a relocation entry's relocation type or symbol table index, it means the result of applying `ELF32_R_TYPE` or `ELF32_R_SYM`, respectively, to the entry's `r_info` member.

```
#define ELF32_R_SYM(i)    ((i)>>8)
#define ELF32_R_TYPE(i)   ((unsigned char)(i))
#define ELF32_R_INFO(s,t) (((s)<<8)+(unsigned char)(t))
```

r_addend This member specifies a constant addend used to compute the value to be stored into the relocatable field.

As shown above, only Elf32_Rela entries contain an explicit addend. Entries of type Elf32_Rel store an implicit addend in the location to be modified. Depending on the processor architecture, one form or the other might be necessary or more convenient. Consequently, an implementation for a particular machine may use one form exclusively or either form depending on context.

A relocation section references two other sections: a symbol table and a section to modify. The section header's sh_info and sh_link members, described in ''Sections'' above, specify these relationships. Relocation entries for different object files have slightly different interpretations for the r_offset member.

- In relocatable files, r_offset holds a section offset. That is, the relocation section itself describes how to modify another section in the file; relocation offsets designate a storage unit within the second section.
- In executable and shared object files, r_offset holds a virtual address. To make these files' relocation entries more useful for the dynamic linker, the section offset (file interpretation) gives way to a virtual address (memory interpretation).

Although the interpretation of r_offset changes for different object files to allow efficient access by the relevant programs, the relocation types' meanings stay the same.

Relocation Types (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

5 PROGRAM LOADING AND DYNAMIC LINKING

Introduction

This chapter describes the object file information and system actions that create running programs. Some information here applies to all systems; information specific to one processor resides in sections marked accordingly.

Executable and shared object files statically represent programs. To execute such programs, the system uses the files to create dynamic program representations, or process images. As section ''Virtual Address Space'' in Chapter 3 of the processor supplement describes, a process image has segments that hold its text, data, stack, and so on. This chapter's major sections discuss the following.

- *Program header.* This section complements Chapter 4, describing object file structures that relate directly to program execution. The primary data structure, a program header table, locates segment images within the file and contains other information necessary to create the memory image for the program.
- *Program loading.* Given an object file, the system must load it into memory for the program to run.
- *Dynamic linking.* After the system loads the program, it must complete the process image by resolving symbolic references among the object files that compose the process.

NOTE The processor supplement defines a naming convention for ELF constants that have processor ranges specified. Names such as DT_, PT_, for processor specific extensions, incorporate the name of the processor: DT_M32_SPECIAL, for example. Pre–existing processor extensions not using this convention will be supported.

Pre-existing Extensions
`DT_JMP_REL`

Program Header

An executable or shared object file's program header table is an array of structures, each describing a segment or other information the system needs to prepare the program for execution. An object file *segment* contains one or more *sections*, as ''Segment Contents'' describes below. Program headers are meaningful only for executable and shared object files. A file specifies its own program header size with the ELF header's `e_phentsize` and `e_phnum` members [see ''ELF Header'' in Chapter 4].

Figure 5-1: Program Header

```
typedef struct {
        Elf32_Word        p_type;
        Elf32_Off         p_offset;
        Elf32_Addr        p_vaddr;
        Elf32_Addr        p_paddr;
        Elf32_Word        p_filesz;
        Elf32_Word        p_memsz;
        Elf32_Word        p_flags;
        Elf32_Word        p_align;
} Elf32_Phdr;
```

`p_type` This member tells what kind of segment this array element describes or how to interpret the array element's information. Type values and their meanings appear below.

`p_offset` This member gives the offset from the beginning of the file at which the first byte of the segment resides.

`p_vaddr` This member gives the virtual address at which the first byte of the segment resides in memory.

`p_paddr` On systems for which physical addressing is relevant, this member is reserved for the segment's physical address. Because System V ignores physical addressing for application programs, this member has unspecified contents for executable files and shared objects.

p_filesz — This member gives the number of bytes in the file image of the segment; it may be zero.

p_memsz — This member gives the number of bytes in the memory image of the segment; it may be zero.

p_flags — This member gives flags relevant to the segment. Defined flag values appear below.

p_align — As ''Program Loading'' describes in this chapter of the processor supplement, loadable process segments must have congruent values for p_vaddr and p_offset, modulo the page size. This member gives the value to which the segments are aligned in memory and in the file. Values 0 and 1 mean no alignment is required. Otherwise, p_align should be a positive, integral power of 2, and p_vaddr should equal p_offset, modulo p_align.

Some entries describe process segments; others give supplementary information and do not contribute to the process image. Segment entries may appear in any order, except as explicitly noted below. Defined type values follow; other values are reserved for future use.

Figure 5-2: Segment Types, p_type

Name	Value
PT_NULL	0
PT_LOAD	1
PT_DYNAMIC	2
PT_INTERP	3
PT_NOTE	4
PT_SHLIB	5
PT_PHDR	6
PT_LOPROC	0x70000000
PT_HIPROC	0x7fffffff

PT_NULL — The array element is unused; other members' values are undefined. This type lets the program header table have ignored entries.

PT_LOAD — The array element specifies a loadable segment, described by p_filesz and p_memsz. The bytes from the file are mapped to the beginning of the memory segment. If the segment's memory size (p_memsz) is larger than the file size (p_filesz), the ''extra'' bytes are defined to hold the value 0 and to follow the segment's initialized area. The file size may not be larger than the memory size. Loadable segment entries in the program header table appear in ascending order, sorted on the `p_vaddr` member.

`PT_DYNAMIC` The array element specifies dynamic linking information. See ''Dynamic Section'' below for more information.

`PT_INTERP` The array element specifies the location and size of a null-terminated path name to invoke as an interpreter. This segment type is meaningful only for executable files (though it may occur for shared objects); it may not occur more than once in a file. If it is present, it must precede any loadable segment entry. See ''Program Interpreter'' below for further information.

`PT_NOTE` The array element specifies the location and size of auxiliary information. See ''Note Section'' below for details.

`PT_SHLIB` This segment type is reserved but has unspecified semantics. Programs that contain an array element of this type do not conform to the ABI.

`PT_PHDR` The array element, if present, specifies the location and size of the program header table itself, both in the file and in the memory image of the program. This segment type may not occur more than once in a file. Moreover, it may occur only if the program header table is part of the memory image of the program. If it is present, it must precede any loadable segment entry. See ''Program Interpreter'' below for further information.

`PT_LOPROC` through `PT_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

NOTE Unless specifically required elsewhere, all program header segment types are optional. That is, a file's program header table may contain only those elements relevant to its contents.

Base Address

As ''Program Loading'' in this chapter of the processor supplement describes, the virtual addresses in the program headers might not represent the actual virtual addresses of the program's memory image. Executable files typically contain absolute code. To let the process execute correctly, the segments must reside at the virtual addresses used to build the executable file. On the other hand, shared object segments typically contain position-independent code. This lets a segment's virtual address change from one process to another, without invalidating execution behavior. Though the system chooses virtual addresses for individual processes, it maintains the segments' *relative positions*. Because position-independent code uses relative addressing between segments, the difference between virtual addresses in memory must match the difference between virtual addresses in the file. The difference between the virtual address of any segment in memory and the corresponding virtual address in the file is thus a single constant value for any one executable or shared object in a given process. This difference is the *base address*. One use of the base address is to relocate the memory image of the program during dynamic linking.

An executable or shared object file's base address is calculated during execution from three values: the virtual memory load address, the maximum page size, and the lowest virtual address of a program's loadable segment. To compute the base address, one determines the memory address associated with the lowest `p_vaddr` value for a `PT_LOAD` segment. This address is truncated to the nearest multiple of the maximum page size. The corresponding `p_vaddr` value itself is also truncated to the nearest multiple of the maximum page size. The base address is the difference between the truncated memory address and the truncated `p_vaddr` value.

See this chapter in the processor supplement for more information and examples. ''Operating System Interface'' of Chapter 3 in the processor supplement contains more information about the virtual address space and page size.

Segment Permissions

A program to be loaded by the system must must have at least one loadable segment (although this is not required by the file format). When the system creates loadable segments' memory images, it gives access permissions as specified in the `p_flags` member.

Figure 5-3: Segment Flag Bits, `p_flags`

Name	Value	Meaning
PF_X	0x1	Execute
PF_W	0x2	Write
PF_R	0x4	Read
PF_MASKPROC	0xf0000000	Unspecified

All bits included in the PF_MASKPROC mask are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

If a permission bit is 0, that type of access is denied. Actual memory permissions depend on the memory management unit, which may vary from one system to another. Although all flag combinations are valid, the system may grant more access than requested. In no case, however, will a segment have write permission unless it is specified explicitly. The following table shows both the exact flag interpretation and the allowable flag interpretation. ABI-conforming systems may provide either.

Figure 5-4: Segment Permissions

Flags	Value	Exact	Allowable
none	0	All access denied	All access denied
PF_X	1	Execute only	Read, execute
PF_W	2	Write only	Read, write, execute
PF_W+PF_X	3	Write, execute	Read, write, execute
PF_R	4	Read only	Read, execute
PF_R+PF_X	5	Read, execute	Read, execute
PF_R+PF_W	6	Read, write	Read, write, execute
PF_R+PF_W+PF_X	7	Read, write, execute	Read, write, execute

For example, typical text segments have read and execute—but not write—permissions. Data segments normally have read, write, and execute permissions.

Segment Contents

An object file segment comprises one or more sections, though this fact is transparent to the program header. Whether the file segment holds one or many sections also is immaterial to program loading. Nonetheless, various data must be present for program execution, dynamic linking, and so on. The diagrams below illustrate segment contents in general terms. The order and membership of sections within a segment may vary; moreover, processor-specific constraints may alter the examples below. See the processor supplement for details.

Text segments contain read-only instructions and data, typically including the following sections described in Chapter 4. Other sections may also reside in loadable segments; these examples are not meant to give complete and exclusive segment contents.

Figure 5-5: Text Segment

`.text`
`.rodata`
`.hash`
`.dynsym`
`.dynstr`
`.plt`
`.rel.got`

Data segments contain writable data and instructions, typically including the following sections.

Figure 5-6: Data Segment

`.data`
`.dynamic`
`.got`
`.bss`

A `PT_DYNAMIC` program header element points at the `.dynamic` section, explained in ''Dynamic Section'' below. The `.got` and `.plt` sections also hold information related to position-independent code and dynamic linking. Although the `.plt` appears in a text segment above, it may reside in a text or a data segment, depending on the processor. See ‘‘Global Offset Table’’ and ‘‘Procedure Linkage Table’’ in this chapter of the processor supplement for details.

As ‘‘Sections’’ in Chapter 4 describes, the `.bss` section has the type `SHT_NOBITS`. Although it occupies no space in the file, it contributes to the segment’s memory image. Normally, these uninitialized data reside at the end of the segment, thereby making `p_memsz` larger than `p_filesz` in the associated program header element.

Note Section

Sometimes a vendor or system builder needs to mark an object file with special information that other programs will check for conformance, compatibility, etc. Sections of type `SHT_NOTE` and program header elements of type `PT_NOTE` can be used for this purpose. The note information in sections and program header elements holds any number of entries, each of which is an array of 4-byte words in the format of the target processor. Labels appear below to help explain note information organization, but they are not part of the specification.

Figure 5-7: Note Information

`namesz`
`descsz`
`type`
`name` . . .
`desc` . . .

`namesz` and `name`

The first `namesz` bytes in `name` contain a null-terminated character representation of the entry’s owner or originator. There is no formal mechanism for avoiding name conflicts. By convention, vendors use their own name, such as ‘‘XYZ Computer Company,’’ as the identifier. If no name is present, `namesz` contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the descriptor. Such padding is not included in `namesz`.

descsz and desc
: The first descsz bytes in desc hold the note descriptor. The ABI places no constraints on a descriptor's contents. If no descriptor is present, descsz contains 0. Padding is present, if necessary, to ensure 4-byte alignment for the next note entry. Such padding is not included in descsz.

type
: This word gives the interpretation of the descriptor. Each originator controls its own types; multiple interpretations of a single type value may exist. Thus, a program must recognize both the name and the type to ''understand'' a descriptor. Types currently must be non-negative. The ABI does not define what descriptors mean.

To illustrate, the following note segment holds two entries.

Figure 5-8: Example Note Segment

	+0	+1	+2	+3	
namesz	7				
descsz	0				No descriptor
type	1				
name	X	Y	Z		
	C	o	\0	*pad*	
namesz	7				
descsz	8				
type	3				
name	X	Y	Z		
	C	o	\0	*pad*	
desc	*word 0*				
	word 1				

NOTE The system reserves note information with no name (namesz==0) and with a zero-length name (name[0]=='\0') but currently defines no types. All other names must have at least one non-null character.

NOTE

Note information is optional. The presence of note information does not affect a program's ABI conformance, provided the information does not affect the program's execution behavior. Otherwise, the program does not conform to the ABI and has undefined behavior.

Program Loading (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Dynamic Linking

Program Interpreter

An executable file that participates in dynamic linking shall have one `PT_INTERP` program header element. During the function `exec`, the system retrieves a path name from the `PT_INTERP` segment and creates the initial process image from the interpreter file's segments. That is, instead of using the original executable file's segment images, the system composes a memory image for the interpreter. It then is the interpreter's responsibility to receive control from the system and provide an environment for the application program.

As ''Process Initialization'' in Chapter 3 of the processor supplement mentions, the interpreter receives control in one of two ways. First, it may receive a file descriptor to read the executable file, positioned at the beginning. It can use this file descriptor to read and/or map the executable file's segments into memory. Second, depending on the executable file format, the system may load the executable file into memory instead of giving the interpreter an open file descriptor. With the possible exception of the file descriptor, the interpreter's initial process state matches what the executable file would have received. The interpreter itself may not require a second interpreter. An interpreter may be either a shared object or an executable file.

- A shared object (the normal case) is loaded as position-independent, with addresses that may vary from one process to another; the system creates its segments in the dynamic segment area used by the function `mmap` and related services [see ''Virtual Address Space'' in Chapter 3 of the processor supplement]. Consequently, a shared object interpreter typically will not conflict with the original executable file's original segment addresses.
- An executable file is loaded at fixed addresses; the system creates its segments using the virtual addresses from the program header table. Consequently, an executable file interpreter's virtual addresses may collide with the first executable file; the interpreter is responsible for resolving conflicts.

Dynamic Linker

When building an executable file that uses dynamic linking, the link editor adds a program header element of type `PT_INTERP` to an executable file, telling the system to invoke the dynamic linker as the program interpreter.

> **NOTE** The locations of the system provided dynamic linkers are processor–specific.

`Exec` and the dynamic linker cooperate to create the process image for the program, which entails the following actions:

- Adding the executable file's memory segments to the process image;
- Adding shared object memory segments to the process image;
- Performing relocations for the executable file and its shared objects;
- Closing the file descriptor that was used to read the executable file, if one was given to the dynamic linker;
- Transferring control to the program, making it look as if the program had received control directly from the function `exec` X

The link editor also constructs various data that assist the dynamic linker for executable and shared object files. As shown above in ''Program Header,'' these data reside in loadable segments, making them available during execution. (Once again, recall the exact segment contents are processor-specific. See the processor supplement for complete information.)

- A `.dynamic` section with type `SHT_DYNAMIC` holds various data. The structure residing at the beginning of the section holds the addresses of other dynamic linking information.
- The `.hash` section with type `SHT_HASH` holds a symbol hash table.
- The `.got` and `.plt` sections with type `SHT_PROGBITS` hold two separate tables: the global offset table and the procedure linkage table. Chapter 3 discusses how programs use the global offset table for position-independent code. Sections below explain how the dynamic linker uses and changes the tables to create memory images for object files.

Because every ABI-conforming program imports the basic system services from a shared object library [see ''System Library'' in Chapter 6], the dynamic linker participates in every ABI-conforming program execution.

As ''Program Loading'' explains in the processor supplement, shared objects may occupy virtual memory addresses that are different from the addresses recorded in the file's program header table. The dynamic linker relocates the memory image, updating absolute addresses before the application gains control. Although the absolute address values would be correct if the library were loaded at the addresses specified in the program header table, this normally is not the case.

If the process environment [see the function `exec`] contains a variable named `LD_BIND_NOW` with a non-null value, the dynamic linker processes all relocation before transferring control to the program. For example, all the following environment entries would specify this behavior. X

- `LD_BIND_NOW=1`
- `LD_BIND_NOW=on`
- `LD_BIND_NOW=off`

Otherwise, `LD_BIND_NOW` either does not occur in the environment or has a null value. The dynamic linker is permitted to evaluate procedure linkage table entries lazily, thus avoiding symbol resolution and relocation overhead for functions that are not called. See ''Procedure Linkage Table'' in this chapter of the processor supplement for more information.

Dynamic Section

If an object file participates in dynamic linking, its program header table will have an element of type `PT_DYNAMIC`. This ''segment'' contains the `.dynamic` section. A special symbol, `_DYNAMIC`, labels the section, which contains an array of the following structures.

Figure 5-9: Dynamic Structure

```
typedef struct {
        Elf32_Sword       d_tag;
        union {
                Elf32_Word       d_val;
                Elf32_Addr       d_ptr;
        } d_un;
} Elf32_Dyn;

extern Elf32_Dyn   _DYNAMIC[];
```

For each object with this type, `d_tag` controls the interpretation of `d_un`.

`d_val` These `Elf32_Word` objects represent integer values with various interpretations.

`d_ptr` These `Elf32_Addr` objects represent program virtual addresses. As mentioned previously, a file's virtual addresses might not match the memory virtual addresses during execution. When interpreting addresses contained in the dynamic structure, the dynamic linker computes actual addresses, based on the original file value and the memory base address. For consistency, files do *not* contain relocation entries to ''correct'' addresses in the dynamic structure.

The following table summarizes the tag requirements for executable and shared object files. If a tag is marked ''mandatory,'' then the dynamic linking array for an ABI-conforming file must have an entry of that type. Likewise, ''optional'' means an entry for the tag may appear but is not required.

Figure 5-10: Dynamic Array Tags, `d_tag`

Name	Value	d_un	Executable	Shared Object
DT_NULL	0	ignored	mandatory	mandatory
DT_NEEDED	1	d_val	optional	optional
DT_PLTRELSZ	2	d_val	optional	optional
DT_PLTGOT	3	d_ptr	optional	optional
DT_HASH	4	d_ptr	mandatory	mandatory
DT_STRTAB	5	d_ptr	mandatory	mandatory
DT_SYMTAB	6	d_ptr	mandatory	mandatory

Figure 5-10: Dynamic Array Tags, `d_tag` (continued)

Name	Value	d_un	Executable	Shared Object
DT_RELA‡	7	d_ptr	mandatory	optional
DT_RELASZ	8	d_val	mandatory	optional
DT_RELAENT	9	d_val	mandatory	optional
DT_STRSZ	10	d_val	mandatory	mandatory
DT_SYMENT	11	d_val	mandatory	mandatory
DT_INIT	12	d_ptr	optional	optional
DT_FINI	13	d_ptr	optional	optional
DT_SONAME	14	d_val	ignored	optional
DT_RPATH	15	d_val	optional	ignored
DT_SYMBOLIC	16	ignored	ignored	optional
DT_REL†	17	d_ptr	mandatory	optional
DT_RELSZ	18	d_val	mandatory	optional
DT_RELENT	19	d_val	mandatory	optional
DT_PLTREL	20	d_val	optional	optional
DT_DEBUG	21	d_ptr	optional	ignored
DT_TEXTREL	22	ignored	optional	optional
DT_JMPREL	23	d_ptr	optional	optional
DT_BIND_NOW	24	ignored	optional	optional M
DT_LOPROC	0x70000000	unspecified	unspecified	unspecified
DT_HIPROC	0x7fffffff	unspecified	unspecified	unspecified

† See the description of DT_RELA and DT_REL below for the relationship between these two tags. M

DT_NULL — An entry with a DT_NULL tag marks the end of the _DYNAMIC array.

DT_NEEDED — This element holds the string table offset of a null-terminated string, giving the name of a needed library. The offset is an index into the table recorded in the DT_STRTAB entry. See ''Shared Object Dependencies'' for more information about these names. The dynamic array may contain multiple entries with this type. These entries' relative order is significant, though their relation to entries of other types is not.

DT_PLTRELSZ — This element holds the total size, in bytes, of the relocation entries associated with the procedure linkage table. If an entry of type DT_JMPREL is present, a DT_PLTRELSZ must accompany it.

DT_PLTGOT — This element holds an address associated with the procedure linkage table and/or the global offset table. See this section in the processor supplement for details.

DT_HASH — This element holds the address of the symbol hash table, described in ''Hash Table.'' This hash table refers to the symbol table referenced by the DT_SYMTAB element.

DT_STRTAB — This element holds the address of the string table, described in Chapter 4. Symbol names, library names, and other strings reside in this table.

DT_SYMTAB — This element holds the address of the symbol table, described in Chapter 4, with Elf32_Sym entries for the 32-bit class of files.

DT_RELA — This element holds the address of a relocation table, described in Chapter 4. Entries in the table have explicit addends, such as Elf32_Rela for the 32-bit file class. An object file may have multiple relocation sections. When building the relocation table for an executable or shared object file, the link editor catenates those sections to form a single table. Although the sections remain independent in the object file, the dynamic linker sees a single table. When the dynamic linker creates the process image for an executable file or adds a shared object to the process image, it reads the relocation table and performs the associated actions. If this element is present, the dynamic structure must also have DT_RELASZ and DT_RELAENT elements. When relocation is ''mandatory'' for a file, either DT_RELA or DT_REL must occur (both are permitted but only one is required).

DT_RELASZ — This element holds the total size, in bytes, of the DT_RELA relocation table.

DT_RELAENT — This element holds the size, in bytes, of the DT_RELA relocation entry.

DT_STRSZ — This element holds the size, in bytes, of the string table.

DT_SYMENT — This element holds the size, in bytes, of a symbol table entry.

DT_INIT — This element holds the address of the initialization function, discussed in ''Initialization and Termination Functions'' below.

DT_FINI — This element holds the address of the termination function, discussed in ''Initialization and Termination Functions'' below.

`DT_SONAME` This element holds the string table offset of a null-terminated string, giving the name of the shared object. The offset is an index into the table recorded in the `DT_STRTAB` entry. See ''Shared Object Dependencies'' below for more information about these names.

`DT_RPATH` This element holds the string table offset of a null-terminated search library search path string, discussed in ''Shared Object Dependencies.'' The offset is an index into the table recorded in the `DT_STRTAB` entry.

`DT_SYMBOLIC` This element's presence in a shared object library alters the dynamic linker's symbol resolution algorithm for references within the library. Instead of starting a symbol search with the executable file, the dynamic linker starts from the shared object itself. If the shared object fails to supply the referenced symbol, the dynamic linker then searches the executable file and other shared objects as usual.

`DT_REL` This element is similar to `DT_RELA`, except its table has implicit addends, such as `Elf32_Rel` for the 32-bit file class. If this element is present, the dynamic structure must also have `DT_RELSZ` and `DT_RELENT` elements.

`DT_RELSZ` This element holds the total size, in bytes, of the `DT_REL` relocation table.

`DT_RELENT` This element holds the size, in bytes, of the `DT_REL` relocation entry.

`DT_PLTREL` This member specifies the type of relocation entry to which the procedure linkage table refers. The `d_val` member holds `DT_REL` or `DT_RELA`, as appropriate. All relocations in a procedure linkage table must use the same relocation.

`DT_DEBUG` This member is used for debugging. Its contents are not specified for the ABI; programs that access this entry are not ABI-conforming.

`DT_TEXTREL` This member's absence signifies that no relocation entry should cause a modification to a non-writable segment, as specified by the segment permissions in the program header table. If this member is present, one or more relocation entries might request modifications to a non-writable segment, and the dynamic linker can prepare accordingly.

`DT_JMPREL` If present, this entries' `d_ptr` member holds the address of relocation entries associated solely with the procedure linkage table. Separating these relocation entries lets the dynamic linker ignore them during process initialization, if lazy binding is enabled. If this entry is present, the related entries of types `DT_PLTRELSZ` and `DT_PLTREL` must also be present.

`DT_BIND_NOW` If present in a shared object or executable, this entry instructs the dynamic linker to process all relocations for the object containing this entry before transferring control to the program. The presence of this entry takes precedence over a directive to use lazy binding for this object when specified through the environment or via `dlopen`(BA_LIB).

`DT_LOPROC` through `DT_HIPROC`
Values in this inclusive range are reserved for processor-specific semantics. If meanings are specified, the processor supplement explains them.

