IRIX™ System Programming Guide

About This Guide

This guide discusses a variety of issues and tools involved in programming under the IRIX™ operating system. It describes the components of the IRIS®–4D™ compiler system, and other programming tools and interfaces.

What This Guide Contains

This guide contains the following chapters:

- **Chapter 1, "System Overview,"** provides a brief overview of IRIX.
- **Chapter 2, "Using the Compiler System,"** describes the components of the IRIS–4D Series compiler system and explains how to use them.
- **Chapter 3, "Dynamic Shared Objects,"** explains how to build and use dynamic shared objects, which replace the static shared libraries used by previous versions of IRIX.
- **Chapter 4, "Improving Program Performance,"** explains how to reduce program execution time using profiling and optimization tools.
- **Chapter 5, "Inter–Process Communication,"** describes System V and IRIX inter–process communication mechanisms.
- **Chapter 6, "File and Record Locking,"** describes how to lock and unlock files and parts of files from within a program.
- **Chapter 7, "Using Real–Time Programming Features,"** describes features available for real–time programming.
- **Chapter 8, "Working with Fonts,"** discusses typography and font use on Silicon Graphics computers, and describes the Font Manager library.
- **Chapter 9, "Internationalization,"** explains how to create an application that can be adapted for use in different countries.
- **Appendix A, "Position–Independent Coding in Assembly Language,"** describes assembly language coding techniques required by this version of IRIX.
- **Appendix B, "ISO 3166 Country Names and Abbreviations,"** lists country codes for use with internationalization and localization.

What You Should Know Before Reading This Guide

This guide is for anyone who wants to program effectively under IRIX. We assume you are familiar with the IRIX (or UNIX®) operating system and a programming language such as C. This guide does not explain how to write or compile programs.

Suggestions for Further Reading

The following references provide more information on internationalization:

- Nye, Adrian. *Xlib Programming Manual for Version 11 of the X Window System, Volume One*. Sebastapol, CA: O'Reilly & Associates, ISBN 1–56592–002–3 (covers Release 4 and Release 5).
- Scheifler, Robert and Gettys, Jim. *X Window System, Third Edition*. Digital Press, ISBN 1–55558–088–2.
- X/Open Company, Ltd. *X/Open Portability Guide*. Englewood Cliffs: Prentice Hall Publishing Company, ISBN 0–13–685819–8 (set of 7 volumes).

Conventions Used in This Guide

This guide uses the following conventions and symbols:

`Courier` In text, the Courier font represents function names, file names, and keywords. It is also

	used for command syntax, output, and program listings.
`bold`	Boldface is used along with Courier font to represent user input.
italics	Words in italics represent characters or numerical values that you define. Replace the abbreviation with the defined value. Also, italics are used for manual page names and commands. The section number, in parentheses, follows the name.
[ ]	Brackets enclose optional items.
{ }	Braces enclose two or more items; you must specify at least one of the items.
|	The OR symbol separates two or more optional items.
…	A horizontal ellipsis in a syntax statement indicates that the preceding optional items can appear more than once in succession.
( )	Parentheses enclose entities and must be typed.

The following two examples illustrate the syntax conventions:

```
DIMENSION a(d) [,a(d)] …
```

indicates that the Fortran keyword DIMENSION must be typed as shown, that the user−defined entity *a(d)* is required, and that one or more of *a(d)* can be specified. The parentheses ( ) enclosing *d* are required.

```
{STATIC | AUTOMATIC} v [,v] …
```

indicates that either the STATIC or AUTOMATIC keyword must be typed as shown, that the user−defined entity *v* is required, and that one or more *v* items can be specified.

Chapter 1

System Overview

This chapter provides a brief overview of the IRIX operating system, Silicon Graphics's implementation of the UNIX operating system.

About the IRIX Operating System

IRIX is based on UNIX System V, Release 4 (SVR4). IRIX provides the standard SVR4 programming interfaces together with support for features such as fast 3D graphics, digital media, and symmetric multiprocessing.