Except for the `DT_NULL` element at the end of the array, and the relative order of `DT_NEEDED` elements, entries may appear in any order. Tag values not appearing in the table are reserved.

Shared Object Dependencies

When the link editor processes an archive library, it extracts library members and copies them into the output object file. These statically linked services are available during execution without involving the dynamic linker. Shared objects also provide services, and the dynamic linker must attach the proper shared object files to the process image for execution.

When the dynamic linker creates the memory segments for an object file, the dependencies (recorded in `DT_NEEDED` entries of the dynamic structure) tell what shared objects are needed to supply the program's services. By repeatedly connecting referenced shared objects and their dependencies, the dynamic linker builds a complete process image. When resolving symbolic references, the dynamic linker examines the symbol tables with a breadth-first search. That is, it first looks at the symbol table of the executable program itself, then at the symbol tables of the `DT_NEEDED` entries (in order), then at the second level `DT_NEEDED` entries, and so on. Shared object files must be readable by the process; other permissions are not required.

NOTE Even when a shared object is referenced multiple times in the dependency list, the dynamic linker will connect the object only once to the process.

Names in the dependency list are copies either of the `DT_SONAME` strings or the path names of the shared objects used to build the object file. For example, if the link editor builds an executable file using one shared object with a `DT_SONAME` entry of `lib1` and another shared object library with the path name `/usr/lib/lib2`, the executable file will contain `lib1` and `/usr/lib/lib2` in its dependency list.

If a shared object name has one or more slash (`/`) characters anywhere in the name, such as `/usr/lib/lib2` above or `directory/file`, the dynamic linker uses that string directly as the path name. If the name has no slashes, such as `lib1` above, three facilities specify shared object path searching, with the following precedence.

- First, the dynamic array tag `DT_RPATH` may give a string that holds a list of directories, separated by colons (`:`). For example, the string `/home/dir/lib:/home/dir2/lib:` tells the dynamic linker to search first the directory `/home/dir/lib`, then `/home/dir2/lib`, and then the current directory to find dependencies.
- Second, a variable called `LD_LIBRARY_PATH` in the process environment [see the function `exec`] may hold a list of directories as above, optionally followed by a semicolon (`;`) and another directory list. The following values would be equivalent to the previous example: X
  - `LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib:`
  - `LD_LIBRARY_PATH=/home/dir/lib;/home/dir2/lib:`
  - `LD_LIBRARY_PATH=/home/dir/lib:/home/dir2/lib:;`

  All `LD_LIBRARY_PATH` directories are searched after those from `DT_RPATH`. Although some programs (such as the link editor) treat the lists before and after the semicolon differently, the dynamic linker does not. Nevertheless, the dynamic linker accepts the semicolon notation, with the semantics described above.
- Finally, if the other two groups of directories fail to locate the desired library, the dynamic linker searches `/usr/lib`.

NOTE For security, the dynamic linker ignores environmental search specifications (such as `LD_LIBRARY_PATH`) for set-user and set-group ID programs. It does, M however, search `DT_RPATH` directories and `/usr/lib`. The same restriction M may be applied to processes that have more than minimal privileges on systems with installed extended security systems.

Global Offset Table (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Procedure Linkage Table (Processor-Specific)

NOTE This section requires processor-specific information. The ABI supplement for the desired processor describes the details.

Hash Table

A hash table of `Elf32_Word` objects supports symbol table access. Labels appear below to help explain the hash table organization, but they are not part of the specification.

Figure 5-11: Symbol Hash Table

`nbucket`
`nchain`

`bucket[0]`
`. . .`
`bucket[nbucket - 1]`

`chain[0]`
`. . .`
`chain[nchain - 1]`

The `bucket` array contains `nbucket` entries, and the `chain` array contains `nchain` entries; indexes start at 0. Both `bucket` and `chain` hold symbol table indexes. Chain table entries parallel the symbol table. The number of symbol table entries should equal `nchain`; so symbol table indexes also select chain table entries. A hashing function (shown below) accepts a symbol name and returns a value that may be used to compute a `bucket` index. Consequently, if the hashing function returns the value x for some name, `bucket[x%nbucket]` gives an index, y, into both the symbol table and the chain table. If the symbol table entry is not the one desired, `chain[y]` gives the next symbol table entry with the same hash value. One can follow the `chain` links until either the selected symbol table entry holds the desired name or the `chain` entry contains the value `STN_UNDEF`.

Figure 5-12: Hashing Function

```
unsigned long
elf_hash(const unsigned char *name)
{
        unsigned long      h = 0, g;

        while (*name)
        {
                h = (h << 4) + *name++;
                if (g = h & 0xf0000000)
                        h ^= g >> 24;
                h &= ~g;
        }
        return h;
}
```

Initialization and Termination Functions

After the dynamic linker has built the process image and performed the relocations, each shared object gets the opportunity to execute some initialization code. All shared object initializations happen before the executable file gains control.

Before the initialization code for any object A is called, the initialization code for any other objects that object A depends on are called. For these purposes, an object A depends on another object B, if B appears in A's list of needed objects (recorded in the DT_NEEDED entries of the dynamic structure). The order of initialization for circular dependencies is undefined.

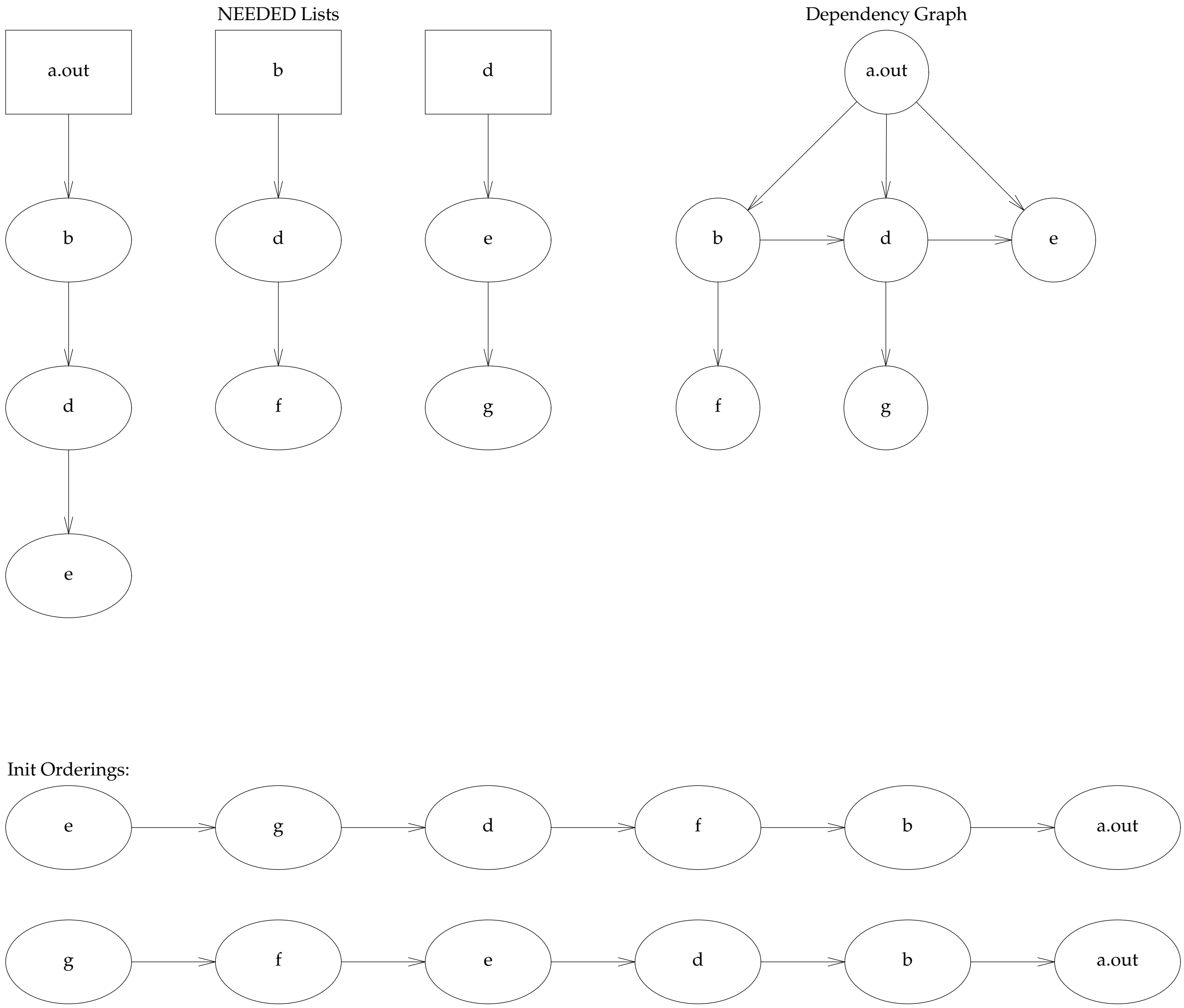
The initialization of objects occurs by recursing through the needed entries of each object. The initialization code for an object is invoked after the needed entries for that object have been processed. The order of processing among the entries of a particular list of needed objects is unspecified.

> **NOTE** Each processor supplement may optionally further restrict the algorithm used to determine the order of initialization. Any such restriction, however, may not conflict with the rules described by this specification.

The following example illustrates two of the possible correct orderings which can be generated for the example NEEDED lists. In this example the a.out is dependent on b, d, and e. b is dependent on d and f, while d is dependent on e and g. From this information a dependency graph can be drawn. The above algorithm on initialization will then allow the following specified initialization orderings among others.

Initialization Ordering Example

Figure 5-13: Initialization Ordering Example



Similarly, shared objects may have termination functions, which are executed with the function `atexit` mechanism after the base process begins its termination sequence. The order in which the dynamic linker calls termination functions is the exact reverse order of their corresponding initialization functions. If a shared object has a termination function, but no initialization function, the termination function will execute in the order it would have as if the shared object's initialization function was present. The dynamic linker ensures that it will not execute any initialization or termination functions more than once.

Shared objects designate their initialization and termination functions through the `DT_INIT` and `DT_FINI` entries in the dynamic structure, described in ''Dynamic Section'' above. Typically, the code for these functions resides in the `.init` and `.fini` sections, mentioned in ''Sections'' of Chapter 4.

NOTE Although the function `atexit` termination processing normally will be done, it is not guaranteed to have executed upon process death. In particular, the process will not execute the termination processing if it calls `_exit` [see the function `exit`] or if the process dies because it received a signal that it neither caught nor ignored.

The dynamic linker is not responsible for calling the executable file's `.init` section or registering the executable file's `.fini` section with the function `atexit`. Termination functions specified by users via the `atexit` mechanism must be executed before any termination functions of shared objects.

6 LIBRARIES

Introduction

Every ABI-conforming system supports some general-purpose libraries. Facilities in these libraries manipulate system data files, trap to the operating system, and so on. Together, these libraries hold routines appearing in sections BA_OS, BA_LIB, KE_OS, RS_LIB, and MT_LIB of the *System V Interface Definition, Fourth Edition*.

`libc` The C library, containing various facilities defined by System V, ANSI C, ISO MSE, POSIX, and so on. It contains interfaces to basic system services. M

`libthread` M
The thread library, containing interfaces to support multiprocessing M
applications and asynchronous I/O operations included in the MT M
extension of the *System V Interface Definition, Fourth Edition*. M

`libdl` The dynamic linking library, containing routines that give the user M
direct access to the dynamic linking facilities.

`libnsl` The networking services library, contains the transport layer interface X
routines, as well as routines for machine-independent data representa- X
tion, remote procedure calls, and other networking support. These X
routines are described in the Networking Services volume of the X
X/Open CAE Specification, Issue 4.2, and the BA_OS and RS_LIB sections X
of the *System V Interface Definition, Fourth Edition* (see the Conformance X
Rule in chapter1).

`libsocket`
A library containing the sockets routines as described in the Network- X
ing Services volume of the *X/Open CAE Specification, Issue 4.2*. X

`libcurses` X
This library provides routines for updating character screens as X
described in the X/Open Curses, Issue 4 of the *X/Open CAE* X
Specification, Issue 4.2.

The following libraries may be supported as extensions to the ABI. G

`libX` A library for building applications using the X11 Window System pro- G
tocol described in the Graphics chapter. G

`libXt` A library for building applications using the X Toolkit Intrinsics. G

`libXext`	A library for building applications using the X11 Nonrectangular Window Shape Extension.
`libXm`	A library for building applications using the Motif 1.2 Interface.
`libMrm`	A library containing the resource manager for the Motif 1.2 Interface.

NOTE `libsys` has been removed from the Application Binary Interface. Originally `libsys` was devised as a subset of `libc` functionality comprised of a useful superset of system calls. Difficulties in the division between the two libraries, additional maintenance efforts and lack of use of `libsys` by applications pointing to the ABI have led to the elimination of the library from the *ABI*.

As a *binary* specification, the *ABI* gives shared library organization; that is, it tells what services reside in what shared libraries. Programs use the dynamic linking mechanism described in Chapter 5 to access their services.

The *ABI* does not duplicate the descriptions available in the *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition* and other references that tell what the facilities do, how to use them, and so on. However, the interfaces to some services may have different names and syntax at the system level than they do at the source level. When these differences exist, this document (the *System V ABI*) specifies the name of the source-level services that must be supported on conforming systems, and the names and descriptions of these interfaces are given in each processor supplement to the *ABI*.

Shared libraries contribute to the application execution environment and thus appear in the *ABI*. Functions that reside directly in application files are not specified. For example, mathematical routines, such as `sin`, do not appear below. They would be available in a System V development environment, but an application's executable file would contain the associated code. Assuming the implementations of the functions themselves are ABI-conforming, their presence does not affect the conformance of the application. Moreover, the absence of shared library versions of particular services does not imply a deprecation of those services.

The *ABI* requires conforming applications to use the dynamic linking mechanism described in Chapter 5 to access the services provided in the C Library, `libc`, the Dynamic Linking Library, `libdl`, the threads Library, `libthread`, and in the Networking Services Library, `libnsl`. In addition, on systems that support the graphical windowing terminal interface, the dynamic linking mechanism must be used to access the X Window System Library services, `libX`. Use of the other shared libraries documented here is optional for applications; the services they contain may be obtained through the use of equivalent archive library routines. An application that accesses ABI services from both the shared library and the static archive version of the same library is not ABI conforming.

Shared Library Names

As Chapter 5 describes, executable and shared object files contain the names of required shared libraries. In the names below, `number` represents the current version number of the library. The version number of each library is specified in the *ABI* supplement for the desired processor. M M

Figure 6-1: Shared Library Names

Library	Reference Name	
`libc`	`/usr/lib/libc.so.number`	
`libthread`	`/usr/lib/libthread.so.number`	M
`libdl`	`/usr/lib/libdl.so.number`	M
`libnsl`	`/usr/lib/libnsl.so.number`	
`libsocket` `/usr/lib/libsocket.so.2`		X
`libcurses` `/usr/lib/libcurses.so.1`		X
`libX`	`/usr/lib/libX11.so.number`	
`libXt`	`/usr/lib/libXt.so.number`	G
`libXext`	`/usr/lib/libXext.so.number`	M
`libXm`	`/usr/lib/libXm.so.number`	M
`libMrm`	`/usr/lib/libMrm.so.number`	M

Dependencies Among Libraries

System libraries may depend on other libraries both defined by the ABI and not defined by the ABI. The runtime environment should not require that such dependencies be reflected in the executable if the executable itself does not have any dependencies on those libraries. M M M

System Service Synonyms

In addition to the routine names listed in the tables below, the C library includes synonyms for some of its services. These other symbols are available to conform to language and system standards. As an example, System V defines `read` as the name of an operating system facility. On the other hand, ANSI C does not define `read`, and it prohibits a *strictly conforming* implementation from usurping application names without a leading underscore (_). Thus if a synonym for `read` were not available, the system could not support a strictly conforming implementation of the ANSI C language.

name This gives the traditional name, such as `read`.

_name This gives a system service name that follows the ANSI C convention of reserving symbols beginning with an underscore, such as `_read`.

The title of each of the below tables specifies whether synonyms must exist for the entries in that table.

Although many system services have two names, two exceptions exist to this synonym convention. System V defines both `exit` and `_exit` as different facilities. Consequently, the symbols `exit` and `_exit` have no synonyms and refer to different services.

C Library

The C library, `libc`, contains all of the symbols listed in the following two tables. `libc` contains the basic system services. Although special instructions are necessary to change from user to kernel mode, the *ABI* explicitly does not specify the correspondence of these instructions to system calls. The tables below contain routines that correspond to basic system services, as well as to service routines that provide a functional interface to system data files.

Though all the function symbols listed in the tables below must be present in `libc`, not all of the functions they reference may actually be implemented. See the ''Implementation of `libc` Routines'' section that follows for more detail.

The first table lists routines associated with the ANSI C standard and the ISO MSE routines as they are currently defined in the SVID. M

NOTE Although synonyms are not required for the following interfaces, they are permitted to exist. However applications that make use of the synonyms are not considered ABI compliant. M

Figure 6-2: `libc` **Contents, Names without Synonyms**

abort	ferror	fscanf	isdigit	iswupper*
abs	fflush	fseek	isgraph	iswxdigit*
asctime	fgetc	fsetpos	islower	isxdigit
atexit	fgetpos	ftell	isprint	labs
atof	fgets	fwprintf*	ispunct	ldexp
atoi	fgetwc*	fwrite	isspace	ldiv
atol	fgetws*	fwscanf*	isupper	localeconv
bsearch	fopen	getc	iswalnum*	localtime
calloc	fprintf	getchar	iswalpha*	longjmp
clearerr	fputc	getenv	iswcntrl*	malloc
clock	fputs	gets	iswctype*	mblen
ctime	fputwc*	getwchar*	iswdigit*	mbrlen*
difftime	fputws*	getwc*	iswgraph*	mbrtowc*
div	fread	gmtime	iswlower*	mbsinit*
exit	free	isalnum	iswprint*	mbsrtowcs*
fclose	freopen	isalpha	iswpunct*	mbstowcs
feof	frexp	iscntrl	iswspace*	mbtowc

Figure 6-2: `libc` **Contents, Names without Synonyms** (continued)

memchr	rewind	strncmp	towupper*	wcsncmp*
memcmp	scanf	strncpy	ungetc	wcsncpy*
memcpy	setbuf	strpbrk	ungetwc*	wcspbrk*
memmove	setjmp	strrchr	vfprintf	wcsrchr*
memset	setlocale	strspn	vfwprintf*	wcsrtombs*
mktime	setvbuf	strstr	vprintf	wcsspn*
modf	signal	strtod	vsprintf	wcsstr*
perror	sprintf	strtok	vswprintf*	wcstod*
printf	srand	strtol	vwprintf*	wcstok*
putc	sscanf	strtoul	wcrtomb*	wcstol*
putchar	strcat	strxfrm	wcscat*	wcstombs
puts	strchr	swprintf*	wcschr*	wcstoul*
putwchar*	strcmp	swscanf*	wcscmp*	wcsxfrm*
putwc*	strcoll	system	wcscoll*	wctob*
qsort	strcpy	tmpfile	wcscpy*	wctomb
raise	strcspn	tmpnam	wcscspn*	wctype*
rand	strerror	tolower	wcsftime*	wprintf*
realloc	strftime	toupper	wcslen*	wscanf*
remove	strlen	towlower*	wcsncat*	_exit
rename	strncat			

*Function is new to the Fourth Edition of the ABI. M

Each of the routines listed in the table below is present in `libc` in the listed form, as well as in synonym form, as described in a following section.

Figure 6-3: `libc` **Contents, Names with Synonyms**

access	flockfile*	getsubopt	modload*
acct#	fnmatch*	gettxt	modpath*
alarm	fork	getuid	modstat*
asctime_r*	fork1*	getw	moduload*
catclose	forkall*	glob*	monitor
catgets	fpathconf	globfree*	mount
catopen	fstatvfs	gmtime_r*	mprotect
cfgetispeed	fsync	grantpt	msgctl
cfgetospeed	ftok	hcreate	msgget
cfsetispeed	ftrylockfile*	hdestroy	msgrcv
cfsetospeed	funlockfile*	hsearch	msgsnd

Figure 6-3: `libc` **Contents, Names with Synonyms** (continued)

chdir	getc_unlocked*	iconv*	msync
chmod	getchar_unlocked*	iconv_close*	munlock
chown	getcontext	iconv_open*	munmap
chroot	getcwd	initgroups	nextafter
close	getdate	ioctl	nftw
closedir	getegid	isascii	nice
confstr*	geteuid	isastream	nl_langinfo
creat	getgid	isatty	open
ctermid	getgrgid	isnan	opendir
ctime_r*	getgrnam	isnand#	pathconf
cuserid	getgroups	kill	pause
dup	getksym*	lchown	pclose
dup2	getlogin	lfind	pipe
execl	getlogin_r*	link	poll
execle	getmsg	localtime_r*	popen
execlp	getopt	lockf	pread*
execv	getpass	logb	priocntl*
execve	getpass_r*	lsearch	profil#
execvp	getpgid	lseek	ptrace
fattach	getpgrp	makecontext	ptsname
fchdir	getpid	memccpy	putc_unlocked*
fchmod	getpmsg	memcntl	putchar_unlocked*
fchown	getppid	mkdir	putenv
fcntl	getpwnam	mkfifo	putmsg
fdetach	getpwuid	mktemp	putpmsg
fdopen	getrlimit	mlock	putw
fileno	getsid	mmap	pwrite*
rand_r*	shmctl	strfmon*	ttyname
read	shmdt	strptime*	ttyname_r*
readdir	shmget	strtof*	twalk
readdir_r*	sigaction	strtok_r*	tzset
readlink	sigaddset	strtold*	ulimit
readv	sigaltstack	swab	umask
regcomp*	sigdelset	swapcontext	umount
regerror*	sigemptyset	symlink	unlink
regexec*	sigfillset	sync	unlockpt
regfree*	sighold	sysconf	utime
rewinddir	sigignore	tcdrain	vfscanf*

Figure 6-3: `libc` **Contents, Names with Synonyms** (continued)

rmdir	sigismember	tcflow	vfwscanf*
scalb	siglongjmp	tcflush	vscanf*
seekdir	sigpause	tcgetattr	vsnprintf*
semctl	sigpending	tcgetpgrp	vsscanf*
semget	sigprocmask	tcgetsid	vswscanf*
semop	sigrelse	tcsendbreak	vwscanf*
setcat*	sigsend	tcsetattr	wait
setcontext	sigsendset	tcsetpgrp	waitid
setgid	sigset	tdelete	waitpid
setgroups	sigsetjmp	tell#	wcstof*
setlabel#	sigsuspend	telldir	wcstold*
setpgid	sigwait*	tempnam	wcswidth*
setpgrp	sleep	tfind	wcwidth*
setrlimit	snprintf*	time	wordexp*
setsid	statvfs	times	wordfree*
setuid	stime	toascii	write
shmat	strdup	tsearch	writev

*Function is new to the Fourth Edition of the ABI. M
Function is DEPRECATED X

NOTE Other than for use in streams devices, the specific devices supported by `ioctl` are processor specific. |

`libc` in the *System V Application Binary Interface, Edition 4.1* also contains the following routines which are conformant to the System Interfaces and Headers volume of the *X/Open CAE Specification Guide, Issue 4.2* (see the Conformance Rule in Chapter 1). This table lists routines from the ANSI C standard. X

Figure 6-4: `libc` **Contents from XSH4.2, Names without Synonyms** X

advance#	erand48	lrand48	nrand48	srand48	X
compile#	jrand48	mrand48	seed48	step#	X
drand48	lcong48				X

Function is DEPRECATED X

Additionally, libc holds the following services (not contained in the ANSI C standard).

Figure 6-5: libc Contents from XSH4.2, Names with Synonyms

_longjmp	dbm_store	getpwent	re_comp	setstate
_setjmp	dirname	getrusage	re_exec	setutxent
a64l	ecvt	gettimeofday	realpath	siginterrupt
basename	endgrent	getutxent	regcmp	signgam
bcmp	endpwent	getutxid	regex	sigstack
bcopy	endutxent	getutxline	regexp	srandom
brk	fcvt	getwd	remque	strcasecmp
bsd_signal	ffs	ilogb	rindex	strncasecmp
bzero	ftime	index	rint	syslog
closelog	ftruncate	initstate	sbrk	truncate
dbm_clearerr	ftw	insque	select	ttyslot
dbm_close	gcvt	killpg	setgrent	ualarm
dbm_delete	getdtablesize	l64a	setitimer	usleep
dbm_error	getgrent	log1p	setlogmask	utimes
dbm_fetch	gethostid	mkstemp	setpriority	valloc
dbm_firstkey	getitimer	openlog	setpwent	vfork
dbm_nextkey	getpagesize	pututxline	setregid	wait3
dbm_open	getpriority	random	setreuid	wcswcs

The routines listed in the table below are needed by the library to work properly. These routines should not be used by applications. E

Figure 6-6: libc Contents, Internal Names without Synonyms

_cleanup	_xftw	_ _flsbuf	_ _trwctype*
_tolower	_ _assert	_ _iswctype*	_ _ _errno*
_toupper	_ _filbuf	_ _thr_errno*	

*Function is new to the Fourth Edition of the ABI. M

Of the routines listed in the table above, the following are not defined elsewhere.

`void _cleanup(void);` E

Functionally equivalent to `fflush(NULL)`.

`int __filbuf(FILE *f);`

This function returns the next input character for `f`, filling its buffer as appropriate. `__filbuf` is intended to be called when its buffer is empty, as it will return the first character added to the buffer, whatever the content of the buffer when the call to `__filbuf` is made. It returns `EOF` if an error occurs.

`int __flsbuf(int x, FILE *f);`

This function flushes the output characters for `f` as if `putc(x,f)` had been called and then appends the value of `x` to the resulting output stream. It returns `EOF` if an error occurs and `x` otherwise.

`int _xftw(int, char *, int (*)(char *, struct stat *, int), int);`

Calls to the `ftw` function are mapped to this function when applications are compiled. This function is identical to `ftw`, except that `_xftw()` takes an interposed first argument, which must have the value 2.

`int * __thr_errno(void);`

This function returns a pointer to the location of the thread-specific errno for a particular thread of control when multiple threads of control are being used in a program. Assignments to errno are mapped to this function when multi-threaded applications are compiled.

`int * ___errno(void);`

Functionally equivalent to `__thr_errno(void)` described above.

`int __iswctype(wint_t wc, wctype_t charclass);`

This function returns 0 if the wide character represented by the `wint_t` argument is a member of the wide character classification, `charclass`, or a non-zero value if it is not. It is the underlying engine for the `isw*` macros and functions.

`wint_t __trwctype(wint_t wc, wctype_t charclass);`

This function returns the value of the wide character `wc`, converted to the corresponding character class as defined by `charclass`, or `wc` if no transformation could be done. It is the underlying engine for the `tow*` macros and functions.

See this chapter's other library sections for more SVID, ANSI C, and POSIX facilities. See ''System Data Interfaces'' later in this chapter for more information.

Additional Entry Points (Processor-Specific)

ABI-conforming systems must provide a `libc` entry point for each of the source-level services shown in the list below. The name and syntax of this entry point may be the same as those characteristics of the source-level service or they may vary across processor architectures. The actual names of the entry points are specified in each processor's supplement to the ABI, together with the entry points' syntax information if names differ from those of the source-level services.

This information is for the use of system implementors and compiler writers, and does not affect the source-level system interface used by application programmers.

Figure 6-7: `libc` **Contents, Additional Services**

`fstat`	`lstat`	`mknod`	`stat`	`uname`

NOTE This section requires processor–specific information. Consequently, the *ABI* supplement for the desired processor describes the details.

NOTE Because the *ABI* specifies neither the correspondence of system calls to traps nor the formats of system data files, ABI–conforming programs access `libc` services through dynamic linking.

See ''System Data Interfaces'' later in this chapter for more information.

Support Routines (Processor-Specific)

NOTE This section requires processor–specific information. The *ABI* supplement for the desired processor describes the details.

Global Data Symbols

The `libc` library requires that some global external data symbols be defined for its routines to work properly. All the data symbols listed in the table below must be provided by the `libc` library.

For formal declarations of the data objects represented by these symbols, see the X
X/Open CAE Specification, Issue 4.2 and the *System V Interface Definition, Fourth Edition* (see the Conformance Rule in chapter 1) or the ‘‘Data Definitions’’ section of Chapter 6 in the appropriate processor supplement to the *System V ABI*.

For entries in the following table that are in *name* - *_name* form, both symbols in each pair represent the same data. The underscore synonyms are provided to
satisfy the ANSI C standard. If the application references a weak symbol that has E
a global synonym, it must define both the weak symbol and the global synonym at E
the same address. An ANSI C-conforming application can define its own *name*
symbols, unrelated to the *X/Open CAE Specification, Issue 4.2* and the *System V* X
Interface Definition, Fourth Edition meanings. If the application intends those sym- X
bols to have the *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition* semantics, it must also define the *_name* symbols so that both refer to the same data object.

Figure 6-8: `libc` **Contents, Global External Data Symbols**

`__ctype`	`_getdate_err`	`daylight`	`loc2`	`optind`	
`__iob`	`_numeric`	`errno`	`locs`	`optopt`	
`__loc1`	`_timezone`	`getdate_err`	`optarg`	`timezone`	
`_altzone`	`_tzname`	`loc1`	`opterr`	`tzname`	
`_daylight`	`altzone`*				
`__xpg4`					X

*Function is new to the Fourth Edition of the ABI. M

`time_t _altzone;`

This variable contains the difference, in seconds, between Universal Coordinated Time and the alternate time zone, as established with the function `tzset`. X

`unsigned char _numeric[2];`

This array holds local-specific information, as established by the
function `setlocale`. Specifically, `_numeric[0]` holds the X
decimal-point character, and `_numeric[1]` holds the character used to separate groups of digits to the left of the decimal-point
character in formatted non-monetary quantities. See the function X
`localeconv` for more information.

`extern int _ _xpg4;`

This variable's value specifies the execution environment for the program. If the value is 0 or unset, results are implementation-specific. Otherwise, if the value is 1, the application will get *System V Application Binary Interface, Edition 4.1* functionality. All other values for _ _xpg4 are reserved for future use.

NOTE Typically, _ _xpg4 being initialized to 0 or uninitialized would represent a backwards compatibility environment for *System V Application Binary Interface, Fourth Edition* applications. X

Application Constraints

As described above, `libc` provides symbols for applications. In a few cases, however, an application is obliged to provide symbols for the library.

`extern char **environ;`

Normally, this symbol is synonymous with `environ`, as the function `exec` describes. This isn't always true, though, because ANSI C does not define `environ`. Thus, an ANSI C-conforming application can define its own `environ` symbol, unrelated to the process environment. If the application defines `environ` and intends it to have the *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition* semantics (see the Conformance Rule in chapter 1), it must also define `_environ` so that the two symbols refer to the same data object. X X X X

`extern const int _lib_version;`

This variable's value specifies the compilation and execution mode for the program. If the value is zero, the program wants to preserve the semantics of older (pre-ANSI) C, where conflicts exist with ANSI. Otherwise, the value is non-zero, and the program wants ANSI C semantics.

Implementation of libc Routines

All ABI-conforming systems must provide a `libc` entry point for all routines
listed as belonging to this library. However, only the routines necessary to pro-
vide the source-level programming interfaces defined in the *X/Open CAE* X
Specification, Issue 4.2 and defined in the *System V Interface Definition, Fourth Edition*
sections BA_OS, BA_LIB, and KE_OS, as described in the introduction to the *Sys-
tem V ABI*, must be implemented on a conforming system. For example, this
means that the routine `memcntl`(RT_OS) need not be fully implemented on an
ABI-conforming system, though the entry point for this function must be present
in the library.

Routines not required for the *System V Interface Definition, Fourth Edition* sections
listed above or *X/Open CAE Specification, Issue 4.2* may or may not be imple- X
mented, at the discretion of the system implementor. Unimplemented routines E
must be represented in the library by a stub that when called causes failure and E
sets the the assignable lvalue `errno` to the value `ENOSYS`.

Vendor Extensions

Besides the services listed above, **libc** may contain other symbols. An ABI-
conforming system vendor may add a symbol to the C library to provide vendor-
specific services. The *ABI* does not define these services, and programs using
these services are not ABI-conforming. Nonetheless, the *ABI* defines a recom- E
mended extension *mechanism*, providing a way to avoid conflict among the ser-
vices from multiple vendors.

A symbol of the form `_$vendor.`*company* provides an operating system entry for
the vendor named *company*. The C library does not have unadorned alternatives
for these names. Conventionally, a vendor uses the single name to provide multi-
ple services, letting the first argument to `_$vendor.`*company* select among the
alternatives. As an example, the ‘‘XYZ Computer Company’’ might add
`_$vendor.xyz` to the C library. M

Threads Library

The Threads Library, `libthread`, contains interfaces supporting multithreaded applications and asynchronous input/output which are defined in the MT extension of the *System V Interface Definition, Fourth Edition*. The multithreading interfaces in the SVID MT extension are semantically similar to those found in the IEEE POSIX 1003.1c 1995 draft standard (formerly known as 1003.4a or the Pthreads standard), although the MT extension is more powerful in terms of the synchronization and thread control interfaces provided. It is anticipated that a future version of the ABI and SVID will contain all interfaces in the completed POSIX multithreading standard.

The following multithreading functions reside in `libthread` and must be provided on all ABI-conforming systems.

Figure 6-9: `libthread` **Contents** *, **Part 1 of 2**

barrier_destroy	rmutex_unlock	thr_getconcurrency
barrier_init	rw_rdlock	thr_getprio
barrier_wait	rw_tryrdlock	thr_getscheduler
cond_broadcast	rw_trywrlock	thr_getspecific
cond_destroy	rw_unlock	thr_join
cond_init	rw_wrlock	thr_keycreate
cond_signal	rwlock_destroy	thr_keydelete
cond_timedwait	rwlock_init	thr_kill
cond_wait	sema_destroy	thr_minstack
mutex_destroy	sema_init	thr_self
mutex_init	sema_post	thr_setconcurrency
mutex_lock	sema_trywait	thr_setprio
mutex_trylock	sema_wait	thr_setscheduler
mutex_unlock	thr_continue	thr_setspecific
rmutex_destroy	thr_create	thr_sigsetmask
rmutex_init	thr_exit	thr_suspend
rmutex_lock	thr_get_rr_interval	thr_yield
rmutex_trylock		

* All functions in this table are new to the Fourth Edition of the ABI.

The following functions to support asynchronous I/O also reside in `libthread` and must be provided on all ABI-conforming systems.

Figure 6-10: `libthread` **Contents** *, **Part 2 of 2**

`aio_cancel`	`aio_read`	`aio_suspend`	`lio_listio`
`aio_error`	`aio_return`	`aio_write`	

* All functions in this table are new to the Fourth Edition of the ABI. M

Dynamic Linking Library M

The Dynamic Linking library, `libdl`, contains the library routines that provide the user direct access to the dynamic linking facilities. The following functions reside in `libdl` and must be provided on all ABI-conforming systems. M

Figure 6-11: `libdl` **Contents** * M

`dlclose`	`dlerror`	`dlopen`	`dlsym`	M

* All functions in this table are new to the Fourth Edition of the ABI. M

Network Services Library

The Network Services library, `libnsl`, contains library routines that provide a transport-level interface to networking services for applications, facilities for machine-independent data representation, a remote procedure call mechanism, and other networking services useful for application programs. This library contains two sets of interfaces: one conforming to the Transport Level Interface (TLI) specification and another to the *X/Open Transport Interface (XTI) Version 2 specification*. The TLI interface allows for binary compatibility with systems that conform to previous editions of the ABI.

The following TLI conforming functions reside in `libnsl` and must be provided on all ABI-conforming systems.

Figure 6-12: `libnsl` **Contents, Part 1 of 3**

t_accept	t_listen	t_rcvudata
t_alloc	t_look	t_rcvuderr
t_bind	t_open	t_snd
t_close	t_optmgmt	t_snddis
t_connect	t_rcv	t_sndrel
t_error	t_rcvconnect	t_sndudata
t_free	t_rcvdis	t_sync
t_getinfo	t_rcvrel	t_unbind
t_getstate		

The following XTI interfaces also reside in `libnsl` and must be provided on all ABI-conforming systems.

Figure 6-13: `libnsl` **Contents** *, **Part 2 of 3**

_xti_accept	_xti_getinfo	_xti_rcvconnect
_xti_alloc	_xti_getprotaddr	_xti_rcvdis
_xti_bind	_xti_getstate	_xti_rcvrel
_xti_close	_xti_listen	_xti_rcvudata
_xti_connect	_xti_look	_xti_rcvuderr
_xti_error	_xti_open	_xti_snd
_xti_free	_xti_rcv	_xti_snddis

Figure 6-13: `libnsl` **Contents** *, **Part 2 of 3** (continued)

_xti_sndrel	_xti_strerror	_xti_unbind
_xti_sndudata	_xti_sync	

* All functions in this table are new to the Fourth Edition of the ABI. M

In addition, the following functions must be provided in `libnsl` on all ABI-conforming systems with a networking capability. Systems with no networking capability are not required to implement these functions, but must provide an entry point into `libnsl` for each of the following function names. Functions that E
are present only as stubs and are not implemented must fail normally and per- E
form no action other setting the external `errno` variable to the value `ENOSYS` when they are called by an application.

Figure 6-14: `libnsl` **Contents, Part 3 of 3**

authdes_getucred	getpublickey	rpc_broadcast_exp*
authdes_seccreate	getsecretkey	rpc_call
authnone_create	host2netname	rpc_reg
authsys_create	key_decryptsession	setnetconfig
authsys_create_default	key_encryptsession	setnetpath
clnt_create	key_gendes	svcerr_auth
clnt_dg_create	key_setsecret	svcerr_decode
clnt_pcreateerror	nc_perror	svcerr_noproc
clnt_perrno	nc_sperror*	svcerr_noprog
clnt_perror	netdir_free	svcerr_progvers
clnt_raw_create	netdir_getbyaddr	svcerr_systemerr
clnt_spcreateerror	netdir_getbyname	svcerr_weakauth
clnt_sperrno	netdir_options	svc_create
clnt_sperror	netdir_perror*	svc_dg_create
clnt_tli_create	netdir_sperror*	svc_fd_create
clnt_tp_create	netname2host	svc_getreqset
clnt_vc_create	netname2user	svc_getreq_common*
endnetconfig	rpcb_getaddr	svc_getreq_poll*
endnetpath	rpcb_getmaps	svc_raw_create
freenetconfigent	rpcb_gettime	svc_reg
getnetconfig	rpcb_rmtcall	svc_run
getnetconfigent	rpcb_set	svc_sendreply
getnetname	rpcb_unset	svc_tli_create
getnetpath	rpc_broadcast	svc_tp_create

Figure 6-14: `libnsl` **Contents, Part 3 of 3** (continued)

svc_unreg	xdr_bytes	xdr_rejected_reply
svc_vc_create	xdr_callhdr	xdr_replymsg
taddr2uaddr	xdr_callmsg	xdr_short
uaddr2taddr	xdr_char	xdr_string
user2netname	xdr_double	xdr_union
xdrmem_create	xdr_enum	xdr_u_char
xdrrec_create	xdr_float	xdr_u_int*
xdrrec_endofrecord*	xdr_free	xdr_u_long
xdrrec_eof	xdr_int	xdr_u_short
xdrrec_skiprecord*	xdr_long	xdr_vector
xdrstdio_create	xdr_opaque	xdr_void
xdr_accepted_reply	xdr_opaque_auth	xdr_wrapstring
xdr_array	xdr_pointer	xprt_register
xdr_authsys_parms	xdr_reference	xprt_unregister
xdr_bool		

*Function is new to the Fourth Edition of the ABI. M

Figure 6-15: `libnsl` **Contents, Global External Data Symbols**

rpc_createerr	svc_fdset	t_errno	_nderror

See ''Data Definitions'' later in this chapter for more information.

Socket Library

The socket library, `libsocket`, contains the socket functions as described in the Networking Services volume of the *X/Open CAE Specification, Issue 4.2*. This library is required for all ABI-conforming systems.

Figure 6-16: `libsocket` **Contents, Part 1 of 2**

`accept`	`listen`	`sendmsg`
`bind`	`pool`	`sendto`
`connect`	`recv`	`setsockopt`
`getpeername`	`recvfrom`	`shutdown`
`getsockname`	`recvmsg`	`socket`
`getsockopt`	`send`	`socketpair`

The socket library, `libsocket`, also contains these IP Address Resolution functions as described in the Networking Services volume of the *X/Open CAE Specification, Issue 4.2*.

Figure 6-18: `libsocket` **Contents, Part 2 of 2**

`endhostent`	`getprotobyname`	`inet_makeaddr`
`endnetent`	`getprotobynumber`	`inet_netof`
`endprotoent`	`getprotoent`	`inet_network`
`endservent`	`getservbyname`	`inet_ntoa`
`gethostbyaddr`	`getservbyport`	`ntohl`
`gethostbyname`	`getservent`	`ntohs`
`gethostent`	`htonl`	`sethostent`
`gethostname`	`htons`	`setnetent`
`getnetbyaddr`	`inet_addr`	`setprotoent`
`getnetbyname`	`inet_lnaof`	`setservent`
`getnetent`		

Curses Library

The curses library, `libcurses`, contains functions that update character screens as X
described in the Curses volume of the *X/Open CAE Specification, Issue 4.2*. X

Figure 6-19: `libcurses` **Contents** X

addch	getnstr	mvgetnstr	X
addchnstr	getn_wstr	mvgetn_wstr	X
addchstr	getparyx	mvgetstr	X
addnstr	getstr	mvget_wch	X
addnwstr	getwin	mvget_wstr	X
addstr	getyx	mvhline	X
addwstr	get_wch	mvhline_set	X
add_wch	get_wstr	mvinch	X
add_wchnstr	halfdelay	mvinchnstr	X
add_wchstr	has_colors	mvinchstr	X
attroff	has_ic	mvinnstr	X
attron	has_il	mvinnwstr	X
attrset	hline	mvinsch	X
attr_get	hline_set	mvinsnstr	X
attr_off	idcok	mvinsstr	X
attr_on	idlok	mvinstr	X
attr_set	immedok	mvins_nwstr	X
baudrate	inch	mvins_wch	X
beep	inchnstr	mvins_wstr	X
bkgd	inchstr	mvinwstr	X
bkgdset	initscr	mvin_wch	X
bkgrnd	init_color	mvin_wchnstr	X
bkgrndset	init_pair	mvin_wchstr	X
border	innstr	mvprintw	X
border_set	innwstr	mvscanw	X
box	insch	mvvline	X
box_set	insdelln	mvvline_set	X
can_change_color	insertln	mvwaddch	X
cbreak	insnstr	mvwaddchnstr	X
chgat	insstr	mvwaddchstr	X
clear	instr	mvwaddnstr	X
clearok	ins_nwstr	mvwaddnwstr	X

Figure 6-19: `libcurses` **Contents** (continued) X

clrtobot	ins_wch	mvwaddstr	X
clrtoeol	ins_wstr	mvwaddwstr	X
color_content	intrflush	mvwadd_wch	X
copywin	inwstr	mvwadd_wchnstr	X
curscr	in_wch	mvwadd_wchstr	X
curs_set	in_wchnstr	mvwchgat	X
cur_term	in_wchstr	mvwdelch	X
def_prog_mode	isendwin	mvwgetch	X
def_shell_mode	is_linetouched	mvwgetnstr	X
delay_output	is_wintouched	mvwgetn_wstr	X
delch	keyname	mvwgetstr	X
deleteln	keypad	mvwget_wch	X
delscreen	key_name	mvwget_wstr	X
delwin	killchar	mvwhline	X
del_curterm	killwchar	mvwhline_set	X
derwin	leaveok	mvwin	X
doupdate	longname	mvwinch	X
dupwin	meta	mvwinchnstr	X
echo	move	mvwinchstr	X
echochar	mvaddch	mvwinnstr	X
echo_wchar	mvaddchnstr	mvwinnwstr	X
endwin	mvaddchstr	mvwinsch	X
erase	mvaddnstr	mvwinsnstr	X
erasechar	mvaddnwstr	mvwinsstr	X
erasewchar	mvaddstr	mvwinstr	X
filter	mvaddwstr	mvwins_nwstr	X
flash	mvadd_wch	mvwins_wch	X
flushinp	mvadd_wchnstr	mvwins_wstr	X
getbegyx	mvadd_wchstr	mvwinwstr	X
getbkgd	mvchgat	mvwin_wch	X
getbkgrnd	mvcur	mvwin_wchnstr	X
getcchar	mvdelch	mvwin_wchstr	X
getch	mvderwin	mvwprintw	X
getmaxyx	mvgetch	mvwscanw	X
mvwvline	slk_set	wbkgdset	X
mvwvline_set	slk_touch	wbkgrnd	X
napms	slk_wset	wbkgrndset	X
newpad	standend	wborder	X

Figure 6-19: `libcurses` **Contents** (continued) X

newterm	standout	wborder_set	X
newwin	start_color	wchgat	X
nl	stdscr	wclear	X
no	subpad	wclrtobot	X
nocbreak	subwin	wclrtoeol	X
nodelay	syncok	wcursyncup	X
noecho	termattrs	wdelch	X
nonl	termname	wdeleteln	X
noqiflush	tgetent#	wechochar	X
noraw	tgetflag#	wecho_wchar	X
notimeout	tgetnum#	werase	X
overlay	tgetstr#	wgetbkgrnd	X
overwrite	tgoto#	wgetch	X
pair_content	tigetflag	wgetnstr	X
pechochar	tigetnum	wgetn_wstr	X
pecho_wchar	tigetstr	wgetstr	X
pnoutrefresh	timeout	wget_wch	X
prefresh	touchline	wget_wstr	X
printw	touchwin	whline	X
putp	tparm	whline_set	X
putwin	tputs	winch	X
qiflush	typeahead	winchnstr	X
raw	unctrl	winchstr	X
redrawwin	ungetch	winnstr	X
refresh	unget_wch	winnwstr	X
resetty	untouchwin	winsch	X
reset_prog_mode	use_env	winsdelln	X
reset_shell_mode	vidattr	winsertln	X
restartterm	vidputs	winsnstr	X
ripoffline	vid_attr	winsstr	X
savetty	vid_puts	winstr	X
scanw	vline	wins_nwstr	X
scrl	vline_set	wins_wch	X
scroll	vwprintw#	wins_wstr	X
scrollok	vwscanw#	winwstr	X
scr_dump	vw_printw	win_wch	X
scr_init	vw_scanw	win_wchnstr	X
scr_restore	waddch	win_wchstr	X

Figure 6-19: `libcurses` **Contents** (continued) X

scr_set	waddchnstr	wmove	X
setcchar	waddchstr	wnoutrefresh	X
setscrreg	waddnstr	wprintw	X
setupterm	waddnwstr	wredrawln	X
set_curterm	waddstr	wrefresh	X
set_term	waddwstr	wscanw	X
slk_attroff	wadd_wch	wscrl	X
slk_attron	wadd_wchnstr	wsetscrreg	X
slk_attrset	wadd_wchstr	wstandend	X
slk_attr_off	wattroff	wstandout	X
slk_attr_on	wattron	wsyncdown	X
slk_attr_set	wattrset	wsyncup	X
slk_clear	wattr_get	wtimeout	X
slk_init	wattr_off	wtouchln	X
slk_label	wattr_on	wunctrl	X
slk_noutrefresh	wattr_set	wvline	X
slk_refresh	wbkgd	wvline_set	X
slk_restore			X

Function is DEPRECATED X

Figure 6-20: `libcurses` **Contents, Global External Data Symbols**

bit_attributes	cur_strs	strfnames	X
boolcodes	Mouse_status	strnames	X
boolfnames	numcodes	TABSIZE	X
boolnames	numfnames	term_errno	X
curs_errno	numnames	term_parm_err	X
curs_parm_err	outchcount	ttytype	X
cur_bools	SP	wcolor_set	X
cur_nums	strcodes		X

X Window System Library

The X Window System library, `libX`, contains library routines that provide primitives for the operation of the X Window System. This library is required for all ABI-conforming systems that implement a graphical windowing terminal interface. See Chapter 10 of the *System V ABI*, ''Windowing and Terminal Interfaces'' for more information on windowing software requirements on ABI-conforming systems.

The following functions reside in `libX` and must be provided on all ABI-conforming systems.