Standards Compliance

IRIX complies with the following standards:

- System V Interface Definition, Issue 3 (SVID3), which is the defining document for SVR4. IRIX provides the SVR4 Applications Programming Interface (API) and Applications Binary Interface (ABI) as defined in SVID3.
- X/Open Portability Guide, Issue 3 (XPG3), which specifies a set of programming interfaces to be provided by operating systems, in order to facilitate the writing of portable programs.
- POSIX P1003.1, another standard for portable programming.
- X Window System, Version 11, Release 5 (X11R5). IRIX provides the latest version of the industry−standard X Window System.
- OSF/Motif™ Release 1.2. IRIX includes IRIS IM™, the Silicon Graphics port of the industry−standard OSF/Motif user−interface toolkit.
- OpenGL. IRIX provides a full implementation of this emerging standard for 3D graphics.

X Window System

The IRIX X Server includes support for multiple visuals, overlay windows, the Display PostScript™ Extension, the Shape Extension, the X Input Extension, the Shared Memory Extension, and simultaneous display on multiple graphics monitors. In addition, the IRIX X server has been extended to support IRIS GL and OpenGL programs.

Internationalization

IRIX includes support for internationalizing programs, so that they can be adapted for use in different countries, in different languages, and so forth. IRIX supports the internationalization facilities described in two of the standards mentioned above, SVR4 and XPG3. Application programs can use either scheme. Internationalization is described in **Chapter 9, "Internationalization."**

Networking

IRIX includes several interfaces for network programming. In addition to standard SVR4 networking (the Transport Layer Interface, or TLI), IRIX offers BSD network interfaces (sockets). IRIX also provides an implementation of the Sun Microsystems Remote Procedure Call (RPC) library. For information on network programming, see the *IRIX Network Programming Guide*.

Silicon Graphics Extras

In addition to standards, IRIX provides a number of special features.

Fast 3D Graphics

Support for fast 3D graphics is built into IRIX. Most Silicon Graphics systems are equipped with specialized graphics hardware. Programs can take advantage of this graphics support using either the IRIS Graphics Library™ (GL) or OpenGL interfaces.

The IRIS GL programming interfaces were the standard for 3D graphics under previous version of IRIX. They are provided largely for backward compatibility. The IRIS GL interfaces provide limited support for windowing and event handling. Programs that require more sophisticated event handling or windowing than the IRIS GL interfaces provide can use the X Window System interfaces for interacting with the window system, and the IRIS GL interfaces for rendering. Programs that manage windowing and event handling through the IRIS GL interfaces are referred to as "pure GL" programs, while programs that make use of the X Window System interfaces directly are referred to as "mixed−model" programs.

The OpenGL interfaces are designed to provide standard interfaces for 3D rendering across multiple platforms. The OpenGL interfaces do not include any support for windowing or event handling, allowing these to be dealt with differently on different platforms. Under IRIX, OpenGL programs use X for windowing and event handling, like mixed−model IRIS GL programs. The OpenGL interfaces are described in the *OpenGL Programming Guide*.

Symmetric Multiprocessing

IRIX is designed to take advantage of multiple processors, and provides interfaces to support parallel programming. These interfaces include low−overhead interprocess communication routines, described in **"IRIX IPC" in Chapter 5** , and multiprocessor control routines described in **Chapter 7, "Using Real−Time Programming Features."**

Real−Time Enhancements

IRIX includes a number of extensions to enable programs to achieve real−time response. These features are described in **Chapter 7, "Using Real−Time Programming Features."**

High−Performance I/O Extensions

IRIX includes a several special programming interfaces for high−performance I/O. These include memory−mapped files, asynchronous I/O, and direct I/O. These interfaces are described in **Chapter 7, "Using Real−Time Programming Features."**

Indigo Magic Environment

Indigo Magic is an end−user environment that provides an icon−oriented interface to the IRIX system. Users can launch applications and select files using icons instead of pathnames. You can find

Users can launch applications and select files using icons instead of pathnames. You can find information on integrating an application into the Indigo Magic environment in the *Indigo Magic Desktop Integration Guide*.