Figure 6-21: `libX` **Contents**

XActivateScreenSaver
XAddExtension
XAddHost
XAddHosts
XAddPixel
XAddToExtensionList
XAddToSaveSet
XAllocClassHint
XAllocColor
XAllocColorCells
XAllocColorPlanes
XAllocIconSize
XAllocNamedColor
XAllocSizeHints
XAllocStandardColormap
XAllocWMHints
XAllowEvents
XAllPlanes
XAutoRepeatOff
XAutoRepeatOn
XBaseFontNameListOfFontSet
XBell
XBitmapBitOrder
XBitmapPad
XBitmapUnit
XBlackPixel
XBlackPixelOfScreen
XCellsOfScreen
XChangeActivePointerGrab
XChangeGC
XChangeKeyboardControl
XChangeKeyboardMapping
XChangePointerControl
XChangeProperty
XChangeSaveSet
XChangeWindowAttributes
XCheckIfEvent
XCheckMaskEvent
XCheckTypedEvent
XCheckTypedWindowEvent
XCheckWindowEvent
XCirculateSubwindows
XCirculateSubwindowsDown
XCirculateSubwindowsUp
XClearArea
XClearWindow
XClipBox
XCloseDisplay
XCloseIM
XcmsAddColorSpace
XcmsAddFunctionSet
XcmsAllocColor
XcmsAllocNamedColor
XcmsCCCofColormap
XcmsCIELabClipab*
XcmsCIELabClipLab*
XcmsCIELabClipL*
XcmsCIELabQueryMaxC
XcmsCIELabQueryMaxL
XcmsCIELabQueryMaxLC
XcmsCIELabQueryMinL
XcmsCIELabToCIEXYZ
XcmsCIELabWhiteShiftColors*
XcmsCIELuvClipLuv*
XcmsCIELuvClipL*
XcmsCIELuvClipuv*
XcmsCIELuvQueryMaxC
XcmsCIELuvQueryMaxL
XcmsCIELuvQueryMaxLC
XcmsCIELuvQueryMinL
XcmsCIELuvToCIEuvY
XcmsCIELuvWhiteShiftColors*
XcmsCIEuvYToCIELuv
XcmsCIEuvYToCIEXYZ
XcmsCIEuvYToTekHVC
XcmsCIExyYToCIEXYZ
XcmsCIEXYZToCIELab
XcmsCIEXYZToCIEuvY
XcmsCIEXYZToCIExyY
XcmsCIEXYZToRGBi
XcmsClientWhitePointOfCCC
XcmsConvertColors
XcmsCreateCCC
XcmsDefaultCCC
XcmsDisplayOfCCC
XcmsFormatOfPrefix
XcmsFreeCCC
XcmsLookupColor
XcmsPrefixOfFormat
XcmsQueryBlack
XcmsQueryBlue
XcmsQueryColor
XcmsQueryColors
XcmsQueryGreen
XcmsQueryRed
XcmsQueryWhite
XcmsRGBiToCIEXYZ
XcmsRGBiToRGB
XcmsRGBToRGBi
XcmsScreenNumberOfCCC
XcmsScreenWhitePointOfCCC
XcmsSetCompressionProc
XcmsSetWhiteAdjustProc
XcmsSetWhitePoint
XcmsStoreColor
XcmsStoreColors
XcmsTekHVCClipC*
XcmsTekHVCClipVC*
XcmsTekHVCClipV*
XcmsTekHVCQueryMaxC
XcmsTekHVCQueryMaxV
XcmsTekHVCQueryMaxVC
XcmsTekHVCQueryMaxVSamples
XcmsTekHVCQueryMinV

Figure 6-21: `libX` **Contents** (continued)

XcmsTekHVCToCIEuvY
XcmsTekHVCWhiteShiftColors*
XcmsVisualOfCCC
XConfigureWindow
XConnectionNumber
XContextDependentDrawing
XConvertSelection
XCopyArea
XCopyColormapAndFree
XCopyGC
XCopyPlane
XCreateBitmapFromData
XCreateColormap
XCreateFontCursor
XCreateFontSet
XCreateGC
XCreateGlyphCursor
XCreateIC
XCreateImage
XCreatePixmap
XCreatePixmapCursor
XCreatePixmapFromBitmapData
XCreateRegion
XCreateSimpleWindow
XCreateWindow
XDefaultColormap
XDefaultColormapOfScreen
XDefaultDepth
XDefaultDepthOfScreen
XDefaultGC
XDefaultGCOfScreen
XDefaultRootWindow
XDefaultScreen
XDefaultScreenOfDisplay
XDefaultString
XDefaultVisual
XDefaultVisualOfScreen
XDefineCursor
XDeleteContext
XDeleteModifiermapEntry
XDeleteProperty
XDestroyIC
XDestroyImage
XDestroyRegion
XDestroySubwindows
XDestroyWindow
XDisableAccessControl
XDisplayCells
XDisplayHeight
XDisplayHeightMM
XDisplayKeycodes
XDisplayMotionBufferSize
XDisplayName
XDisplayOfIM
XDisplayOfScreen
XDisplayPlanes
XDisplayString
XDisplayWidth
XDisplayWidthMM
XDoesBackingStore
XDoesSaveUnders
XDrawArc
XDrawArcs
XDrawImageString
XDrawImageString16
XDrawLine
XDrawLines
XDrawPoint
XDrawPoints
XDrawRectangle
XDrawRectangles
XDrawSegments
XDrawString
XDrawString16
XDrawText
XDrawText16
XEmptyRegion
XEnableAccessControl
XEqualRegion
XEventMaskOfScreen
XEventsQueued
XExtentsOfFontSet
XFetchBuffer
XFetchBytes
XFetchName
XFillArc
XFillArcs
XFillPolygon
XFillRectangle
XFillRectangles
XFilterEvent
XFindContext
XFindOnExtensionList*
XFlush
XFlushGC
XFontsOfFontSet
XForceScreenSaver
XFree
XFreeColormap
XFreeColors
XFreeCursor
XFreeExtensionList
XFreeFont
XFreeFontInfo
XFreeFontNames
XFreeFontPath
XFreeFontSet
XFreeGC
XFreeModifiermap
XFreePixmap
XFreeStringList
XGContextFromGC
XGeometry
XGetAtomName

Figure 6-21: `libX` **Contents** (continued)

XGetClassHint	XGetWMName	XLookupKeysym
XGetCommand	XGetWMNormalHints	XLookupString
XGetDefault	XGetWMProtocols	XLowerWindow
XGetErrorDatabaseText	XGetWMSizeHints	XMapRaised
XGetErrorText	XGrabButton	XMapSubwindows
XGetFontPath	XGrabKey	XMapWindow
XGetFontProperty	XGrabKeyboard	XMaskEvent
XGetGCValues	XGrabPointer	XMatchVisualInfo
XGetGeometry	XGrabServer	XMaxCmapsOfScreen
XGetIconName	XHeightMMOfScreen	XMaxRequestSize
XGetIconSizes	XHeightOfScreen	XmbDrawImageString
XGetICValues	XIconifyWindow	XmbDrawString
XGetImage	XIfEvent	XmbDrawText
XGetIMValues	XImageByteOrder	XmbLookupString
XGetInputFocus	XIMOfIC	XmbResetIC
XGetKeyboardControl	XInitExtension	XmbSetWMProperties
XGetKeyboardMapping	XInsertModifiermapEntry	XmbTextEscapement
XGetModifierMapping	XInstallColormap	XmbTextExtents
XGetMotionEvents	XInternAtom	XmbTextListToTextProperty
XGetNormalHints	XIntersectRegion	XmbTextPerCharExtents
XGetPixel	XKeycodeToKeysym	XmbTextPropertyToTextList
XGetPointerControl	XKeysymToKeycode	XMinCmapsOfScreen
XGetPointerMapping	XKeysymToString	XMoveResizeWindow
XGetRGBColormaps	XKillClient	XMoveWindow
XGetScreenSaver	XLastKnownRequestProcessed	XNewModifiermap
XGetSelectionOwner	XListDepths	XNextEvent
XGetSizeHints	XListExtensions	XNextRequest
XGetStandardColormap	XListFonts	XNoOp
XGetSubImage	XListFontsWithInfo	XOffsetRegion
XGetTextProperty	XListHosts	XOpenDisplay
XGetTransientForHint	XListInstalledColormaps	XOpenIM
XGetVisualInfo	XListPixmapFormats	XParseColor
XGetWindowAttributes	XListProperties	XParseGeometry
XGetWindowProperty	XLoadFont	XPeekEvent
XGetWMClientMachine	XLoadQueryFont	XPeekIfEvent
XGetWMColormapWindows	XLocaleOfFontSet	XPending
XGetWMHints	XLocaleOfIM	Xpermalloc
XGetWMIconName	XLookupColor	XPlanesOfScreen

Figure 6-21: `libX` **Contents** (continued)

XPointInRegion	XrmDestroyDatabase	XSetAccessControl
XPolygonRegion	XrmEnumerateDatabase	XSetAfterFunction
XProtocolRevision	XrmGetDatabase	XSetArcMode
XProtocolVersion	XrmGetFileDatabase	XSetBackground
XPutBackEvent	XrmGetResource	XSetClassHint
XPutImage	XrmGetStringDatabase	XSetClipMask
XPutPixel	XrmInitialize	XSetClipOrigin
XQLength	XrmLocaleOfDatabase	XSetClipRectangles
XQueryBestCursor	XrmMergeDatabases	XSetCloseDownMode
XQueryBestSize	XrmParseCommand	XSetCommand
XQueryBestStipple	XrmPermStringToQuark	XSetDashes
XQueryBestTile	XrmPutFileDatabase	XSetErrorHandler
XQueryColor	XrmPutLineResource	XSetFillRule
XQueryColors	XrmPutResource	XSetFillStyle
XQueryExtension	XrmPutStringResource	XSetFont
XQueryFont	XrmQGetResource	XSetFontPath
XQueryKeymap	XrmQGetSearchList	XSetForeground
XQueryPointer	XrmQGetSearchResource	XSetFunction
XQueryTextExtents	XrmQPutResource	XSetGraphicsExposures
XQueryTextExtents16	XrmQPutStringResource	XSetICFocus
XQueryTree	XrmQuarkToString	XSetIconName
XRaiseWindow	XrmSetDatabase	XSetIconSizes
XReadBitmapFile	XrmStringToBindingQuarkList	XSetICValues
XRebindKeysym	XrmStringToQuark	XSetInputFocus
XRecolorCursor	XrmStringToQuarkList	XSetIOErrorHandler
XReconfigureWMWindow	XrmUniqueQuark	XSetLineAttributes
XRectInRegion	XRootWindow	XSetLocaleModifiers
XRefreshKeyboardMapping	XRootWindowOfScreen	XSetModifierMapping
XRemoveFromSaveSet	XRotateBuffers	XSetNormalHints
XRemoveHost	XRotateWindowProperties	XSetPlaneMask
XRemoveHosts	XSaveContext	XSetPointerMapping
XReparentWindow	XScreenCount	XSetRegion
XResetScreenSaver	XScreenNumberOfScreen	XSetRGBColormaps
XResizeWindow	XScreenOfDisplay	XSetScreenSaver
XResourceManagerString	XScreenResourceString	XSetSelectionOwner
XRestackWindows	XSelectInput	XSetSizeHints
XrmCombineDatabase	XSendEvent	XSetStandardColormap
XrmCombineFileDatabase	XServerVendor	XSetStandardProperties

Figure 6-21: `libX` **Contents** (continued)

XSetState	XStoreName	XUnmapWindow
XSetStipple	XStoreNamedColor	XUnsetICFocus
XSetSubwindowMode	XStringListToTextProperty	XVaCreateNestedList
XSetTextProperty	XStringToKeysym	XVendorRelease
XSetTile	XSubImage	XVisualIDFromVisual
XSetTransientForHint	XSubtractRegion	XWarpPointer
XSetTSOrigin	XSupportsLocale	XwcDrawImageString
XSetWindowBackground	XSync	XwcDrawString
XSetWindowBackgroundPixmap	XSynchronize	XwcDrawText
XSetWindowBorder	XTextExtents	XwcFreeStringList
XSetWindowBorderPixmap	XTextExtents16	XwcLookupString
XSetWindowBorderWidth	XTextPropertyToStringList	XwcResetIC
XSetWindowColormap	XTextWidth	XwcTextEscapement
XSetWMClientMachine	XTextWidth16	XwcTextExtents
XSetWMColormapWindows	XTranslateCoordinates	XwcTextListToTextProperty
XSetWMHints	XUndefineCursor	XwcTextPerCharExtents
XSetWMIconName	XUngrabButton	XwcTextPropertyToTextList
XSetWMName	XUngrabKey	XWhitePixel
XSetWMNormalHints	XUngrabKeyboard	XWhitePixelOfScreen
XSetWMProperties	XUngrabPointer	XWidthMMOfScreen
XSetWMProtocols	XUngrabServer	XWidthOfScreen
XSetWMSizeHints	XUninstallColormap	XWindowEvent
XShrinkRegion	XUnionRectWithRegion	XWithdrawWindow
XStoreBuffer	XUnionRegion	XWMGeometry
XStoreBytes	XUnloadFont	XWriteBitmapFile
XStoreColor	XUnmapSubwindows	XXorRegion
XStoreColors		

*Function is new to the Fourth Edition of the ABI. M

The table below contains callback functions which are well known entry points in libX. These can be defined and used by ABI conforming applications provided that their definition and use complies with the X11R5 graphical user interface specification.

Figure 6-22: `libX` **Contents, Callback Function Names**

`GeometryCallback`	`PreeditDrawCallback`	`StatusDrawCallback`
`PreeditCaretCallback`	`PreeditStartCallback`	`StatusStartCallback`
`PreeditDoneCallback`	`StatusDoneCallback`	

The `libX` library requires that some global external data symbols be defined for its routines to work properly. All the data symbols listed in the table below must be provided by the `libX` library.

For formal declarations of the data objects represented by these symbols, see the ''Data Definitions'' section of Chapter 6 in the appropriate processor supplement to the *System V ABI*.

Figure 6-23: `libX` **Contents** *, **Global External Data Symbols**

`XcmsCIELabColorSpace`	`XcmsLinearRGBFunctionSet`
`XcmsCIELuvColorSpace`	`XcmsRGBColorSpace`
`XcmsCIEuvYColorSpace`	`XcmsRGBiColorSpace`
`XcmsCIExyYColorSpace`	`XcmsTekHVCColorSpace`
`XcmsCIEXYZColorSpace`	`XcmsUNDEFINEDColorSpace`

* All functions in this table are new to the Fourth Edition of the ABI.

X11 Nonrectangular Window Shape Extension Library

The X11 Nonrectangular Window Shape Extension Library Version 1.0, `libXext`, contains library routines that provide mechanisms for changing the visible shape of a window to an arbitrary, possibly disjoint, nonrectangular form. This library is required for all ABI-conforming systems that implement a graphical windowing terminal interface. See Chapter 10 of the *System V ABI*, ''Windowing and Terminal Interfaces'' for more information on windowing software requirements on ABI-conforming systems.

The following functions reside in `libXext` and must be provided on all ABI-conforming systems that implement a graphical windowing terminal interface.

Figure 6-24: `libXext` **Contents** *

`XShapeCombineMask`	`XShapeGetRectangles`	`XShapeQueryExtents`
`XShapeCombineRectangles`	`XShapeInputSelected`	`XShapeQueryVersion`
`XShapeCombineRegion`	`XShapeOffsetShape`	`XShapeSelectInput`
`XShapeCombineShape`	`XShapeQueryExtension`	

* All functions in this table are new to the Fourth Edition of the ABI.

X Toolkit Intrinsics Library

The X Toolkit Intrinsics library, `libXt`, contains library routines that provide primitives for the operation of the X Toolkit Intrinsics of the X Window System. This library is required for all ABI-conforming systems that implement a graphical windowing terminal interface. See Chapter 10 of the *System V ABI*, ''Windowing and Terminal Interfaces'' for more information on windowing software requirements on ABI-conforming systems.

The following functions reside in `libXt` and must be provided on all ABI-conforming systems that implement a graphical windowing terminal interface.

Figure 6-25: `libXt` **Contents**

XtAddCallback	XtCallbackExclusive	XtCvtStringToFloat
XtAddCallbacks	XtCallbackNone	XtCvtStringToFont
XtAddEventHandler	XtCallbackNonexclusive	XtCvtStringToFontSet
XtAddExposureToRegion	XtCallbackPopdown	XtCvtStringToFontStruct
XtAddGrab	XtCallbackReleaseCacheRef	XtCvtStringToInitialState
XtAddRawEventHandler	XtCallbackReleaseCacheRefList	XtCvtStringToInt
XtAllocateGC	XtCallCallbackList	XtCvtStringToPixel
XtAppAddActionHook	XtCallCallbacks	XtCvtStringToShort
XtAppAddActions	XtCallConverter	XtCvtStringToTranslationTable
XtAppAddInput	XtCalloc	XtCvtStringToUnsignedChar
XtAppAddTimeOut	XtClass	XtCvtStringToVisual
XtAppAddWorkProc	XtCloseDisplay	XtDatabase
XtAppCreateShell	XtConfigureWidget	XtDestroyApplicationContext
XtAppError	XtConvertAndStore	XtDestroyWidget
XtAppErrorMsg	XtConvertCase	XtDisownSelection
XtAppGetErrorDatabase	XtCreateApplicationContext	XtDispatchEvent
XtAppGetErrorDatabaseText	XtCreateManagedWidget	XtDisplay
XtAppGetSelectionTimeout	XtCreatePopupShell	XtDisplayInitialize
XtAppInitialize	XtCreateWidget	XtDisplayOfObject
XtAppMainLoop	XtCreateWindow	XtDisplayStringConversionWarning
XtAppNextEvent	XtCvtColorToPixel	XtDisplayToApplicationContext
XtAppPeekEvent	XtCvtIntToBool	XtFindFile
XtAppPending	XtCvtIntToBoolean	XtFree
XtAppProcessEvent	XtCvtIntToColor	XtGetActionKeysym
XtAppReleaseCacheRefs	XtCvtIntToFloat	XtGetActionList
XtAppSetErrorHandler	XtCvtIntToFont	XtGetApplicationNameAndClass
XtAppSetErrorMsgHandler	XtCvtIntToPixel	XtGetApplicationResources
XtAppSetFallbackResources	XtCvtIntToPixmap	XtGetConstraintResourceList
XtAppSetSelectionTimeout	XtCvtIntToShort	XtGetGC
XtAppSetTypeConverter	XtCvtIntToUnsignedChar	XtGetKeysymTable
XtAppSetWarningHandler	XtCvtStringToAcceleratorTable	XtGetMultiClickTime
XtAppSetWarningMsgHandler	XtCvtStringToAtom	XtGetResourceList
XtAppWarning	XtCvtStringToBool	XtGetSelectionRequest
XtAppWarningMsg	XtCvtStringToBoolean	XtGetSelectionValue
XtAugmentTranslations	XtCvtStringToCursor	XtGetSelectionValueIncremental
XtBuildEventMask	XtCvtStringToDimension	XtGetSelectionValues
XtCallAcceptFocus	XtCvtStringToDisplay	XtGetSelectionValuesIncremental
XtCallActionProc	XtCvtStringToFile	XtGetSubresources

Figure 6-25: `libXt` **Contents** (continued)

XtGetSubvalues	XtMoveWidget	XtSetMappedWhenManaged
XtGetValues	XtName	XtSetMultiClickTime
XtGrabButton	XtNameToWidget	XtSetSensitive
XtGrabKey	XtOpenDisplay	XtSetSubvalues
XtGrabKeyboard	XtOverrideTranslations	XtSetTypeConverter
XtGrapPointer	XtOwnSelection	XtSetValues
XtHasCallbacks	XtOwnSelectionIncremental	XtSetWMColormapWindows
XtInitializeWidgetClass	XtParent	XtSuperclass
XtInsertEventHandler	XtParseAcceleratorTable	XtToolkitInitialize
XtInsertRawEventHandler	XtParseTranslationTable	XtTranslateCoords
XtInstallAccelerators	XtPopdown	XtTranslateKey
XtInstallAllAccelerators	XtPopup	XtTranslateKeycode
XtIsApplicationsShell*	XtPopupSpringLoaded	XtUngrabButton
XtIsComposite*	XtQueryGeometry	XtUngrabKey
XtIsConstraint*	XtRealizeWidget	XtUngrabKeyboard
XtIsManaged	XtRealloc	XtUngrabPointer
XtIsObject	XtRegisterCaseConverter	XtUninstallTranslations
XtIsOverrideShell*	XtRegisterGrabAction	XtUnmanageChild
XtIsRealized	XtReleaseGC	XtUnmanageChildren
XtIsRectObj*	XtRemoveActionHook	XtUnmapWidget
XtIsSensitive	XtRemoveAllCallbacks	XtUnrealizeWidget
XtIsShell*	XtRemoveCallback	XtVaAppCreateShell
XtIsSubclass	XtRemoveCallbacks	XtVaAppInitialize
XtIsTopLevelShell*	XtRemoveEventHandler	XtVaCreateArgsList
XtIsTransientShell*	XtRemoveGrab	XtVaCreateManagedWidget
XtIsVendorShell	XtRemoveInput	XtVaCreatePopupShell
XtIsWidget*	XtRemoveRawEventHandler	XtVaCreateWidget
XtIsWMShell*	XtRemoveTimeOut	XtVaGetApplicationResources
XtKeysymToKeycodeList	XtRemoveWorkProc*	XtVaGetSubresources
XtLastTimestampProcessed	XtResizeWidget	XtVaGetSubvalues
XtMakeGeometryRequest	XtResizeWindow	XtVaGetValues
XtMakeResizeRequest	XtResolvePathname	XtVaSetSubvalues
XtMalloc	XtScreen	XtVaSetValues
XtManageChild	XtScreenDatabase	XtWidgetToApplicationContext
XtManageChildren	XtScreenOfObject	XtWindow
XtMapWidget	XtSetKeyboardFocus	XtWindowOfObject
XtMenuPopupAction*	XtSetKeyTranslator	XtWindowToWidget
XtMergeArgLists	XtSetLanguageProc	

*Function is new to the Fourth Edition of the ABI. M

The `libXt` library requires that some global external data objects be defined for the routines to work properly. The data symbols listed in the table below must be provided by the `libXt` library. G G G

For formal declarations of the data objects represented by these symbols, see the ''Data Definitions'' section of Chapter 6 in the appropriate processor supplement to the *System V ABI*. The names of those symbols not defined in the Processor Supplement are reserved for the implementation of `libXt`. G G G G

Figure 6-26: `libXt` **Contents, Global External Data Symbols**

`applicationShellClassRec`	`overrideShellClassRec`	`transientShellWidgetClass`
`applicationShellWidgetClass`	`overrideShellWidgetClass`	`vendorShellClassRec`
`colorConvertArgs`	`rectObjClass`	`vendorShellWidgetClass`
`compositeClassRec`	`rectObjClassRec`	`widgetClass`
`compositeWidgetClass`	`screenConvertArg`	`widgetClassRec`
`constraintClassRec`	`shellClassRec`	`wmShellClassRec`
`constraintWidgetClass`	`shellWidgetClass`	`wmShellWidgetClass`
`coreWidgetClass`	`topLevelShellClassRec`	`XtShellStrings`
`objectClass`	`topLevelShellWidgetClass`	`XtStrings`
`objectClassRec`	`transientShellClassRec`	

NOTE The Data Symbols `XtShellStrings` and `XtStrings` may not be maintained in an upwardly compatible manner. Future treatment of these arrays may be handled differently on each platform. G

Motif Libraries

The libraries `libXm` and `libXMrm` contains library routines that provide the Graphical User Interface components of the Motif interface. The current version of Motif supported by the gABI is Motif 1.2 as defined in OSF’s Motif Programmer Reference Revision 1.2. This library is required for all ABI-conforming systems that implement a graphical windowing terminal interface. See Chapter 10 of the *System V ABI*, ‘‘Windowing and Terminal Interfaces’’ for more information on windowing software requirements on ABI-conforming systems.

The following functions reside in `libXm` and must be provided on all ABI-conforming systems that implement a graphical windowing terminal interface.

Figure 6-27: `libXm` **Contents** *

XmActivateProtocol	XmCreateDrawingArea
XmAddProtocolCallback	XmCreateDrawnButton
XmAddProtocols	XmCreateErrorDialog
XmAddTabGroup	XmCreateFileSelectionBox
XmCascadeButtonGadgetHighlight	XmCreateFileSelectionDialog
XmCascadeButtonHighlight	XmCreateForm
XmChangeColor	XmCreateFormDialog
XmClipboardCancelCopy	XmCreateFrame
XmClipboardCopy	XmCreateInformationDialog
XmClipboardCopyByName	XmCreateLabel
XmClipboardEndCopy	XmCreateLabelGadget
XmClipboardEndRetrieve	XmCreateList
XmClipboardInquireCount	XmCreateMainWindow
XmClipboardInquireFormat	XmCreateMenuBar
XmClipboardInquireLength	XmCreateMenuShell
XmClipboardInquirePendingItems	XmCreateMessageBox
XmClipboardLock	XmCreateMessageDialog
XmClipboardRegisterFormat	XmCreateOptionMenu
XmClipboardRetrieve	XmCreatePanedWindow
XmClipboardStartCopy	XmCreatePopupMenu
XmClipboardStartRetrieve	XmCreatePromptDialog
XmClipboardUndoCopy	XmCreatePulldownMenu
XmClipboardUnlock	XmCreatePushButton
XmClipboardWithdrawFormat	XmCreatePushButtonGadget
XmCommandAppendValue	XmCreateQuestionDialog
XmCommandError	XmCreateRadioBox
XmCommandGetChild	XmCreateRowColumn
XmCommandSetValue	XmCreateScale
XmConvertUnits	XmCreateScrollBar
XmCreateArrowButton	XmCreateScrolledList
XmCreateArrowButtonGadget	XmCreateScrolledText
XmCreateBulletinBoard	XmCreateScrolledWindow
XmCreateBulletinBoardDialog	XmCreateSelectionBox
XmCreateCascadeButton	XmCreateSelectionDialog
XmCreateCascadeButtonGadget	XmCreateSeparator
XmCreateCommand	XmCreateSeparatorGadget
XmCreateDialogShell	XmCreateSimpleCheckBox
XmCreateDragIcon	XmCreateSimpleMenuBar

Figure 6-27: `libXm` **Contents** * (continued)

XmCreateSimpleOptionMenu	XmFontListEntryGetTag
XmCreateSimplePopupMenu	XmFontListEntryLoad
XmCreateSimplePulldownMenu	XmFontListFree
XmCreateSimpleRadioBox	XmFontListFreeFontContext
XmCreateTemplateDialog	XmFontListGetNextFont
XmCreateText	XmFontListInitFontContext
XmCreateTextField	XmFontListNextEntry
XmCreateToggleButton	XmFontListRemoveEntry
XmCreateToggleButtonGadget	XmGetAtomName
XmCreateWarningDialog	XmGetColorCalculation
XmCreateWorkArea	XmGetColors
XmCreateWorkingDialog	XmGetDestination
XmCvtCTToXmString	XmGetDragContext
XmCvtStringToUnitType	XmGetFocusWidget
XmCvtXmStringToCT	XmGetMenuCursor
XmDeactivateProtocol	XmGetPixmap
XmDestroyPixmap	XmGetPixmapByDepth
XmDragCancel	XmGetPostedFromWidget
XmDragStart	XmGetSecondaryResourceData
XmDropSiteConfigureStackingOrder	XmGetTabGroup
XmDropSiteEndUpdate	XmGetTearOffControl
XmDropSiteQueryStackingOrder	XmGetVisibility
XmDropSiteRegister	XmGetXmDisplay
XmDropSiteRetrieve	XmGetXmScreen
XmDropSiteStartUpdate	XmInstallImage
XmDropSiteUnregister	XmInternAtom
XmDropSiteUpdate	XmIsMotifWMRunning
XmDropTransferAdd	XmIsTraversable
XmDropTransferStart	XmListAddItem
XmFileSelectionBoxGetChild	XmListAddItems
XmFileSelectionDoSearch	XmListAddItemsUnselected
XmFontListAdd	XmListAddItemUnselected
XmFontListAppendEntry	XmListDeleteAllItems
XmFontListCopy	XmListDeleteItem
XmFontListCreate	XmListDeleteItems
XmFontListEntryCreate	XmListDeleteItemsPos
XmFontListEntryFree	XmListDeletePos
XmFontListEntryGetFont	XmListDeletePositions

Figure 6-27: `libXm` **Contents** * (continued)

XmListDeselectAllItems	XmRemoveProtocols
XmListDeselectItem	XmRemoveTabGroup
XmListDeselectPos	XmRepTypeAddReverse
XmListGetKbdItemPos	XmRepTypeGetId
XmListGetMatchPos	XmRepTypeGetNameList
XmListGetSelectedPos	XmRepTypeGetRecord
XmListItemExists	XmRepTypeGetRegistered
XmListItemPos	XmRepTypeInstallTearOffModelConverter
XmListPosSelected	XmRepTypeRegister
XmListPosToBounds	XmRepTypeValidValue
XmListReplaceItems	XmResolveAllPartOffsets
XmListReplaceItemsPos	XmResolvePartOffsets
XmListReplaceItemsPosUnselected	XmScaleGetValue
XmListReplaceItemsUnselected	XmScaleSetValue
XmListReplacePositions	XmScrollBarGetValues
XmListSelectItem	XmScrollBarSetValues
XmListSelectPos	XmScrolledWindowSetAreas
XmListSetAddMode	XmScrollVisible
XmListSetBottomItem	XmSelectionBoxGetChild
XmListSetBottomPos	XmSetColorCalculation
XmListSetHorizPos	XmSetFontUnit
XmListSetItem	XmSetFontUnits
XmListSetKbdItemPos	XmSetMenuCursor
XmListSetPos	XmSetProtocolHooks
XmListUpdateSelectedList	XmStringBaseline
XmListYToPos	XmStringByteCompare
XmMainWindowSep1	XmStringCompare
XmMainWindowSep2	XmStringConcat
XmMainWindowSep3	XmStringCopy
XmMainWindowSetAreas	XmStringCreate
XmMapSegmentEncoding	XmStringCreateLocalized
XmMenuPosition	XmStringCreateLtoR
XmMessageBoxGetChild	XmStringCreateSimple
XmOptionButtonGadget	XmStringDirectionCreate
XmOptionLabelGadget	XmStringDraw
XmProcessTraversal	XmStringDrawImage
XmRegisterSegmentEncoding	XmStringDrawUnderline
XmRemoveProtocolCallback	XmStringEmpty

Figure 6-27: `libXm` **Contents** * (continued)

XmStringExtent
XmStringFree
XmStringFreeContext
XmStringGetLtoR
XmStringGetNextComponent
XmStringGetNextSegment
XmStringHasSubstring
XmStringHeight
XmStringInitContext
XmStringLength
XmStringLineCount
XmStringNConcat
XmStringNCopy
XmStringPeekNextComponent
XmStringSegmentCreate
XmStringSeparatorCreate
XmStringWidth
XmTargetsAreCompatible
XmTextClearSelection
XmTextCopy
XmTextCut
XmTextDisableRedisplay
XmTextEnableRedisplay
XmTextFieldClearSelection
XmTextFieldCopy
XmTextFieldCut
XmTextFieldGetBaseline
XmTextFieldGetEditable
XmTextFieldGetInsertionPosition
XmTextFieldGetLastPosition
XmTextFieldGetMaxLength
XmTextFieldGetSelection
XmTextFieldGetSelectionPosition
XmTextFieldGetSelectionWcs
XmTextFieldGetString
XmTextFieldGetStringWcs
XmTextFieldGetSubstring
XmTextFieldGetSubstringWcs
XmTextFieldInsert
XmTextFieldInsertWcs
XmTextFieldPaste
XmTextFieldPosToXY
XmTextFieldRemove
XmTextFieldReplace
XmTextFieldReplaceWcs
XmTextFieldSetAddMode
XmTextFieldSetEditable
XmTextFieldSetHighlight
XmTextFieldSetInsertionPosition
XmTextFieldSetMaxLength
XmTextFieldSetSelection
XmTextFieldSetString
XmTextFieldSetStringWcs
XmTextFieldShowPosition
XmTextFieldXYToPos
XmTextFindString
XmTextFindStringWcs
XmTextGetBaseLine
XmTextGetBaseline
XmTextGetEditable
XmTextGetInsertionPosition
XmTextGetLastPosition
XmTextGetMaxLength
XmTextGetSelection
XmTextGetSelectionPosition
XmTextGetSelectionWcs
XmTextGetSource
XmTextGetString
XmTextGetStringWcs
XmTextGetSubstring
XmTextGetSubstringWcs
XmTextGetTopCharacter
XmTextInsert
XmTextInsertWcs
XmTextPaste
XmTextPosToXY

Figure 6-27: `libXm` **Contents** * (continued)

XmTextRemove	XmToggleButtonGadgetSetState
XmTextReplace	XmToggleButtonGetState
XmTextReplaceWcs	XmToggleButtonSetState
XmTextScroll	XmTrackingEvent
XmTextSetAddMode	XmTrackingLocate
XmTextSetEditable	XmTranslateKey
XmTextSetHighlight	XmUninstallImage
XmTextSetInsertionPosition	XmUpdateDisplay
XmTextSetMaxLength	XmVaCreateSimpleCheckBox
XmTextSetSelection	XmVaCreateSimpleMenuBar
XmTextSetSource	XmVaCreateSimpleOptionMenu
XmTextSetString	XmVaCreateSimplePopupMenu
XmTextSetStringWcs	XmVaCreateSimplePulldownMenu
XmTextSetTopCharacter	XmVaCreateSimpleRadioBox
XmTextShowPosition	XmWidgetGetBaselines
XmTextXYToPos	XmWidgetGetDisplayRect
XmToggleButtonGadgetGetState	

* All functions in this table are new to the Fourth Edition of the ABI. The `libXm` library requires that some global external data objects be defined for the routines to work properly. The data symbols listed in the table below must be provided by the `libXm` library.

For formal declarations of the data objects represented by these symbols, see the ''Data Definitions'' section of Chapter 6 in the appropriate processor supplement to the *System V ABI*. The names of those symbols not defined in the Processor Supplement are reserved for the implementation of `libXm`.

Figure 6-28: `libXm` **Contents** *, **Global External Data Symbols**

vendorShellClassRec	xmExtObjectClass
vendorShellWidgetClass	xmFileSelectionBoxClassRec
xmArrowButtonClassRec	xmFileSelectionBoxWidgetClass
xmArrowButtonGadgetClass	xmFormClassRec
xmArrowButtonGadgetClassRec	xmFormWidgetClass
xmArrowButtonWidgetClass	xmFrameClassRec
xmBulletinBoardClassRec	xmFrameWidgetClass
xmBulletinBoardWidgetClass	xmGadgetClass
xmCascadeButtonClassRec	xmGadgetClassRec
xmCascadeButtonGadgetClass	xmLabelClassRec
xmCascadeButtonGadgetClassRec	xmLabelGadgetClass
xmCascadeButtonGCacheObjClassRec	xmLabelGadgetClassRec
xmCascadeButtonWidgetClass	xmLabelGCacheObjClassRec
xmCommandClassRec	xmLabelWidgetClass
xmCommandWidgetClass	xmListClassRec
xmDesktopClass	xmListWidgetClass
xmDesktopClassRec	xmMainWindowClassRec
xmDialogShellClassRec	xmMainWindowWidgetClass
xmDialogShellExtClassRec	xmManagerClassRec
xmDialogShellExtObjectClass	xmManagerWidgetClass
xmDialogShellWidgetClass	xmMenuShellClassRec
xmDisplayClass	xmMenuShellWidgetClass
xmDisplayClassRec	xmMessageBoxClassRec
xmDragContextClass	xmMessageBoxWidgetClass
xmDragContextClassRec	xmPanedWindowClassRec
xmDragIconClassRec	xmPanedWindowWidgetClass
xmDragIconObjectClass	xmPrimitiveClassRec
xmDragOverShellClassRec	xmPrimitiveWidgetClass
xmDragOverShellWidgetClass	xmProtocolClassRec
xmDrawingAreaClassRec	xmProtocolObjectClass
xmDrawingAreaWidgetClass	xmPushButtonClassRec
xmDrawnButtonClassRec	xmPushButtonGadgetClass
xmDrawnButtonWidgetClass	xmPushButtonGadgetClassRec
xmDropSiteManagerClassRec	xmPushButtonGCacheObjClassRec
xmDropSiteManagerObjectClass	xmPushButtonWidgetClass
xmDropTransferClassRec	XmQmotif
xmDropTransferObjectClass	xmRowColumnClassRec
xmExtClassRec	xmRowColumnWidgetClass

Figure 6-28: `libXm` **Contents** *, **Global External Data Symbols** (continued)

xmSashClassRec	xmShellExtObjectClass
xmSashWidgetClass	xmTearOffButtonClassRec
xmScaleClassRec	xmTearOffButtonWidgetClass
xmScaleWidgetClass	xmTextClassRec
xmScreenClass	xmTextFieldClassRec
xmScreenClassRec	xmTextFieldWidgetClass
xmScrollBarClassRec	xmTextWidgetClass
xmScrollBarWidgetClass	xmToggleButtonClassRec
xmScrolledWindowClassRec	xmToggleButtonGadgetClass
xmScrolledWindowWidgetClass	xmToggleButtonGadgetClassRec
xmSelectionBoxClassRec	xmToggleButtonGCacheObjClassRec
xmSelectionBoxWidgetClass	xmToggleButtonWidgetClass
xmSeparatorClassRec	xmUseVersion
xmSeparatorGadgetClass	xmVendorShellExtClassRec
xmSeparatorGadgetClassRec	xmVendorShellExtObjectClass
xmSeparatorGCacheObjClassRec	xmWorldClass
xmSeparatorWidgetClass	xmWorldClassRec
xmShellExtClassRec	

* All functions in this table are new to the Fourth Edition of the ABI. M

The following functions reside in `libMrm` and must be provided on all ABI- M
conforming systems that implement a graphical windowing terminal interface. M

Figure 6-29: `libMrm` **Contents** *

MrmCloseHierarchy	MrmFetchWidgetOverride
MrmFetchBitmapLiteral	MrmInitialize
MrmFetchColorLiteral	MrmOpenHierarchy
MrmFetchIconLiteral	MrmOpenHierarchyPerDisplay
MrmFetchLiteral	MrmRegisterClass
MrmFetchSetValues	MrmRegisterNames
MrmFetchWidget	MrmRegisterNamesInHierarchy

* All functions in this table are new to the Fourth Edition of the ABI. M

System Data Interfaces

Standard header files that describe system data are available for C application developers to use. These files are referred to by their name in angle brackets: <*name*.h>, <sys/*name*.h> and <X11/*name*.h>. Included in these headers are M
macro definitions, data definitions, and function declarations. The parts of the header files specified in the ANSI C standard are the only parts available to strictly-conforming ANSI C applications. Similarly, only the portions of the header files defined in the POSIX P1003.1 Operating System Standard are available to strictly POSIX-conforming applications.

Some of the following header files define interfaces directly available to applications, and those interfaces will be common to systems on all processors. Other header files show specific implementations of standard interfaces, where the implementation or data definitions might change from one processor to another. The *ABI* does not distinguish between these files. It gives data definitions to promote binary application portability, not to repeat source interface definitions available elsewhere. System providers and application developers should use the ABI to supplement—not to replace—source interface definition documents.

NOTE Some type and data definitions appear in multiple headers. Special definitions are included in the headers to avoid conflicts.

The application execution environment presents the interfaces described below, but the *ABI* does not require the presence of the header files themselves. In other words, an ABI-conforming system is not required to provide an application development environment.

Required Sizes for Some Data Objects

The continued evolution of System V requires that some fundamental data objects be expanded so that the operating system will work more efficiently with systems of different sizes and with networks of systems. To promote both binary portability for applications and interoperability for networks of systems, the *System V ABI* requires that all conforming implementations use the expanded data object sizes shown in the table below at a minimum.

A given architecture may expand one or more of the following objects' sizes, but any such further expansion must be made explicit in the processor supplement to the *System V ABI* for that processor architecture. Further, the sizes in the table below should be considered as absolute (not minimal) for purposes of interoperation among networks of heterogeneous systems.

Figure 6-30: Minimum Sizes of Fundamental Data Objects

Data Object	Type Definition	Size (bits)
User identifier	`uid_t`	32
Group Identifier	`gid_t`	32
Process identifier	`pid_t`	32
Inode identifier	`ino_t`	32
Device identifier	`dev_t`	32
File system identifier	`dev_t`	32
Error identifier	`int`	32
Time	`time_t`	32
File mode indicator	`mode_t`	32
Link count	`nlink_t`	32

Data Definitions (Processor-Specific)

NOTE This section requires processor-specific information. The *ABI* supplement for the desired processor describes the details.

7 FORMATS AND PROTOCOLS

Introduction

This chapter describes file and data formats and protocols that are visible to applications or that must be portable across System V implementations. Although the system provides standard programs to manipulate files in the formats described here, portability is important enough to warrant descriptions of their formats.

This does not mean applications are encouraged to circumvent the system programs and manipulate the files directly. Instead, it means an ABI for any target processor architecture must provide the system programs and library routines for manipulating these file formats and implementing these protocols. Moreover, those programs must accept formats compatible with the ones described here.

Some system file formats explicitly do not appear in this chapter. For example, the mount table file, traditionally called `/etc/mnttab`, is not a documented format. For information such as this, the system libraries described in Chapter 6 provide functions that applications may use to access the data. Programs that depend on undocumented formats—or that depend on the existence of undocumented files—are not ABI-conforming. Other files, such as the password file in `/etc/passwd`, have formats defined in other documents. These cases therefore do not appear in the ABI, because the ABI does not replicate information available in other standards documents. Nonetheless, an ABI-conforming program may use the files when the file names and formats are defined implicitly for the ABI through references to other documents.

Archive File

Archives package multiple files into one. They are commonly used as libraries of relocatable object files to be searched by the link editor. An archive file has the following format:

- An archive magic string, `ARMAG` below;
- An optional archive symbol table (created only if at least one member is an object file that defines non-local symbols);
- An optional archive string table (created only if at least one archive member's name is more than 15 bytes long);
- For each ''normal'' file in the archive, an archive header and the unchanged contents of the file.

The archive file's magic string contains `SARMAG` bytes and does *not* include a terminating null byte.

Figure 7-1: `<ar.h>`

```
#define ARMAG      "!<arch>\n"
#define SARMAG     8
#define ARFMAG     "'\n"

struct ar_hdr {
        char      ar_name[16];
        char      ar_date[12];
        char      ar_uid[6];
        char      ar_gid[6];
        char      ar_mode[8];
        char      ar_size[10];
        char      ar_fmag[2];
};
```

All information in the member headers is printable ASCII.

`ar_name` This member represents the member's file name. If the name fits, it resides in the member directly, terminated with slash (`/`) and padded with blanks on the right. If the member's name is too long to fit, the member contains a slash, followed by the decimal representation of the name's offset in the archive string table.

ar_date	This member holds the decimal representation of the modification date of the file at the time of its insertion into the archive. Stat and time describe the modification time value. X
ar_uid	This member holds the decimal representation of the member's user identification number.
ar_gid	This member holds the decimal representation of the member's group identification number.
ar_mode	This member holds the octal representation of the file system mode.
ar_size	This member holds the decimal representation of the member's size in bytes.
ar_fmag	This member holds the first two bytes of the ARFMAG string, defined above.

Each member begins on an even byte boundary; a newline is inserted between files if necessary. Nevertheless the ar_size member reflects the actual size of the member exclusive of padding. There is no provision for empty areas in an archive file.

If some archive member's name is more than 15 bytes long, a special archive member contains a table of file names, each followed by a slash and a newline. This string table member, if present, will precede all ''normal'' archive members. The special archive symbol table is not a ''normal'' member, and must be first if it exists (see below). The ar_name entry of the string table's member header holds a zero length name (ar_name[0]=='/'), followed by one trailing slash (ar_name[1]=='/'), followed by blanks (ar_name[2]==' ', and so on). Offsets into the string table begin at zero. Example ar_name values for short and long file names appear below.

Offset	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9
0	f	i	l	e	n	a	m	e	s	a
10	m	p	l	e	/	\n	l	o	n	g
20	e	r	f	i	l	e	n	a	m	e
30	x	a	m	p	l	e	/	\n		

Figure 7-2: Example String Table

Member Name	`ar_name`	Note
`short-name`	`short-name/`	Not in string table
`filenamesample`	`/0`	Offset 0 in string table
`longerfilenamexample`	`/16`	Offset 16 in string table

If an archive file has one or more object file members, its first member will be an archive symbol table. This archive member has a zero length name, followed by blanks (`ar_name[0]=='/', ar_name[1]==' '`, and so on). All *words* in this symbol table have four bytes, using the machine-independent encoding shown below.

NOTE All machines use the encoding described here for the symbol table, even if the machine's ‘‘natural’’ byte order is different.

Figure 7-3: Archive Word Encoding

	0	1	2	3
`0x01020304`	`01`	`02`	`03`	`04`

The symbol table holds the following:

- A word containing the number of symbols in the symbol table (which is the same as the number of entries in the file offset array);
- An array of words, holding file offsets into the archive;
- The string table containing `ar_size-4*(`*number symbols*`+1)` bytes, whose initial byte is numbered 0 and whose last byte holds a 0 value.

Entries in the string table and in the file offset array exactly correspond to each other, and they both parallel the order of archive members. Thus if two or more archive members define symbols with the same name (which is allowed), their string table entries will appear in the same order as their corresponding members in the archive file. Each array entry associates a symbol with the archive member that defines the symbol; the file offset is the location of the archive header for the designated archive member.

As an example, the following symbol table defines 4 symbols. The archive member at file offset 114 defines `name` and `object`. The archive member at file offset 426 defines `function` and a second version of `name`.

Figure 7-4: Example Symbol Table

Offset	+0	+1	+2	+3	
0	4				4 offset entries
4	114				name
8	114				object
12	426				function
16	426				name
20	n	a	m	e	
24	\0	o	b	j	
28	e	c	t	\0	
32	f	u	n	c	
36	t	i	o	n	
40	\0	n	a	m	
44	e	\0			

Other Archive Formats

ABI-conforming systems support archives created by the `cpio` command. These archives are commonly used as a vehicle for collecting ASCII files for storage or transmission.

For more information about these archives see the `cpio` manual page in the X
X/Open CAE Specification, Issue 4.2 and the *System V Interface Definition, Fourth Edi-* X
tion (see the Conformance Rule in chapter 1). The format of the archives the com-
mand creates is included in the *IEEE POSIX P1003.1* specification. Also supported E
are archives in the ASC/CRC format, created using the `cpio` command with the E
''`-H crc`'' option.

Terminfo Data Base

Each terminal's capabilities are stored in a separate file named `/usr/share/lib/terminfo/`*L*/*terminal_name*, where *terminal_name* is the name of the terminal, and *L* is the first letter of the terminal's name. The specific capabilities described in the database and their names are given in the *System V Interface Definition, Fourth Edition* on the `terminfo`(TI_ENV) pages. The format of this file is given in the separate section that follows.

The `terminfo` database format is hardware-independent. An 8-bit byte is assumed. Short integers are stored in two contiguous 8-bit bytes. The first byte contains the least significant 8 bits of the value, and the second byte contains the most significant 8 bits. (Thus, the value represented is 256*second+first.) The value `-1` is represented by `0377, 0377`, and the value `-2` is represented by `0376, 0377`; other negative values are illegal. Computers where this does not correspond to the hardware read the integers as two bytes and compute the result, making the compiled entries portable across machine types. A `-1` or a `-2` in a capability field means that the capability is missing from a terminal.

The file contains six sections:

header
: The header contains six short integers in the format described below. These integers are (1) the magic number (octal `0432`); (2) the size, in bytes, of the names section; (3) the number of bytes in the boolean section; (4) the number of short integers in the numbers section; (5) the number of offsets (short integers) in the strings section; (6) the size, in bytes, of the string table.

terminal names
: The terminal names section contains the first line of the `terminfo`(TI_ENV) description, listing the various names for the terminal, separated by the '`|`' character. The section is terminated with an ASCII NUL character.

boolean flags
: The boolean flags contain one byte for each flag. This byte is either `0` or `1` to indicate whether a capability is present or absent. The terminal capabilities are stored here in the same order in which that are listed under the `Booleans` heading of the capability table in the `terminfo`(TI_ENV) section of the *System V Interface Definition, Fourth Edition*. The flag value of `2` means that the flag is invalid.

Between the boolean section and the number section, a null byte will be inserted, if necessary, to insure that the number section begins on a byte with an even-numbered address. All short integers are aligned on a short word boundary.

numbers — The numbers section is similar to the preceding boolean flags section. Each capability is stored in two bytes as a short integer. Terminal capabilities are stored here in the same order in which they are listed under the **Numbers** heading of the capability table in the `terminfo`(TI_ENV) section of the *System V Interface Definition, Fourth Edition*. If the value represented is `−1` or `−2`, the capability is missing.

strings — In the strings section, each capability is stored as a short integer, in the same format given for preceding section. Terminal capabilities are stored here in the same order in which they are listed under the **Strings** heading of the capability table in the `terminfo`(TI_ENV) section of the *System V Interface Definition, Fourth Edition*. A value of `−1` or `−2` means that a capability is missing. Otherwise, the value is taken as an offset from the beginning of the string table.

string table — The final section is the string table. It contains all the values of string capabilities referenced in the string section. Each capability is stored as a null terminated string. Special characters in `^X` or \ notation are stored in their interpreted form, not the printing representation. Padding information ($<nn>) and parameter information (%x) are stored intact in uninterpreted form.