Chapter 2
Using the Compiler System

This chapter contains these sections:

- **"Overview"** lists the components of the IRIS–4D compiler system.
- **"Object File Format and Dynamic Linking"** discusses the major differences between the latest version of IRIX and previous versions.
- **"Source File Considerations"** explains source file naming conventions and the procedure for including header files.
- **"Compiler Drivers"** lists and explains the general compiler–driver options.
- **"Linking"** explains how to manually link–edit programs (using *ld* or a compiler driver) and how to compile multilanguage programs. It also describes Dynamic Shared Objects and how to link them into your programs.
- **"Debugging"** explains the compiler–driver options for debugging.
- **"Getting Information About Object Files"** explains how to use the object file tools to analyze object files.
- **"Using the Archiver to Create Libraries"** explains how to use the archiver, *ar*.

Overview

The IRIS–4D compiler system consists of a set of components that enable you to create executable programs using such languages as C, Fortran 77, and Pascal.

Table 2–1 summarizes the IRIS–4D compiler system components and the task each performs.

Table 2–1 Compiler System Functional Components

Tool	Task	Examples
Text editor	Write and edit programs	*vi*, *jot, emacs*
Compiler driver	Compile, link, and load programs	*cc*, *f77*, *pc*
Object file analyzer	Analyze object files	*elfdump*, *file*, *nm*, *odump*, *size*
Profiler	Analyze program performance	*prof*, *pixie*
Optimizer	Improve program performance	*uopt*
Archiver	Produce object–file libraries	*ar*
Runtime loader	Link Dynamic Shared Objects at runtime	*rld*
Debugger	Debug programs	*dbx*

A single program called a compiler driver (such as *cc, f77*, or *pc*) invokes the following major components of the compiler system (refer to **Figure 2–1**).

- Macro preprocessor (*cfe*, *cpp*, *acpp*)
- Parallel analyzer (*pca, pfa*)
- Scalar optimizer (*copt*)
- Compiler front end (*cfe, fcom*, *upas*, *accom_mp*, *ccom_mp*)
- Ucode tools (*ujoin, uld, umerge*)
- Optimizer (*uopt*)
- Code generator (*ugen*)
- Assembler (*as1*)
- Link editor (*ld*)

Note: C++ has a specialized driver, *CC*, with slightly different options from *cc*, *f77*, and *pc*. Refer to the *C++ Programming Guide* for details.

You can invoke a compiler driver with various options (described later in this chapter) and with one or more source files as arguments. All specified source files are automatically sent to the macro preprocessor.

Note: Preprocessing is now done by *cfe*. The old preprocessors (*cpp* for "traditional" Kernighan & Ritchie C, or *acpp* for ANSI C) are still available for non−compilation preprocessing and preprocessing for *copt*, *ccom_mp*, and *accom_mp*, in case you want to use them.

Although the macro preprocessor was originally designed for C programs, it is now run by default as part of almost all compilations. To prevent the preprocessor from being run, specify the *-nocpp* option on the driver command line.

If available, the parallel analyzers *pca* and *pfa* produce parallelized source code from standard source code. The result takes advantage of multiple CPUs (when present) to achieve higher computation rates. *pca* and *pfa* are part of the Power C and Power Fortran packages; for more information about these packages and how to obtain them, contact your dealer or sales representative.

The compilers proper, often called "front ends," translate source code into intermediate code. The available compiler front ends are *cfe* (C), *ccom_mp* and *accom_mp* (parallel C), *fcom* (Fortran 77), and *upas* (Pascal). *ujoin*, *uld*, *umerge,* and *uopt* comprise the optimization subsystem of the compiler system. (For more information about optimization and profiling, see **Chapter 4, "Improving Program Performance."**) *ugen* and *as1* make up the code−generation subsystem of the compiler system.

The link editor *ld* combines several object files into one, performs relocation, resolves external symbols, and merges symbol table information for symbolic debugging. The driver automatically runs *ld* unless you specify the *-c* option to skip the linking step.

To see the various utilities a program passes through during compilation, invoke the appropriate driver with the *-v* option (or *+v* for the C++ driver *CC*).