As an example, an octal dump of the compiled description for an AT&T Model 37 KSR is included:

```
37|tty37|AT&T model 37 teletype,
   hc, os, xon,
   bel=^G, cr=\r, cub1=\b, cud1=\n, cuu1=\E7, hd=\E9,
   hu=\E8, ind=\n,
```

```
0000000 032 001      \0 032  \0 013  \0 021 001   3  \0   3   7   |   t
0000020   t   y   3   7   |   A   T   &   T       m   o   d   e   l    
0000040   3   7       t   e   l   e   t   y   p   e  \0  \0  \0  \0  \0
0000060  \0  \0  \0 001  \0  \0  \0  \0  \0  \0  \0 001  \0  \0  \0  \0
0000100 001  \0  \0  \0  \0  \0 377 377 377 377 377 377 377 377 377 377
0000120 377 377 377 377 377 377 377 377 377 377 377 377 377 377   &  \0
0000140      \0 377 377 377 377 377 377 377 377 377 377 377 377 377 377
0000160 377 377   "  \0 377 377 377 377   (  \0 377 377 377 377 377 377
0000200 377 377   0  \0 377 377 377 377 377 377 377 377   -  \0 377 377
0000220 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0000520 377 377 377 377 377 377 377 377 377 377 377 377 377 377   $  \0
0000540 377 377 377 377 377 377 377 377 377 377 377 377 377 377   *  \0
0000560 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377 377
*
0001160 377 377 377 377 377 377 377 377 377 377 377 377 377 377   3   7
0001200   |   t   t   y   3   7   |   A   T   &   T       m   o   d   e
0001220   l       3   7       t   e   l   e   t   y   p   e  \0  \r  \0
0001240  \n  \0  \n  \0 007  \0  \b  \0 033   8  \0 033   9  \0 033   7
0001260  \0  \0
0001261
```

Some limitations: total compiled entries cannot exceed 4096 bytes and all entries in the name field cannot exceed 128 bytes.

Formats and Protocols for Networking

This section describes a language for machine-independent data representation and two protocols used in remote procedure call service. All ABI-conforming systems that provide these services must implement these formats and protocols in the appropriate routines in the `libnsl` library. See Chapter 6 of the *System V ABI* for a list of the routines in this library.

XDR: External Data Representation

XDR is a method for the description and encoding of data. It is useful for transferring data between different computer architectures, where such characteristics as internal byte-ordering may vary.

XDR uses a language to describe data formats. This language allows one to describe intricate data formats in a concise manner.

The XDR language assumes that bytes (or octets) are portable, where a byte is defined to be 8 bits of data. A given hardware device should encode the bytes onto the various media in such a way that other hardware devices may decode the bytes without loss of meaning.

Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through n-1. The bytes are read or written to some byte stream such that byte m always precedes byte m+1. If the n bytes needed to contain the data are not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of 4.

The familiar graphic box notation has been used for illustration and comparison. In most illustrations, each box (delimited by a plus sign at the 4 corners and vertical bars and dashes) depicts a byte. Ellipses (. . .) between boxes show zero or more additional bytes where required.

A Block

```
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)>----------->|
```

XDR Data Types

Each of the sections that follow describes a data type defined in the XDR language, shows how it is declared in the language, and includes a graphic illustration of its encoding.

For each defined data type we show a general paradigm declaration. Note that angle brackets (< and >) denote variable length sequences of data and square brackets ([and]) denote fixed-length sequences of data. ''n'', ''m'' and ''r'' denote integers. For the full language specification and more formal definitions of terms such as ''identifier'' and ''declaration,'' refer to ''The XDR Language Specification'', given in the *System V Interface Definition, Fourth Edition*.

For some data types, more specific examples are included below.

Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648,2147483647]. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. Integers are declared as follows:

Integer

```
(MSB)                  (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0,4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. An unsigned integer is declared as follows:

Unsigned Integer

```
(MSB)                  (LSB)
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
+-------+-------+-------+-------+
<------------32 bits------------>
```

Enumeration

Enumerations have the same representation as signed integers. Enumerations are handy for describing subsets of the integers. Enumerated data is declared as follows:

```
enum { name-identifier = constant, . . . } identifier;
```

For example, the three colors red, yellow, and blue could be described by an enumerated type:

```
enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

It is an error to encode as an enum any other integer than those that have been given assignments in the enum declaration.

Boolean

Booleans are important enough and occur frequently enough to warrant their own explicit type in the language. Booleans are declared as follows:

```
bool identifier;
```

This is equivalent to:

```
enum { FALSE = 0, TRUE = 1 } identifier;
```

Floating-point

XDR defines the floating-point data type ‘‘float’’ (32 bits or 4 bytes). The encoding used is the IEEE standard for normalized single-precision floating-point numbers. The following three fields describe the single-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 8 bits are devoted to this field. The exponent is biased by 127.

F: The fractional part of the number’s mantissa, base 2. 23 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

Single-Precision Floating-Point

```
+-------+-------+-------+-------+
|byte 0 |byte 1 |byte 2 |byte 3 |
S|   E   |          F           |
+-------+-------+-------+-------+
1|<- 8 ->|<-------23 bits------>|
<------------32 bits------------>
```

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating- point number are 0 and 31. The beginning bit (and most significant bit) offsets of S, E, and F are 0, 1, and 9, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the ''NaN'' (not a number) is system dependent and should not be used externally.

Double-precision Floating-point

XDR defines the encoding for the double-precision floating- point data type ''double'' (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized double-precision floating-point numbers. XDR encodes the following three fields, which describe the double-precision floating-point number:

S: The sign of the number. Values 0 and 1 represent positive and negative, respectively. One bit.

E: The exponent of the number, base 2. 11 bits are devoted to this field. The exponent is biased by 1023.

F: The fractional part of the number's mantissa, base 2. 52 bits are devoted to this field.

Therefore, the floating-point number is described by:

```
(-1)**S * 2**(E-Bias) * 1.F
```

It is declared as follows:

Double-Precision Floating-Point

```
+------+------+------+------+------+------+------+------+
|byte 0|byte 1|byte 2|byte 3|byte 4|byte 5|byte 6|byte 7|
S|    E   |                  F                        |
+------+------+------+------+------+------+------+------+
1|<--11-->|<-----------------52 bits------------------->|
<-----------------------64 bits------------------------->
```

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a double-precision floating- point number are 0 and 63. The beginning bit (and most significant bit) offsets of S, E , and F are 0, 1, and 12, respectively. Note that these numbers refer to the mathematical positions of the bits, and NOT to their actual physical locations (which vary from medium to medium).

The IEEE specifications should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). According to IEEE specifications, the ''NaN'' (not a number) is system dependent and should not be used externally.

Fixed-length Opaque Data

At times, fixed-length uninterpreted data needs to be passed among machines. This data is called ''opaque'' and is declared as follows:

opaque identifier[n];

where the constant n is the (static) number of bytes necessary to contain the opaque data. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count of the opaque object a multiple of four.

Fixed-Length Opaque

```
0        1     ...
+--------+--------+...+--------+--------+...+--------+
| byte 0 | byte 1 |...|byte n-1|    0   |...|    0   |
+--------+--------+...+--------+--------+...+--------+
|<-----------n bytes---------->|<------r bytes------>|
|<-----------n+r (where (n+r) mod 4 = 0)------------>|
```

Variable-length Opaque Data

XDR also provides for variable-length (counted) opaque data, defined as a sequence of n (numbered 0 through n-1) arbitrary bytes to be the number n encoded as an unsigned integer (as described below), and followed by the n bytes of the sequence.

Byte m of the sequence always precedes byte m+1 of the sequence, and byte 0 of the sequence always follows the sequence's length (count). If n is not a multiple of four, the the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Variable-length opaque data is declared in the following way:

opaque identifier<m>;

or

opaque identifier<>;

The constant m denotes an upper bound of the number of bytes that the sequence may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that the maximum data transfer size is 8192 bytes, as follows:

opaque filedata<8192>;

This can be illustrated as follows:

Variable-Length Opaque

```
0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
|<----n+r (where (n+r) mod 4 = 0)---->|
```

It is an error to encode a length greater than the maximum described in the specification.

String

XDR defines a string of n (numbered 0 through n-1) ASCII bytes to be the number n encoded as an unsigned integer (as described above), and followed by the n bytes of the string. Byte m of the string always precedes byte m+1 of the string, and byte 0 of the string always follows the string's length. If n is not a multiple of four, then the n bytes are followed by enough (0 to 3) residual zero bytes, r, to make the total byte count a multiple of four. Counted byte strings are declared as follows:

string object<m>;

or

string object<>;

The constant m denotes an upper bound of the number of bytes that a string may contain. If m is not specified, as in the second declaration, it is assumed to be (2**32) - 1, the maximum length. The constant m would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 255 bytes, as follows:

string filename<255>;

Which can be illustrated as:

A String

```
0     1     2     3     4     5   ...
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|        length n       |byte0|byte1|...| n-1 |  0  |...|  0  |
+-----+-----+-----+-----+-----+-----+...+-----+-----+...+-----+
|<-------4 bytes------->|<------n bytes------>|<---r bytes--->|
|<----n+r (where (n+r) mod 4 = 0)---->|
```

It is an error to encode a length greater than the maximum described in the specification.

Fixed-length Array

Declarations for fixed-length arrays of homogeneous elements are in the following form:

type-name identifier[n];

Fixed-length arrays of elements numbered 0 through n-1 are encoded by individually encoding the elements of the array in their natural order, 0 through n-1. Each element's size is a multiple of four bytes. Though all elements are of the same type, the elements may have different sizes. For example, in a fixed-length array of strings, all elements are of ''type string,'' yet each element will vary in its length.

Fixed-Length Array

```
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|   element 0   |   element 1   |...|  element n-1  |
+---+---+---+---+---+---+---+---+...+---+---+---+---+
|<--------------------n elements------------------->|
```

Variable-length Array

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count n (an unsigned integer) followed by the encoding of each of the array's elements, starting with element 0 and progressing through element n- 1. The declaration for variable-length arrays follows this form:

type-name identifier<m>;

or

type-name identifier<>;

The constant m specifies the maximum acceptable element count of an array; if m is not specified, as in the second declaration, it is assumed to be (2**32) - 1.

Counted Array

```
0  1  2  3
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|     n     | element 0 | element 1 |...|element n-1|
+--+--+--+--+--+--+--+--+--+--+--+--+...+--+--+--+--+
|<-4 bytes->|<--------------n elements------------->|
```

It is an error to encode a value of n that is greater than the maximum described in the specification.

Structure

Structures are declared as follows:

```
struct {
        component-declaration-A;
        component-declaration-B;
        ...
} identifier;
```

The components of the structure are encoded in the order of their declaration in the structure. Each component's size is a multiple of four bytes, though the components may be different sizes.

Structure

```
+------------+------------+...
| component A | component B |...
+------------+------------+...
```

Discriminated Union

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of discriminant is either ''int,'' ''unsigned int,'' or an enumerated type, such as ''bool''. The component types are called ''arms'' of the union, and are preceded by the value of the discriminant which implies their encoding. Discriminated unions are declared as follows:

```
union switch (discriminant-declaration) {
        case discriminant-value-A:
        arm-declaration-A;
        case discriminant-value-B:
        arm-declaration-B;
        ...
        default: default-declaration;
} identifier;
```

Each ''case'' keyword is followed by a legal value of the discriminant. The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. The size of the implied arm is always a multiple of four bytes.

The discriminated union is encoded as its discriminant followed by the encoding of the implied arm.

Discriminated Union

```
0   1   2   3
+---+---+---+---+---+---+---+---+
|  discriminant |  implied arm  |
+---+---+---+---+---+---+---+---+
|<---4 bytes--->|
```

Void

An XDR void is a 0-byte quantity. Voids are useful for describing operations that take no data as input or no data as output. They are also useful in unions, where some arms may contain data and others do not. The declaration is simply as follows:

```
void;
```

Voids are illustrated as follows:

```
++
||
++
--><-- 0 bytes
```

Constant

The data declaration for a constant follows this form:

```
const name-identifier = n;
```

‘‘const’’ is used to define a symbolic name for a constant; it does not declare any data. The symbolic constant may be used anywhere a regular constant may be used. For example, the following defines a symbolic constant DOZEN, equal to 12.

```
const DOZEN = 12;
```

Typedef

‘‘"typedef’’ does not declare any data either, but serves to define new identifiers for declaring data. The syntax is:

```
typedef declaration;
```

The new type name is actually the variable name in the declaration part of the typedef. For example, the following defines a new type called ‘‘eggbox’’ using an existing type called ‘‘egg’’:

```
typedef egg eggbox[DOZEN];
```

Variables declared using the new type name have the same type as the new type name would have in the typedef, if it was considered a variable. For example, the following two declarations are equivalent in declaring the variable ‘‘fresheggs’’:

```
eggbox  fresheggs;
egg     fresheggs[DOZEN];
```

When a typedef involves a struct, enum, or union definition, there is another (preferred) syntax that may be used to define the same type. In general, a typedef of the following form:

typedef <<struct, union, or enum definition>> identifier;

may be converted to the alternative form by removing the ''typedef'' part and placing the identifier after the ''struct,'' ''union,'' or ''enum'' keyword, instead of at the end. For example, here are the two ways to define the type ''bool'':

```
typedef enum {    /* using typedef */
         FALSE = 0,
         TRUE = 1
         } bool;

enum bool {       /* preferred alternative */
         FALSE = 0,
         TRUE = 1
         };
```

The reason this syntax is preferred is one does not have to wait until the end of a declaration to figure out the name of the new type.

Optional-data

Optional-data is one kind of union that occurs so frequently that we give it a special syntax of its own for declaring it. It is declared as follows:

type-name *identifier;

This is equivalent to the following union:

```
union switch (bool opted) {
         case TRUE:
             type-name element;
         case FALSE:
             void;
} identifier;
```

It is also equivalent to the following variable-length array declaration, since the boolean ''opted'' can be interpreted as the length of the array:

type-name identifier<1>;

Optional-data is not so interesting in itself, but it is very useful for describing recursive data-structures such as linked-lists and trees. For example, the following defines a type ''stringlist'' that encodes lists of arbitrary length strings:

```
struct *stringlist {
        string item<>;
        stringlist next;
};
```

It could have been equivalently declared as the following union:

```
union stringlist switch (bool opted) {
    case TRUE:
        struct {
            string item<>;
            stringlist next;
        } element;
    case FALSE:
        void;
};
```

or as a variable-length array:

```
struct stringlist<1> {
        string item<>;
        stringlist next;
};
```

Both of these declarations obscure the intention of the stringlist type, so the optional-data declaration is preferred over both of them. The optional-data type also has a close correlation to how recursive data structures are represented in high-level languages such as Pascal or C by use of pointers. In fact, the syntax is the same as that of the C language for pointers. E

Messaging Catalogues E

An ABI conforming implementation will include the `gencat` utility to convert the E
source of XPG3 conforming source message catalogues into a form understand- E
able by the run-time system. Applications that attempt to provide message catalo- E
gues in any other format are not ABI conforming.

RPC: Remote Procedure Call Protocol

This section specifies the protocol to be used in the remote procedure call package implemented by routines in the `libnsl` library described in Chapter 6. The protocol is specified with the external data representation (XDR) language partly described in the preceding section of the *System V ABI*.

Terminology

This section discusses servers, services, programs, procedures, clients, and versions. A server is a piece of software where network services are implemented. A network service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters, and results are documented in the specific program's protocol specification (see the *rpcbind protocol*, below, for an example). Network clients are pieces of software that initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol deals only with specification and interpretation of messages.

It is important to point out that RPC does not try to implement any kind of reliability and that the application must be aware of the type of transport protocol underneath RPC. If it knows it is running on top of a reliable transport such as TCP/IP, then most of the work is already done for it. On the other hand, if it is running on top of an unreliable transport such as UDP/IP, it must implement is own retransmission and time-out policy as the RPC layer does not provide this service.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, consider RPC running on top of an unreliable transport such as UDP/IP.

If an application retransmits RPC messages after short time-outs, the only thing it can infer if it receives no reply is that the procedure was executed zero or more times. If it does receive a reply, then it can infer that the procedure was executed at least once.

A server may wish to remember previously granted requests from a client and not regrant them in order to insure some degree of execute-at-most-once semantics. A server can do this by taking advantage of the transaction ID that is packaged with every RPC request. The main use of this transaction is by the client RPC layer in matching replies to requests. However, a client application may choose to reuse its previous transaction ID when retransmitting a request. The server application, knowing this fact, may choose to remember this ID after granting a request and not regrant requests with the same ID in order to achieve some degree of execute-at-most-once semantics. The server is not allowed to examine this ID in any other way except as a test for equality.

On the other hand, if using a reliable transport such as TCP/IP, the application can infer from a reply message that the procedure was executed exactly once, but if it receives no reply message, it cannot assume the remote procedure was not executed. Note that even if a connection-oriented protocol like TCP is used, an application still needs time-outs and reconnection to handle server crashes.

There are other possibilities for transports besides datagram- or connection-oriented protocols. For example, a request-reply protocol such as VMTP is perhaps the most natural transport for RPC.

Binding and Rendezvous Independence

The act of binding a client to a service is NOT part of the remote procedure call specification. This important and necessary function is left up to some higher-level software. (The software may use RPC itself—see the *rpcbind protocol*, below).

Implementors should think of the RPC protocol as the jump-subroutine instruction ("JSR") of a network; the loader (binder) makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, using RPC to accomplish this task.

Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice-versa. Security and access control mechanisms can be built on top of the message authentication. Several different authentication protocols can be supported. A field in the RPC header indicates which protocol is being used. More information on specific authentication protocols can be found in the Authentication Protocols, below.

Programs and Procedures

The RPC call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority. Once an implementor has a program number, he can implement his remote program; the first implementation would most likely have the version number of 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is ''read'' and procedure number 12 is ''write''.

Just as remote program protocols may change over several versions, the actual
RPC message protocol could also change. Therefore, the call message also has in it E
the RPC version number, which is equal to two for the version of RPC described E
here.

The reply message to a request message has enough information to distinguish the following error conditions:

1. The remote implementation of RPC does not speak the requested protocol E
   version number. The lowest and highest supported RPC version numbers E
   are returned.

2. The remote program is not available on the remote system.

3. The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.

4. The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)

5. The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is usually caused by a disagreement about the protocol between client and service.)

The remote implementation of RPC supports protocol version 2. E

Authentication

Provisions for authentication of caller to service and vice-versa are provided as a part of the RPC protocol. The call message has two authentication fields, the credentials and verifier. The reply message has one authentication field, the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
        AUTH_NULL  = 0,
        AUTH_SYS   = 1,
        AUTH_DES   = 3
        /*  and more to be defined  */
};

struct opaque_auth {
        auth_flavor flavor;
        opaque body<400>;
};
```

Any `opaque_auth` structure is an `auth_flavor` enumeration followed by bytes which are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. (See *Authentication Protocols*, below, for definitions of the various authentication protocols.)

If authentication parameters were rejected, the response message contains information stating why they were rejected.

Program Number Assignment

Program numbers are given out in groups of `0x20000000` (decimal 536870912) according to the following chart:

Program Numbers	*Description*
`0 - 1fffffff`	*Defined by a central authority*
`20000000 - 3fffffff`	*Defined by user*
`40000000 - 5fffffff`	*Transient*
`60000000 - 7fffffff`	*Reserved*
`80000000 - 9fffffff`	*Reserved*
`a0000000 - bfffffff`	*Reserved*
`c0000000 - dfffffff`	*Reserved*
`e0000000 - ffffffff`	*Reserved*

The first group is a range of numbers administered by a central authority and should be identical for all sites. The second range is for applications peculiar to a particular site. This range is intended primarily for debugging new programs. When a site develops an application that might be of general interest, that application should be given an assigned number in the first range. The third group is for applications that generate program numbers dynamically. The final groups are reserved for future use, and should not be used.

Other Uses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message-passing protocol with which other (non-RPC) protocols can be implemented. Here are two examples:

Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching typically uses reliable byte stream protocols (like TCP/IP) for its transport. In the case of batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgement).

Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPC uses unreliable, packet-based protocols (like UDP/IP) as its transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors. Broadcast RPC uses the `rpcbind` service to achieve its semantics. See the *rpcbind protocol*, below, for more information.

DRAFT COPY
March 18, 1997
File: chap7
386:adm.book:sum

The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top-down style.

```
enum msg_type {
    CALL  = 0,
    REPLY = 1
};

/*
* A reply to a call message can take on two forms:
* The message was either accepted or rejected.
*/
enum reply_stat {
    MSG_ACCEPTED = 0,
    MSG_DENIED   = 1
};

/*
* Given that a call message was accepted, the following is the
* status of an attempt to call a remote procedure.
*/
enum accept_stat {
    SUCCESS       = 0, /* RPC executed successfully      */
    PROG_UNAVAIL  = 1, /* remote hasn't exported program  */
    PROG_MISMATCH = 2, /* remote can't support version # */
    PROC_UNAVAIL  = 3, /* program can't support procedure */
    GARBAGE_ARGS  = 4  /* procedure can't decode params   */
};
/*
* Reasons why a call message was rejected:
*/
enum reject_stat {
    RPC_MISMATCH = 0, /* RPC version number != 2*/
    AUTH_ERROR = 1    /* remote can't authenticate caller */
};

/*
* Why authentication failed:
*/
enum auth_stat {
    AUTH_BADCRED      = 1,  /* bad credentials */
    AUTH_REJECTEDCRED = 2,  /* client must begin new session */
    AUTH_BADVERF      = 3,  /* bad verifier */
    AUTH_REJECTEDVERF = 4,  /* verifier expired or replayed  */
    AUTH_TOOWEAK      = 5   /* rejected for security reasons */
};
```

(continued on next page)

```
/*
* The RPC message:
* All messages start with a transaction identifier, xid,
* followed by a two-armed discriminated union.  The union's
* discriminant is a msg_type which switches to one of the two
* types of the message.  The xid of a REPLY message always
* matches that of the initiating CALL message.  NB: The xid
* field is only used for clients matching reply messages with
* call messages or for servers detecting retransmissions; the
* service side cannot treat this id as any type of sequence
* number.
*/
struct rpc_msg {
    unsigned int xid;
    union switch (msg_type mtype) {
        case CALL:
            call_body cbody;
        case REPLY:
            reply_body rbody;
    } body;
};

/*
* Body of an RPC request call:
* rpcvers may
* be equal to 2 or 3.  The fields prog, vers, and proc specify the
* remote program, its version number, and the procedure within
* the remote program to be called.  After these fields are two
* authentication parameters: cred (authentication credentials)
* and verf (authentication verifier).  The two authentication
* parameters are followed by the parameters to the remote
* procedure, which are specified by the specific program
* protocol.
*/
struct call_body {
    unsigned int rpcvers;  /* must be equal to two (2) */
    unsigned int prog;
    unsigned int vers;
    unsigned int proc;
    opaque_auth cred;
    opaque_auth verf;
    /* procedure specific parameters start here */
};
```

(continued on next page)

E

```
/*
* Body of a reply to an RPC request:
* The call message was either accepted or rejected.
*/
union reply_body switch (reply_stat stat) {
    case MSG_ACCEPTED:
        accepted_reply areply;
    case MSG_DENIED:
        rejected_reply rreply;
} reply;

/*
* Reply to an RPC request that was accepted by the server:
* there could be an error even though the request was accepted.
* The first field is an authentication verifier that the server
* generates in order to validate itself to the caller.  It is
* followed by a union whose discriminant is an enum
* accept_stat.  The SUCCESS arm of the union is protocol
* specific.  The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGP
* arms of the union are void.  The PROG_MISMATCH arm specifies
* the lowest and highest version numbers of the remote program
* supported by the server.
*/
struct accepted_reply {
    opaque_auth verf;
    union switch (accept_stat stat) {
        case SUCCESS:
            opaque results[0];
            /*  procedure-specific results start here  */
        case PROG_MISMATCH:
            struct {
                unsigned int low;
                unsigned int high;
            } mismatch_info;
        default:
      /*
      * Void.  Cases include PROG_UNAVAIL, PROC_UNAVAIL,
      * and GARBAGE_ARGS.
      */
            void;
    } reply_data;
};
```

(continued on next page)

```
/*
* Reply to an RPC request that was rejected by the server:
* The request can be rejected for two reasons: either the
* server is not running a compatible version of the RPC
* protocol (RPC_MISMATCH), or the server refuses to
* authenticate the caller (AUTH_ERROR).  In case of an RPC
* version mismatch, the server returns the lowest and highest
* supported RPC version numbers.  In case of refused
* authentication, failure status is returned.
*/
union rejected_reply switch (reject_stat stat) {
    case RPC_MISMATCH:
        struct {
            unsigned int low;
            unsigned int high;
        } mismatch_info;
    case AUTH_ERROR:
        auth_stat stat;
};
```

Authentication Protocols

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some ''flavors'' of authentication which are already implemented. Other sites are free to invent new authentication types, with the same rules of flavor number assignment as there is for program number assignment.

Null Authentication

Often calls must be made where the caller does not know who he is or the server does not care who the caller is. In this case, the flavor value (the discriminant of the `opaque_auth`'s union) of the RPC message's credentials, verifier, and response verifier is `AUTH_NULL`. The bytes of the opaque_auth's body are undefined. It is recommended that the opaque length be zero.

Basic Authentication for UNIX Systems

The callers of a remote procedure may wish to identify themselves as they are identified on a UNIX system. The value of the credential's discriminant of an RPC call message is `AUTH_SYS`. The bytes of the credential's opaque body encode the following structure:

```
struct authsys_parms {
    u_long aup_time;
    char  *aup_machname;
    uid_t  aup_uid;
    gid_t  aup_gid;
    u_int  aup_len;
    gid_t *aup_gids;
};
```

The aup_machname is the name of the caller's machine (like ''krypton''). The aup_uid is the caller's effective user ID. The aup_gid is the caller's effective group ID. The aup_gids is a counted array of groups which contain the caller as a member. The verifier accompanying the credentials should be of AUTH_NULL (defined above).

The value of the discriminant of the response verifier received in the reply message from the server may be AUTH_NULL.

DES Authentication

Basic authentication for UNIX systems suffers from two major problems:

1. The naming is oriented for UNIX systems.
2. There is no verifier, so credentials can easily be faked.

DES authentication attempts to fix these two problems.

Naming

The first problem is handled by addressing the caller by a simple string of characters instead of by an operating system specific integer. This string of characters is known as the ''netname'' or network name of the caller. The server is not allowed to interpret the contents of the caller's name in any other way except to identify the caller. Thus, netnames should be unique for every caller in the naming domain.

It is up to each operating system's implementation of DES authentication to generate netnames for its users that insure this uniqueness when they call upon remote servers. Operating systems already know how to distinguish users local to their systems. It is usually a simple matter to extend this mechanism to the network. For example, a user with a user ID of 515 might be assigned the following netname: ''unix.515@sun.com''. This netname contains three items that serve to insure it is unique. Going backwards, there is only one naming domain called ''sun.com'' in the internet. Within this domain, there is only one user with user ID 515. However, there may be another user on another operating system within the same naming domain that, by coincidence, happens to have the same user ID. To insure that these two users can be distinguished we add the operating system name. So one user is ''unix.515@sun.com'' and the other could be ''vms.515@sun.com''.

The first field is actually a naming method rather than an operating system name. It just happens that today there is almost a one-to-one correspondence between naming methods and operating systems. If the world could agree on a naming standard, the first field could be the name of that standard, instead of an operating system name.

DES Authentication Verifiers

Unlike authentication for UNIX systems, DES authentication does have a verifier so the server can validate the client's credential (and vice-versa). The contents of this verifier is primarily an encrypted timestamp. The server can decrypt this timestamp, and if it is close to what the real time is, then the client must have encrypted it correctly. The only way the client could encrypt it correctly is to know the ''conversation key'' of the RPC session. And if the client knows the conversation key, then it must be the real client.

The conversation key is a DES key which the client generates and notifies the server of in its first RPC call. The conversation key is encrypted using a public key scheme in this first transaction. The particular public key scheme used in DES authentication is Diffie-Hellman with 192-bit keys. The details of this encryption method are described later.

The client and the server need the same notion of the current time in order for all of this to work. If network time synchronization cannot be guaranteed, then client can synchronize with the server before beginning the conversation.

The way a server determines if a client timestamp is valid is somewhat complicated. For any other transaction but the first, the server just checks for two things:

1. The timestamp is greater than the one previously seen from the same client.
2. The timestamp has not expired.

A timestamp is expired if the server's time is later than the sum of the client's timestamp plus what is known as the client's "window". The ''window'' is a number the client passes (encrypted) to the server in its first transaction. It can be thought of as a lifetime for the credential.

This explains everything but the first transaction. In the first transaction, the server checks only that the timestamp has not expired. If this was all that was done though, then it would be quite easy for the client to send random data in place of the timestamp with a fairly good chance of succeeding. As an added check, the client sends an encrypted item in the first transaction known as the ''window verifier'' which must be equal to the window minus 1, or the server will reject the credential.

DRAFT COPY
March 18, 1997
File: chap7
386:adm.book:sum

The client too must check the verifier returned from the server to be sure it is legitimate. The server sends back to the client the encrypted timestamp it received from the client, minus one second. If the client gets anything different than this, it will reject it.

Nicknames and Clock Synchronization

After the first transaction, the server's DES authentication subsystem returns in its verifier to the client an integer ''nickname'' which the client may use in its further transactions instead of passing its netname, encrypted DES key and window every time. The nickname is most likely an index into a table on the server which stores for each client its netname, decrypted DES key and window.

Though they originally were synchronized, the client's and server's clocks can get out of sync again. When this happens the client RPC subsystem most likely will get back `RPC_AUTHERROR` at which point it should resynchronize.

A client may still get the `RPC_AUTHERROR` error even though it is synchronized with the server. The reason is that the server's nickname table is a limited size, and it may flush entries whenever it wants. A client should resend its original credential in this case and the server will give it a new nickname. If a server crashes, the entire nickname table gets flushed, and all clients will have to resend their original credentials.

DES Authentication Protocol (in XDR language)

```
/*
* There are two kinds of credentials: one in which the client uses
* its full network name, and one in which it uses its "nickname"
* (just an unsigned integer) given to it by the server.  The
* client must use its fullname in its first transaction with the
* server, in which the server will return to the client its
* nickname.  The client may use its nickname in all further
* transactions with the server.  There is no requirement to use the
* nickname, but it is wise to use it for performance reasons.
*/
enum authdes_namekind {
    ADN_FULLNAME = 0,
    ADN_NICKNAME = 1
};

/*
* A 64-bit block of encrypted DES data
*/
typedef opaque des_block[8];
```

(continued on next page)

```
/*
* Maximum length of a network user's name
*/
const MAXNETNAMELEN = 255;

/*
* A fullname contains the network name of the client, an encrypted
* conversation key and the window.  The window is actually a
* lifetime for the credential.  If the time indicated in the
* verifier timestamp plus the window has past, then the server
* should expire the request and not grant it.  To insure that
* requests are not replayed, the server should insist that
* timestamps are greater than the previous one seen, unless it is
* the first transaction.  In the first transaction, the server
* checks instead that the window verifier is one less than the
* window.
*/
struct authdes_fullname {
    string name<MAXNETNAMELEN>;  /* name of client */
    des_block key;               /* PK encrypted conversation key */
    unsigned int window;         /* encrypted window */
};

/*
* A credential is either a fullname or a nickname
*/
union authdes_cred switch (authdes_namekind adc_namekind) {
    case ADN_FULLNAME:
        authdes_fullname adc_fullname;
    case ADN_NICKNAME:
        unsigned int adc_nickname;
};

/*
* A timestamp encodes the time since midnight, January 1, 1970.
*/
struct timestamp {
    unsigned int seconds;    /* seconds */
    unsigned int useconds;   /* and microseconds */
};

/*
* Verifier: client variety
* The window verifier is only used in the first transaction.  In
* conjunction with a fullname credential, these items are packed
* into the following structure before being encrypted:
*
* struct {
*   adv_timestamp;        -- one DES block
*   adv_fullname.window;     -- one half DES block
```

(continued on next page)

```
*  adv_winverf;         -- one half DES block
* }
* This structure is encrypted using CBC mode encryption with an
* input vector of zero.  All other encryptions of timestamps use
* ECB mode encryption.
*/
struct authdes_verf_clnt {
    timestamp adv_timestamp;     /*  encrypted timestamp      */
    unsigned int adv_winverf;    /*  encrypted window verifier */
};

/*
* Verifier: server variety
* The server returns (encrypted) the same timestamp the client
* gave it minus one second.  It also tells the client its nickname
* to be used in future transactions (unencrypted).
*/
struct authdes_verf_svr {
    timestamp adv_timeverf;      /*  encrypted verifier     */
    unsigned int adv_nickname;   /*  new nickname for client */
};
```

Diffie-Hellman Encryption

In this scheme, there are two constants, `BASE` and `MODULUS`. The particular values chosen for these for the DES authentication protocol are:

```
const BASE = 3;
const MODULUS =
   "d4a0ba0250b6fd2ec626e7efd637df76c716e22d0944b88b";
```

The way this scheme works is best explained by an example. Suppose there are two persons ''A'' and ''B'' who want to send encrypted messages to each other. So, A and B both generate ''secret'' keys at random which they do not reveal to anyone. Let these keys be represented as SK(A) and SK(B). They also publish in a public directory their ''public'' keys. These keys are computed as follows:

```
PK(A) = ( BASE ** SK(A) ) mod MODULUS
PK(B) = ( BASE ** SK(B) ) mod MODULUS
```

The ''**'' notation is used here to represent exponentiation. Now, both A and B can arrive at the ''common'' key between them, represented here as CK(A, B), without revealing their secret keys.

A computes:

```
CK(A, B) = ( PK(B) ** SK(A)) mod MODULUS
```

while B computes:

CK(A, B) = (PK(A) ** SK(B)) mod MODULUS

These two can be shown to be equivalent:

(PK(B) ** SK(A)) mod MODULUS = (PK(A) ** SK(B)) mod MODULUS

We drop the ''mod MODULUS'' parts and assume modulo arithmetic to simplify things:

PK(B) ** SK(A) = PK(A) ** SK(B)

Then, replace PK(B) by what B computed earlier and likewise for PK(A).

((BASE ** SK(B)) ** SK(A) = (BASE ** SK(A)) ** SK(B)

which leads to:

BASE ** (SK(A) * SK(B)) = BASE ** (SK(A) * SK(B))

This common key CK(A, B) is not used to encrypt the timestamps used in the protocol. Rather, it is used only to encrypt a conversation key which is then used to encrypt the timestamps. The reason for doing this is to use the common key as little as possible, for fear that it could be broken. Breaking the conversation key is a far less serious offense, since conversations are relatively short-lived.

The conversation key is encrypted using 56-bit DES keys, yet the common key is 192 bits. To reduce the number of bits, 56 bits are selected from the common key as follows. The middle-most 8-bytes are selected from the common key, and then parity is added to the lower order bit of each byte, producing a 56-bit key with 8 bits of parity.

Record Marking Standard

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a four-byte header followed by 0 to (2**31) - 1 bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values—a boolean which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the highest-order bit of the header; the length is the 31 low-order bits. (Note that this record specification is NOT in XDR standard form!)

rpcbind Mechanism

An rpcbind mechanism maps RPC program and version numbers to universal addresses, thus making dynamic binding of remote programs possible. This mechanism should run at a well-known address, and other programs register their dynamically allocated network addresses with it. It then makes those addresses publically available. Universal addresses are defined by the addressing authority of the given transport. They are NULL-terminated strings.

rpcbind also aids in broadcast RPC. There is no fixed relationship between the addresses which a given RPC program will have on different machines, so there's no way to directly broadcast to all of these programs. The rpcbind mechanism, however, has a well-known address. So, to broadcast to a given program, the client actually sends its message to the rpcbind process on the machine it wishes to reach. rpcbind picks up the broadcast and calls the local service specified by the client. When rpcbind gets a reply from the local service, it passes it on to the client.

rpcbind Protocol Specification (in RPC Language)

```
/*
 * rpcbind procedures
 */
program RPCBPROG {
    version RPCBVERS {
        void
        RPCBPROC_NULL(void) = 0;

        bool
        RPCBPROC_SET(rpcb) = 1;

        bool
        RPCBPROC_UNSET(rpcb) = 2;

        string
        RPCBPROC_GETADDR(rpcb) = 3;

        rpcblist
        RPCBPROC_DUMP(void) = 4;

        rpcb_rmtcallres
        RPCBPROC_CALLIT(rpcb_rmtcallargs) = 5;

        unsigned int
        RPCBPROC_GETTIME(void) = 6;
    } = 3;
} = 100000;
```

rpcbind Operation

The rpcbind mechanism is contacted by way of an assigned address specific to the transport which is used. In case of IP, for example, it is port number 111. Each transport has such an assigned well known address. The following is a description of each of the procedures supported by rpcbind:

RPCBPROC_NULL:
: This procedure does no work. By convention, procedure zero of any protocol takes no parameters and returns no results.

RPCBPROC_SET:
: When a program first becomes available on a machine, it registers itself with the rpcbind program running on the same machine. The program passes its program number, version number, network identifier and the universal address on which it awaits service requests. The procedure returns a boolean response whose value is TRUE if the procedure successfully established the mapping and FALSE otherwise. The procedure refuses to establish a mapping if one already exists for the tuple. Note that neither network identifier nor universal address can be NULL, and that network identifier should be valid on the machine making the call.

RPCBPROC_UNSET:
: When a program becomes unavailable, it should unregister itself with the rpcbind program on the same machine. The parameters and results have meanings identical to those of RPCBPROC_SET. The mapping of the program number, version number and network identifier tuple with universal address is deleted. If network identifier is NULL, all mappings specified by the tuple *(program number, version number, *)* and the corresponding universal addresses are deleted.

RPCBPROC_GETADDR:
: Given a program number, version number and network identifier, this procedure returns the universal address on which the program is awaiting call requests.

RPCBPROC_DUMP:
: This procedure enumerates all entries in rpcbind's database. The procedure takes no parameters and returns a list of program, version, netid, and universal addresses.

RPCBPROC_CALLIT:
: This procedure allows a caller to call another remote procedure on the same machine without knowing the remote procedure's universal address. It is intended for supporting broadcasts to arbitrary remote programs via rpcbind's well-known address.

The parameters and the argument pointer are the program number, version number, procedure number, and parameters of the remote procedure. Note:

1. This procedure only sends a response if the procedure was successfully executed and is silent (no response) otherwise.
2. rpcbind can communicates with remote programs only by using connectionless transports.

The procedure returns the remote program's universal address, and the results of the remote procedure.

RPCBPROC_GETTIME:
This procedure returns the local time on its own machine.

rpcbind can also supports version 2 of the rpcbind (portmapper) program protocol; version 3 should be used. E

8 SYSTEM COMMANDS

Commands for Application Programs

Programs running on ABI-conforming system are capable of creating new processes and executing programs provided by the system. They can also execute a shell in a new process, and then use that shell to interpret a script that causes many system programs to be executed.

The system commands listed below must be available to applications executing on an ABI-conforming system. They include commands from the *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition*, Basic and Advanced Utilities Extensions (see Conformance Rule in chapter 1). X

The following commands are available to application programs running on ABI-conforming systems. All the commands must be accessible through the default PATH environment variable provided by the system (see, section 2.5.3 of the Commands and Utilities volume of the *X/Open CAE Specification, Issue 4.2* or the `envvar`(BA_ENV) manual page in the *System V Interface Definition, Fourth Edition*, for more information on execution environment variables). X

Figure 8-1: Commands required in an ABI Run-time Environment

cat	false	pg#	test
cd	find	pr	touch
chgrp	fmtmsg#	priocntl	tr
chmod	gettxt	pwd	true
chown	grep	rm	tty
cmp	id	rmdir	umask
cp	kill	sed	uname
cpio#	line#	sh	uucp
date	ln	sleep	uulog
dd	logname	sort	uustat
df	lp	stty	uux
echo	ls	su	vi
ed	mkdir	tail	wait
ex	mv	tee	who
expr	passwd	compress	uncompress
make	ar	basename	dirname
gencat	sum#	wc	

Command is DEPRECATED X

The following new commands, now required by the *X/Open CAE Specification, Issue 4.2*, are available to application programs running on ABI-conforming systems. All the commands must be accessible through the default PATH environment variable provided by the system. X

Figure 8-2: XPG4.2 Commands required in an ABI Run-time Environment X

alias	egrep	man	split	X
asa	env	mesg	strings	X
at	expand	mkfifo	tabs	X
awk	fc	more	talk	X
batch	fg	newgrp	tar#	X
bc	fgrep	nice	time	X
bg	file	nl	tput	X
c89	fold	nohup	tsort	X
cal	getconf	od	type	X
calendar#	getopts	pack#	ulimit	X
cksum	hash	paste	unalias	X
col#	head	patch	unexpand	X
comm	iconv	pathchk	uniq	X
command	jobs	pax	unpack#	X
crontab	join	pcat#	uudecode	X
csplit	locale	printf	uuencode	X
cut	localedef	ps	write	X
diff	logger	read	xargs	X
dircmp#	mailx	renice	zcat	X
du	mail#	spell#		X

Command is DEPRECATED

NOTE XSH4.2 conforming commands are accessed by setting the PATH variable equal to the value of the _CS_PATH variable as defined by XSH4.2 If an implementation offers full *System V Application Binary Interface, Fourth Edition* compatibility, the default command set including the shell in the system will be *System V Application Binary Interface, Fourth Edition* compatible. Applications that have strict compatibility requirements for their command usage, should consider only relying on the default command set. X

The following commands are available to application programs running on ABI-conforming systems that implement a graphical windowing terminal interface and therefore provide the Motif interface. M

Figure 8-3: Commands required in an ABI Run-time Environment for Motif* M

uil	mwm

M

*Function is new to the Fourth Edition of the ABI.

9 EXECUTION ENVIRONMENT

Application Environment

This section specifies the execution environment information available to application programs running on *ABI*-conforming computers. It also specifies the programs' interface to that information.

The execution environment contains certain information that is provided by the operating system and is available to executing application programs. Generally speaking, this includes system-wide environment information and per-process information that is meaningful and accessible only to the single process to which it applies. This environment information and the utilities used to retrieve it are specified in detail in the *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition* (see the Conformance Rule in chapter 1).

The environment information available to application programs on an *ABI*-conforming system includes the following:

- System identification

  Application programs may obtain system identification information through the `uname` function or the system command `uname`.

- Date and time

  The current calendar date and time are available to application programs through the `date` system command and the `time` function.

- Numerical Limits

  This refers to the maximum and minimum values of operating system variables and C language limits that application programs require. Most important values are accessible through the `sysconf` and `pathconf` functions defined in the *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition* and in the *POSIX P1003.1 Portable Operating System Specification* (see the Conformance Rule in chapter 1) . These interfaces are the correct method for application programs to retrieve numerical values related to the configuration of their host system. Guaranteed minimum values for these parameters are specified in the ''Data Definition'' section in Chapter 6 of the *ABI*. Other system parameters are accessible through the `getrlimit` function.

- Per-process environment information

  When an application program first begins execution an environment is made available to it. The *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition* (see the Conformance Rule in Chapter 1) pages for `envvar`, `exec`, and `system` contain detailed descriptions of this information. X

The *X/Open CAE Specification, Issue 4.2* and the *System V Interface Definition, Fourth Edition* (see the Conformance Rule in Chapter 1) are the definitive reference for information about the execution environment of processes; all of this information applies to *ABI*-conforming systems. The specific references given above will lead an interested reader to all appropriate information, but are not exhaustive in themselves. X

File System Structure and Contents

The file system on an *ABI*-conforming system is a tree-like structure. All ABI-conforming systems have a ‘‘root’’ (or `/`) directory that contains a `/usr` directory, a `/var` directory, a `/etc` directory, and a `/opt` directory.

The following are primary characteristics of the file system tree:

`/`	This directory contains machine-specific files and executable files required to boot the system, to check, and to repair file systems. It also contains other directories. No application may install files in the `/` directory.	
`/etc`	This directory subtree contains machine-specific configuration files and some executable files, including one command used during application package installation. Application programs should never execute applications in this directory subtree, and should never access directly any data files in this subtree, except for the files they install themselves in the `/etc/opt` directory.	
`/opt`	This subtree contains add-on software application packages.	E
`/tmp`	This directory contains temporary files created by system utilities.	E
`/usr`	This subtree contains only sharable files and executable files provided by the system.	
`/var`	This subtree contains files that change.	

Applications should install or create files only in designated places within the tree, as noted below.

This section describes those aspects of the root, user and var file systems that application programs can rely on being present.

Root subtree

The `/` directory contains executable programs and other files necessary to boot the system, check and repair file systems, and files describing the identity of a particular machine. The following components must be present in the root file system:

/	The root directory of the file system.
/dev	The top directory of the devices subtree containing block and character device files. All terminal and terminal-related device files must be in the /dev directory or in subdirectories of /dev. The following files are required to be present in the /dev directory.

Figure 9-1: Required Devices in an ABI Run-time Environment

/dev/tty	/dev/null	/dev/lpX

Where X may be any integer value. Note that further devices specifically required for networking support are defined in Chapter 12. The following sub-directories of /dev are required to be present.

- /dev/rmt
- /dev/mt

The names of other files present in /dev and naming conventions for sub-directories of /dev are processor-specific.

/etc	The top directory of the /etc subtree.
/opt	The top directory of the /opt subtree.
/usr	The top directory of the /usr subtree.
/var	The top directory of the /var subtree.

The /dev, /usr, and /tmp directories are for the use of the system. Applications should never create files in any of these directories, though they contain subdirectories that may be used by applications, as described below.

The /etc subtree

The directory /etc of the / file system is the point of access to the /etc subtree. This directory contains machine-specific configuration files.

The following describes the structure of the /etc subtree:

`/etc`	The top directory of the `/etc` subtree.	E
`/etc/opt/`*pkg*	This directory contains machine-specific configuration files provided by application packages.	E

The /opt subtree

The directory `/opt` of the `/` file system is the point of access to the `/opt` subtree. This directory subtree contains files installed by add-on application packages.

The following describes the structure of the `/opt` subtree:

`/opt`	The top directory of the `/opt` subtree.	E
`/opt/`*pkg*`/bin`	Executable files provided by application packages and invoked directly by users.	E
`/opt/`*pkg*	Where *pkg* is the abbreviated name of an add-on software package, contains all the static files installed on the system as part of that package.	

The /usr subtree

The directory `/usr` of the `/` file system is the point of access to the `/usr` subtree.

The following describes the structure of the `/usr` subtree:

`/usr`	The top directory in the /usr subtree.	
`/usr/bin`	Utility programs and commands for the use of all applications and users.	
`/usr/share`	The architecture-independent sharable files.	
`/usr/share/lib`	Architecture-independent databases.	M
`/usr/lib/X11`	The directory where X11 Window System libraries reside.	M
`/usr/lib/X11/locale`	The directory where X11 Window System localization packages are installed.	M M M
`/usr/lib/X11/app-defaults`	The directory where X11 Window System default application resource files are installed.	M M

Application programs may execute commands in the `/usr/bin` directory.

The /var subtree

The directory `/var` of the `/` file system is the point of access to the `/var` subtree. The `/var` subtree contains all files that vary in size or presence during normal system operations, including logging, accounting and temporary files created by the system and applications.

The following components of the `/var` subtree are important for application programs:

`/var/opt/`*pkg*	The top level directory used by application packages.	E
`/var/tmp`	Directory for temporary files created by applications.	

Applications should always use `/var/tmp` for creation of temporary files. E

10 WINDOWING AND TERMINAL INTERFACES

DRAFT COPY
March 18, 1997
File: Cchap10
386:adm.book:sum

The System V Window System

> **NOTE** This chapter, ‘‘Windowing and Terminal Interfaces’’, describes the optional ABI extension for the System V Window System.

The System V window system is a graphical window system, and is used as an interface for high-resolution bitmapped display devices. The window system allows multiple cooperating applications to appear on a display screen simultaneously. A server process arbitrates a shared display, a keyboard, and a pointing device, and performs I/O on behalf of one or more client applications. Client applications may execute in the local processor’s memory space, or may run on a remote processor and communicate with the server through a network connection.

The System V window system supports concurrent, overlapping windows, and windows can be created within other windows. The system supports text, two-dimensional graphics, and imaging.

The System V window system is completely network-transparent, in that a client
program can be running on any machine in a network and the client and server
programs need not be executing on machines sharing a common architecture. E
When the client and server reside on different machines, messages are transmitted E
over the network. When the client and server reside on the same machine, mes-
sages are transmitted using a local inter-process communication (IPC) mechanism.

Applications for the System V window system may be written to either the X11
Window System interface, the X11 Toolkit Intrinsics, the X11 Nonrectangular Win- M
dow Shape Extension, or Motif. M

The X11 interface contains mandatory parts of the System V Window System. All
implementations that include a window system extension must support
the X11 interface, the X Toolkit Intrinsics interface, the X11 Nonrectangular Win- M
dow Shape Extension, and Motif. The X11 interface has base and optional com- M
ponents.

X Window System Overview

A client application communicates with the X capabilities of the System V window system server using the X11 protocol. The X11 protocol specifies exchange format, rules for data exchange, and message semantics, but is policy-free and does not impose any specific appearance on the interface. The look and feel of a particular interface is defined by the window manager and different toolkits that define a higher-level program interface to the X capabilities.

The X Version 11 protocol defines the format, syntax, common types, errors codes, keyboard keycodes, pointers, predefined atoms, connection setup, requests, connection close, and events. A detailed description of these can be found in the *X Window System Protocol, Version 11 Specification* (Massachusetts Institute of Technology, 1991).

System V Window System Components

The System V Window System includes the following components. Some components of the window system are `base` components of the *ABI* and must be present, and others are optional components. Because of this mixture, each component's status has been explicitly marked in the list below, together with the component's description.

libX

NOTE The `libX` library is a base *ABI* component and must be present on an ABI-conforming system which has a window system extension.

The X library, `libX`, generates the X11 protocol and buffers traffic between each client application and the server. A full specification of the X library and its contents can be found in *Xlib - C Language Interface, X Window System, X Version 11, Release 5,* (Massachusetts Institute of Technology, 1991). The ABI will track upward-compatible future releases of the X library. *G*

X Toolkit Intrinsics

NOTE The `libXt` library is a base *ABI* component and must be present on an ABI-conforming system which has a window system extension. G

The X Toolkit Intrinsics library `libXt` provides a framework for building X-based toolkits. A full specification of the X Toolkit Intrinsics can be found in *X Toolkit Intrinsics - C Language X Interface, X Window System, X Version 11, Release 5,* (Massachusetts Institute of Technology, 1991). *G*

X11 Nonrectangular Window Shape Extension

> **NOTE** The `libXext` library is a base *ABI* component and must be present on an ABI-conforming system which has a window system extension.

The X11 Nonrectangular Window Shape Extension library Version 1.0, libXext, provides support for irregularly shaped windows under X11. A full specification of the X11 Nonrectangular Window Shape Extension can be found in *X11 Nonrectangular Window Shape Extension Version 1.0, X Version 11, Release 5,* (Massachusetts Institute of Technology, 1991).

Motif

> **NOTE** The `libXm` and `libMrm` libraries are base *ABI* components and must be present on an ABI-conforming system which has a window system extension.

The Motif libraries, `libXm` and `libMrm` provide support for windowing applications. A full specification of the Motif libraries can be found in the *Motif Programmer's Reference Revision 1.2,* (Open Software Foundation,1992).

The following programs are not required to be present on an ABI-conforming system. However, the *protocols* that these programs use to communicate are considered part of the *ABI* and must be present in an environment which supports window systems. Hence all ABI-conforming systems which offer these services must provide them as follows.

server
: The server controls the user's display. As such, a server is usually available for each type of display that can be connected to a system. It manages device dependencies, enabling client applications to be device-independent, although some client applications may need to be aware of the resolution, aspect ratio, or depth of the display device being used. Servers include protocol interpretation facilities for X and PostScript extensions. All ABI-conforming window servers which support X must support the entire X protocol referenced above. (The server side of X protocol communicates with `libX`).

window manager
: The window manager program allows a user to manipulate (for example, move and resize) windows. All client applications that use the windowing system use the facilities of the window manager. Well-behaved clients must follow the ICCCM. All ABI-conforming window managers which support X must support the entire X window manager protocol.

Summary of Requirements

The ‘‘Windowing and Terminal Interfaces’’ section of the *System V ABI* has a layered set of requirements for conforming systems. Those requirements are summarized here.

- The presence of a windowing system is optional on an ABI-conforming system.
- If a windowing system is present on a conforming system, `libX`, `libXt`, `libext`, `libXm`, and `libMrm` must be provided. G M

11 DEVELOPMENT ENVIRONMENTS FOR AN ABI SYSTEM

DRAFT COPY
March 18, 1997
File: Cchap11
386:adm.book:sum

Development Environments

NOTE THE FACILITIES AND INTERFACES DESCRIBED IN THIS SECTION ARE OPTIONAL COMPONENTS OF THE *System V Application Binary Interface*.

Any system may be used to provide a development environment for ABI conforming applications. This chapter describes the commands, options, libraries, and path mechanisms necessary to produce an ABI conforming application. This development environment need not be hosted on an ABI conforming implementation.

Commands

The following commands, defined in the *X/Open CAE Specification, Issue 4.2* and X
the SD_CMD section of the *System V Interface Definition, Fourth Edition* (see the X
Conformance Rule of Chapter 1) are a part of an ABI development environment. All commands defined here are mandatory in ABI Development Environments, except that the command `as` need not be present when ABI conformant code is generated directly by the compiler.

ABI Development Commands

`as`#	`cc`#	`ld`	
`m4`	`lex`	`yacc`	
`c89`†			X

Command is DEPRECATED X

†As defined in the Commands and Utilities volume of the *X/Open CAE Specification, Issue 4.2*. X

Each command shall accept the following required options and provide the functionality required for each option listed, in the *X/Open CAE Specification, Issue 4.2* X
and the *System V Interface Definition, Fourth Edition* (see the Conformance Rule in X
chapter 1).

as

SVID Fourth Edition Options to AS	
o	m

The as command if present shall produce output compliant to chapters 4 and 5 of this document, and chapters 4 and 5 of the appropriate *Processor-specific Supplement*.

cc

SVID Fourth Edition Options to CC			
B	c	d	D
G	I	K	l
L	o	O	U
Y			

M

The cc command shall generate output compliant to Chapter 4 and 5 of this document and Chapter 4 and 5 of the appropriate *Processor-specific Supplement*.

ld

SVID Fourth Edition Options to ld			
a	B	d	e
G	h	l	o
r	s	u	L
Y	z		

The command ld shall generate output compliant to Chapters 4 and 5 of this document and Chapters 4 and 5 of the appropriate *Processor-specific Supplement*.

lex

SVID Fourth Edition Options to lex			
c	t	v	n

m4

SVID Fourth Edition Options to m4		
s	D	U

yacc

SVID Fourth Edition Options to yacc			
v	d	l	t

As there may be multiple development environments hosted on a system, different values for PATH may be required to access each development environment, but all required commands shall be accessible by at least one value of PATH. The processor supplement for each architecture shall state how the value of PATH is to be used to find the location of a development environment for that architecture when it is not the native development environment. If the system is itself ABI conforming and hosts a development environment for its run-time system, the development environment for the run-time system shall be accessible using the system default value for PATH.

To enable the *System V Application Binary Interface, Edition 4.1* environment and X
functionality, applications must use the cc compiler driver. This driver will have X
an implementation-specific sequence of -D directives, include files or libraries to X
enable access to *System V Application Binary Interface, Edition 4.1* features. As a X
result the executable created will have the _ _xpg4 flag set to a value appropriate X
to the base API the application wishes to conform to.

NOTE All .o's should be compiled such that they either don't assume any specific X
shell (or other syntactic feature), or they presume the same shell (or other X
syntactic feature) across all .o's. Information is contained in linked executables, not in individual .o's.

Software Packaging Tools

A development environment for ABI applications shall include each of the following commands as defined in the `AS_CMD` section of SVID Fourth Edition.

`pkgproto` `pkgtrans` `pkgmk`

The `pkgtrans` command shall generate output compliant with Chapter 2 of this document.

Static Archives

Frequently applications must rely on groups of object files not required to be present on an ABI conforming implementation. These may be provided in static archives provided with the development environment. If each member of the archive is itself ABI conforming, then an ABI conforming application may statically link members from this archive and still be ABI conforming. If extensions to an archive are not ABI conforming, then an ABI conforming application may not include that extension in an executable.

All development environments for ABI applications shall contain ABI conforming versions of each of the following libraries.