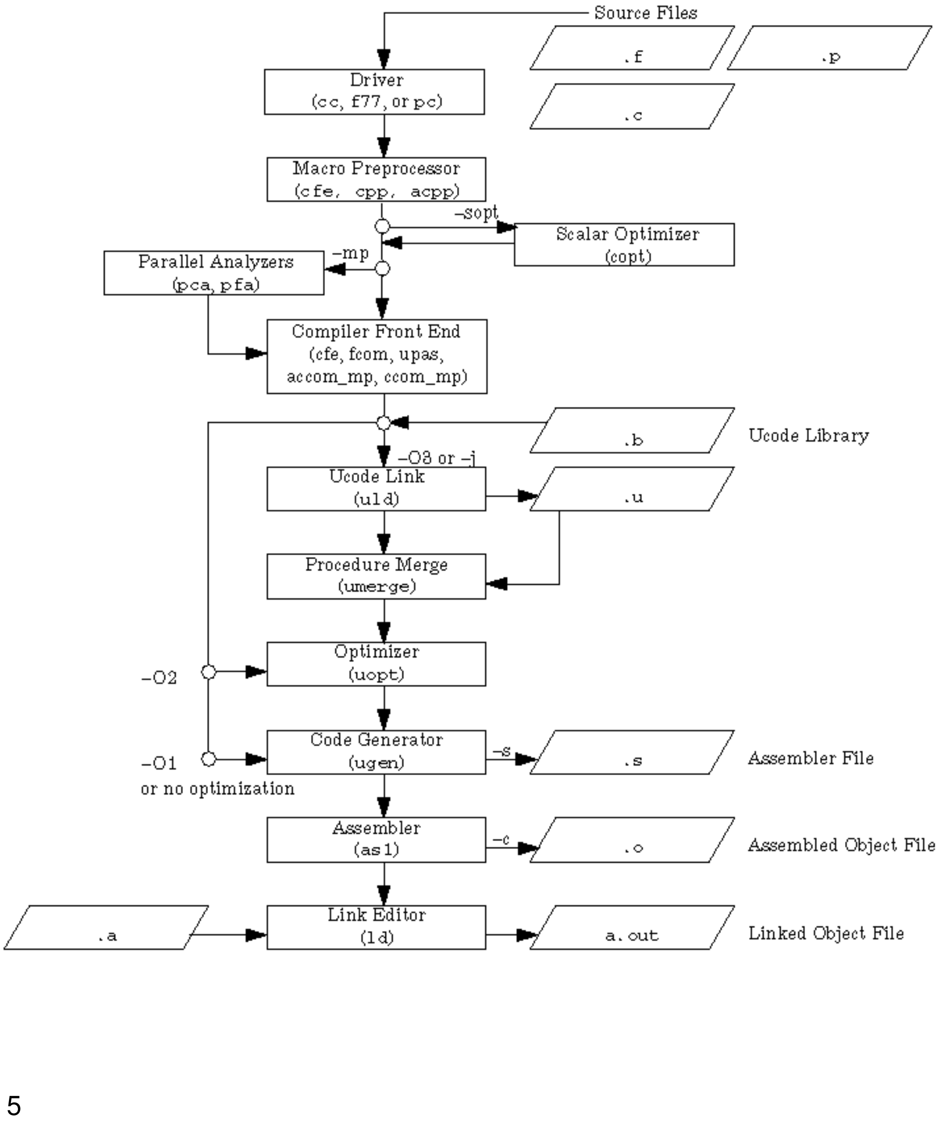

Figure 2–1 Compiler System Flowchart

Object File Format and Dynamic Linking

A new object file format was adopted in IRIX version 5.0. The major differences between the current compiler system and pre–5.0 compiler systems are summarized below:

- The compiler system uses a new format, Executable and Linking Format (ELF), for object files.
- The compiler system uses a new kind of shared library, the Dynamic Shared Object (DSO).
- The compiler system creates position–independent code (PIC) by default to support dynamic linking.