```
libm
libcurses
```
M

The relevant processor supplement for each architecture shall define the path to the directory that contains these libraries. The following are entry-points that must be defined for each respective library as defined in SVID Fourth Edition.

Figure 11-1: Required `libm` Functions

acos	atanh	erfc	hypot	log10
acosh	cbrt	exp	j0	pow
asin	ceil	fabs	j1	remainder
asinh	cos	floor	jn	sin
atan	cosh	fmod#	lgamma	sinh
atan2	erf	gamma#	log	sqrt

Function is DEPRECATED X

Figure 11-2: Required `libcurses` Functions M

addch	dupwin	insdelln	mvgetstr	M
addchnstr	echo	insertln	mvgetwch	M
addchstr	echochar	insnstr	mvgetwstr	M
addnstr	echowchar	insnwstr	mvinch	M
addnwstr	endwin	insstr	mvinchnstr	M
addstr	erase	instr	mvinchstr	M
addwch	erasechar	inswch	mvinnstr	M
addwchnstr	filter	inswstr	mvinnwstr	M
addwchstr	flash	intrflush	mvinsch	M
addwstr	flushinp	inwch	mvinsnstr	M
attroff	getbegyx	inwchnstr	mvinsnwstr	M
attron	getch	inwchstr	mvinsstr	M
attrset	getmaxyx	inwstr	mvinstr	M
baudrate	getnwstr	is_linetouched	mvinswch	M
beep	getparyx	is_wintouched	mvinswstr	M
bkgd	getstr	isendwin	mvinwch	M
bkgdset	getsyx	keyname	mvinwchnstr	M
border	getwch	keypad	mvinwchstr	M
box	getwin	killchar	mvinwstr	M
can_change_color	getwstr	leaveok	mvprintw	M
cbreak	getyx	longname	mvscanw	M
clear	halfdelay	meta	mvwaddch	M
clearok	has_colors	move	mvwaddchnstr	M
clrtobot	has_ic	mvaddch	mvwaddchstr	M
clrtoeol	has_il	mvaddchnstr	mvwaddnstr	M
color_content	hline	mvaddchstr	mvwaddnwstr	M
copywin	idcok	mvaddnstr	mvwaddstr	M
curs_set	idlok	mvaddnwstr	mvwaddwch	M
def_prog_mode	immedok	mvaddstr	mvwaddwchnstr	M
def_shell_mode	inch	mvaddwch	mvwaddwchstr	M
del_curterm	inchnstr	mvaddwchnstr	mvwaddwstr	M
delay_output	inchstr	mvaddwchstr	mvwdelch	M
delch	init_color	mvaddwstr	mvwgetch	M
deleteln	init_pair	mvcur	mvwgetnwstr	M
delscreen	initscr	mvdelch	mvwgetstr	M
delwin	innstr	mvderwin	mvwgetwch	M
derwin	innwstr	mvgetch	mvwgetwstr	M
doupdate	insch	mvgetnwstr	mvwin	M

Figure 11-2: Required `libcurses` **Functions** (continued) M

mvwinch	putp	standout	waddwch	M
mvwinchnstr	putwin	start_color	waddwchnstr	M
mvwinchstr	qiflush	subpad	waddwchstr	M
mvwinnstr	raw	subwin	waddwstr	M
mvwinnwstr	redrawwin	syncok	wattroff	M
mvwinsch	refresh	termattrs	wattron	M
mvwinsnstr	reset_prog_mode	termname	wattrset	M
mvwinsnwstr	reset_shell_mode	tgetent	wbkgd	M
mvwinsstr	resetty	tgetflag	wbkgdset	M
mvwinstr	restartterm	tgetnum	wborder	M
mvwinswch	ripoffline	tgetstr	wclear	M
mvwinswstr	savetty	tgoto	wclrtobot	M
mvwinwch	scanw	tigetflag	wclrtoeol	M
mvwinwchnstr	scr_dump	tigetnum	wcursyncup	M
mvwinwchstr	scr_init	tigetstr	wdelch	M
mvwinwstr	scr_restore	timeout	wdeleteln	M
mvwprintw	scr_set	touchline	wechochar	M
mvwscanw	scrl	touchwin	wechowchar	M
napms	scroll	tparm	werase	M
newpad	scrollok	tputs	wgetch	M
newterm	set_curterm	tputs	wgetnstr	M
newwin	set_term	typeahead	wgetnwstr	M
nl	setscrreg	unctrl	wgetstr	M
nocbreak	setsyx	ungetch	wgetwch	M
nodelay	setterm	ungetwch	wgetwstr	M
noecho	setupterm	untouchwin	whline	M
nonl	slk_attroff	use_env	winch	M
noqiflush	slk_attron	vidattr	winchnstr	M
noraw	slk_attrset	vidputs	winchstr	M
notimeout	slk_clear	vline	winnstr	M
overlay	slk_init	vwprintw	winnwstr	M
overwrite	slk_label	vwscanw	winsch	M
pair_content	slk_noutrefresh	waddch	winsdelln	M
pechochar	slk_refresh	waddchnstr	winsertln	M
pechowchar	slk_restore	waddchstr	winsnstr	M
pnoutrefresh	slk_set	waddnstr	winsnwstr	M
prefresh	slk_touch	waddnwstr	winsstr	M
printw	standend	waddstr	winstr	M

Figure 11-2: Required `libcurses` **Functions** (continued) M

winswch	wmove	wscanw	wsyncdown	M
winswstr	wnoutrefresh	wscrl	wsyncup	M
winwch	wprintw	wsetscrreg	wtimeout	M
winwchnstr	wredrawln	wstandend	wtouchln	M
winwchstr	wrefresh	wstandout	wvline	M
winwstr				M

12 NETWORKING

Networking

NOTE This chapter is new to the *System V Application Binary Interface, Third Edition*.

ABI-conforming systems may support some level of networking, ranging from peer-to-peer to loopback networks. This section describes the network transport level services needed to support the Internet and ISO transport protocols. In addition, it defines required support for basic process to process communication over a network.

Required STREAMS Devices and Modules

The following device drivers must exist on an ABI-conforming system. Their required functionality shall be that specified in the *System V Interface Definition, Fourth Edition* and the *DDN Protocol Handbook, DARPA Internet Protocols*.

Figure 12-1: Required STREAMS Devices

/dev/ticlts	/dev/ticots	/dev/ticotsord	transport loopback support
/dev/tcp	/dev/udp		internet support
/dev/ptmx	/dev/pts/*digits*		pseudo terminal support

Where *digits* are decimal numbers.

The following required STREAMS modules shall be present on conforming systems and their functionality shall be that defined in section BA_DEV of the *System V Interface Definition, Fourth Edition*.

Figure 12-2: Required STREAMS Modules

ldterm	ptem	pckt	pseudo terminal support
tirdwr	timod		transport level support
connld	pipemod		IPC support

Binary interface support for these modules and drivers are defined in the following sections.

Required Interprocess Communication Support

Pseudo Terminal Support

There shall be an appropriate entry in the `/dev/pts` directory for each slave-side pseudo-terminal available on the implementation. Conforming systems shall provide a minimum of 16 pseudo-terminals. The default initial state of a pseudo-terminal, as reported by `tcgetattr`, shall be the same as the default initial state of a terminal as specified in `termio`(BA_DEV).

NOTE The default baud rate as specified by `termio`(BA_DEV) is 300 baud. As this may be inappropriate for pseudo terminals, there are exceptions.

STREAM Based Pipe Support

Functionality for interprocess communication by way of STREAMS-based pipes shall be supported by ABI-conforming systems. STREAMS modules `connld` and `pipemod` must be present in support of this.

Required Transport Layer Support

A transport layer application may access transport services through the ISO or Internet frameworks. This document supports transport access via the XTI interface as it makes use of the *timod* module. The required functionality for these modules is defined in the BA_DEV section of the *System V Interface Definition,* X
Fourth Edition. The definition of XTI is described in the Networking Services X
volume of the *X/Open CAE Specification, Issue 4*.

In order to improve standards conformance and take advantage of the latest technology in XTI interfaces, the TLI interfaces have been DEPRECATED. Applications which make use of TLI should migrate to XTI as a replacement. TLI is a subset of XTI except where noted.

To achieve binary interoperability, an ABI-conforming system must consistently define variables and data structures. The next few displays contain mnemonics required on ABI-conforming systems. Their associated values are specified in the processor specific ABI.

Figure 12-3: TLI-XTI Error Codes

TACCES	TBADQLEN*	TNODATA	TPROTO*
TADDRBUSY*	TBADSEQ	TNODIS	TPROVMISMATCH*
TBADADDR	TBUFOVFLW	TNOREL	TQFULL*
TBADDATA	TFLOW	TNOSTRUCTYPE*	TRESADDR*
TBADF	TINDOUT*	TNOTSUPPORT	TRESQLEN*
TBADFLAG	TLOOK	TNOUDERR	TSTATECHNG
TBADNAME*	TNOADDR	TOUTSTATE	TSYSERR
TBADOPT			

*Function is new to System V ABI Edition 4.1. X

Figure 12-4: `t_look` Events

T_LISTEN	T_EXDATA	T_UDERR	T_CONNECT
T_DISCONNECT	T_ORDREL	T_DATA	T_ERROR
T_EVENTS			
T_GODATA	T_GOEXDATA		

M

Figure 12-5: XTI Flag Definitions

```
T_MORE  T_NEGOTIATE     T_DEFAULT       T_EXPEDITED
T_CHECK T_SUCCESS       T_FAILURE
T_CURRENT T_NOTSUPPORT T_PARTSUCCESS T_READONLY
```
X

Figure 12-6: XTI Service Types

```
T_COTS   T_COTS_ORD   T_CLTS
```

The following are required structure types and bit fields used when dynamically allocating XTI structures via the function `t_alloc` call: X

Figure 12-7: Flags to be used with `t_alloc`

```
T_BIND    T_OPTMGMT    T_CALL
T_DIS     T_UNITDATA   T_UDERROR
T_INFO    T_ADDR       T_OPT
T_UDATA   T_ALL
```

Figure 12-8: XTI Application States

```
T_UNINIT   T_UNBND      T_IDLE     T_OUTCON
T_INCON    T_DATAXFER   T_OUTREL   T_INREL
T_FAKE     T_NOSTATES
```

Figure 12-9: XTI values for t_info flags member

T_SENDZERO

Required Transport Loopback Support

An ABI-conforming system shall support a transport level loopback facility. The device driver support is as defined by `ticlts(BA_DEV)`, `ticots(BA_DEV)` and `ticotsord(BA_DEV)` of the *System V Application Binary Interface, Fourth Edition*. X

Optional Internet Transport Support

A tcp implementation conforming to RFC 793 (as revised by by RFC 1122) shall support a device driver implementing the T_COTS_ORD service type. A udp implementation conforming to RFC 768 shall support a device driver implementing the T_CLTS service type. These services shall be available to the TLI / XTI functions and the provided functionality shall be consistent with TCP (RFC 793) and UDP (RFC 768).

Address Format

The XTI netbuf structure is used to pass TCP/IP addresses. The addr.buf component should point to a struct sockaddr_in. When used by XTI requests, the sin_zero field of this structure shall be zero and the sin_family field shall contain AF_INET.

Programming Interfaces

A conforming TCP/IP implementation shall implement the following transport provider-specific functionality in addition to that required by RFC 793 and 791 (as revised by RFC 1122 and RFC 1349).

t_accept

Under TLI no options shall be supported. The returned udata length shall be zero. X

t_bind

The INADDR_ANY address may be used. If INADDR_ANY is supplied as the address, then the loopback or a local address will be used.

t_connect

No data shall be sent with the connection. The sin_addr field of the struct sockaddr_in pointed to by sndcall->addr.buf may contain a valid TCP/IP address. If rcvcall is not NULL, then the buffer identified by rcvcall->addr.buf shall be filled in with a sockaddr_in structure.

t_getinfo

The following default values shall be returned from the devices associated with the TCP/IP transport provider.

Values Returned by `/dev/tcp`

`addr`	varies	X
`options`	varies	X
`tsdu`	0 (byte stream)	
`etsdu`	–1	
`connect`	–2 (not supported)	
`discon`	–2 (not supported)	
`servtype`	`T_COTS_ORD`	
`flags`	`T_SENDZERO`	

Values Returned by `/dev/udp`

`addr`	varies	X
`options`	varies	X
`tsdu`	varies	X
`etsdu`	–2 (not supported)	
`connect`	–2 (not supported)	
`discon`	–2 (not supported)	
`servtype`	`T_CLTS`	
`flags`	`T_SENDZERO`	

t_listen

The result `udata` variable shall have zero length. Under TLI the result `opt` variable shall have zero length. The buffer identified by `call->addr.buf` shall receive a `struct sockaddr_in` that identifies the host originating the connection. X

t_open

The returned transport characteristics shall be the same as those returned by `t_getinfo`.

t_optmgmt

TLI

Under TLI the following options may be accessible through t_optmgmt: X

Figure 12-10: TCP Options

TCP_NODELAY

Figure 12-11: IP Options

IPOPT_EOL	End of options list
IPOPT_NOP	No operation
IPOPT_LSRR	Loose source and record route
IPOPT_SSRR	Strict source and record route

Options are specified in an options buffer with the opthdr data structure format. An options buffer contains one or more options, with each followed by 0 or more values. The len field of opthdr specifies the length of the option value in bytes. This length must be a multiple of sizeof(long)

Options may be manipulated at the TCP level and the IP level via the TLI t_optmgmt call. To manipulate options at the TCP level, IPPROTO_TCP is specified to t_optmgmt. For the IP level, IPPROTO_IP should be specified for t_optmgmt. The header <netinet/tcp.h> contains the definitions for TCP level options, while the IP level options are defined in <netinet/ip.h>.

All TCP level options are 4 bytes long. A boolean option is either set or reset. Any non-zero value will assert (set) the option while only zero will clear the option.

The IP options consist of a string of IPOPT_* values. These options will be set in every outgoing datagram. Options whose function is not explicitly specified above are copied directly into the output. See IP and RFC 1122 for details.

TCP Level Options If the TCP_NODELAY option is set, a conforming system shall not delay sending data in order to coalesce small packets. When the option is reset, a system may defer sending data in an effort to coalesce small packets to conserve network bandwidth.

IP Level Options

`IPOPT_LSRR` The `IPOPT_LSRR` option enables the loose source and record route option as specified in RFC 1122.

`IPOPT_SSRR` The `IPOPT_SSRR` option enables the strict source and record route option as specified in RFC 1122.

`IPOPT_NOP` The `IPOPT_NOP` does nothing. Since the length of the IP options must be a multiple of 4, this option is useful as a pad.

`IPOPT_EOL` This option identifies the end of an IP option sequence.

XTI

The following information is relevant under XTI. The following options may be accessible through `t_optmgmt`: X X

Figure 12-12: TCP Options X X

TCP Level	INET_TCP
TCP Level Options	TCP_NODELAY
	TCP_MAXSEG
	TCP_KEEPALIVE

X X X X

Figure 12-13: UDP Options X

UDP Level	INET_UDP
UDP Level Options	UDP_CHECKSUM

X X

Figure 12-14: IP Options

IP Level	INET_IP
UDP Level Options	IP_OPTIONS
	IP_TTL
	IP_TOS
	IP_REUSEADDR
	IP_DONTROUTE
	IP_BROADCAST

t_rcv

TCP/IP urgent data shall be returned as expedited data with the semantics described in `t_rcv` for expedited data.

t_rcvconnect

The result `udata` variable shall have zero length. Under TLI, the result `opt` variable shall have zero length.

t_rcvudata

If `opt` is non NULL and there were IP or UDP options sent with the datagram, the IP or UDP options should be returned in `opt`. Under TLI if `opt` is NULL and IP options were sent, they should be silently discarded. Under TLI, UDP will not send options.

t_rcvuderr

Under TLI the returned length of the `opt` variable shall be zero.

t_snd

If `T_EXPEDITED` is set in the *flags* argument, TCP will send the data as urgent data. The TCP urgent data pointer will point to the first byte of data in the next data sent by `t_snd`.

t_snddis

Data shall not be sent with the disconnect request.

t_sndrel

The T_COTS_ORD service of the transport provider (TCP) shall support this function.

t_sndudata

Under TLI, for /dev/udp, the opt field may contain IPPROTO_IP options. The UDP protocol shall support sending zero length data. X

Data Structures

To support interoperability between networked machines, an ABI-conforming system supporting the Internet family protocols must support the following data structures.

There must exist a sockaddr_in data structure containing at least the following elements. Under TLI there must exist a opthdr data structure containing at least the following elements. X

Figure 12-15: Data Structures

```
struct sockaddr_in {
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};

struct opthdr {
      long        level;
      long        name;
      long        len;
};
```

IN Index

Index

B

C

D

G

H

I

K

L

M

P

S

T

U

V

W

X

DRAFT COPY
March 18, 1997
File: index
386:adm.book:sum

Y