Executable and Linking Format

Previous versions of IRIX used an extended version of the Common Object File Format (COFF) for object files. The current compiler system produces ELF object files instead. ELF is the format specified by the System V Release 4 Applications Binary Interface (the SVR4 ABI). In addition, ELF provides support for Dynamic Shared Objects, described below. Types of ELF object files include:

- Relocatable files contain code and data in a format suitable for linking with other object files to make a shared object or executable.
- Dynamic Shared Objects contain code and data suitable for *dynamic linking*. Relocatable files may be linked with DSOs to create a dynamic executable. At runtime, the runtime linker combines the executable and DSOs to produce a process image.
- Executable files are programs ready for execution. They may or may not be dynamically linked.

COFF executables continue to run on new releases of IRIX, but the current compiler system has no facility for creating or linking COFF executables. COFF and ELF object files may not be linked together. To take advantage of new IRIX features, you must recompile your code.

IRIX executes all binaries that are compliant with the SVR4 ABI, as specified in the *System V Applications Binary Interface—Revised Edition* and the *System V ABI MIPS Processor Supplement*. However, binaries compiled under this version of the compiler system are not guaranteed to comply with the SVR4 ABI. The MIPS–specific version of the SVR4 ABI is referred to as the MIPS ABI. Programs that comply with the MIPS ABI can be run on any machine that supports the MIPS ABI.

Dynamic Shared Objects

IRIX 5.0 introduced a new kind of shared object called a Dynamic Shared Object, or DSO. The object code of a DSO is position–independent code (PIC), which can be mapped into the virtual address space of several different processes at once. DSOs are loaded at runtime instead of at linking time, by the runtime loader, *rld*. As is true for static shared libraries, the code for DSOs is not included in executable files; thus, executables built with DSOs are smaller than those built with non–shared libraries, and multiple programs may use the same DSO at the same time.

Static shared libraries are only supported under this release for the purposes of running old (COFF) binaries. The current compiler system has no facilities for generating static shared libraries.

Position–Independent Code

Dynamic linking requires that all object code used in the executable be position–independent code. For source files in high–level languages, you just need to recompile to produce PIC. Assembly language files must be modified to produce PIC; see **Appendix A, "Position–Independent Coding in Assembly Language,"** for details.

Position–independent code satisfies references indirectly by using a *global offset table* (GOT), which allows code to be relocated simply by updating the GOT. Each executable and each DSO has its own GOT.

The compiler system now produces PIC by default when compiling higher–level language files. All of the standard libraries are now provided as DSOs, and therefore contain PIC code; if you compile a program into non–PIC, you will be unable to use those DSOs. One of the few reasons to compile non–PIC is to build a device driver, which doesn’t rely on standard libraries; in this case, you should use the

-non_shared option to the compiler driver to negate the default option, *-KPIC*. For convenience, the C library and math library are provided in non−shared format as well as in DSO format (although the non−shared versions are not installed by default). These libraries can be linked *-non_shared* with other non−PIC files.

When running position−independent code, the global pointer is used to point to the global offset table, so you can no longer use the *-G* option to store data in the global pointer region (that is, *-KPIC*, the default, implies *-G 0*). The compiler ignores any user−specified *-G* number other than zero. For more information about this option, see the *ld*(1) manual page.

Source File Considerations

This section describes conventions for naming source files and including header files.

Source File Naming Conventions

Each compiler driver recognizes the type of an input file by the suffix assigned to the file name. **Table 2−2** describes the possible file name suffixes.

Table 2−2 Driver Input File Suffixes

Suffix	Description
.s	Assembly source
.i	Preprocessed source code in the language of the processing driver
.c	C source
.f	Fortran 77 source
.p	Pascal source
.u	Ucode object file
.b	Ucode object library
.o	Object file
.a	Object library

If you purchased the C++ option, refer to the *C++ Programming Guide* for details about input files for the C++ driver, *CC*.

Examples

calc.p is an example of a Pascal program name.

When invoked as follows, *f77* assumes the file *tickle.i* contains Fortran statements (because the Fortran driver is specified). *f77* also assumes the file has already been preprocessed (because the suffix is *.i*), and therefore does not invoke the *cfe* preprocessor:

```
f77 −c tickle.i
```

Header Files

Header files, also called *include* files, contain information about the libraries they're associated with. They define such things as data structures, symbolic constants, and prototypes and parameters for the functions in the library. For example, the *stdio.h* header file describes, among other things, the data types of the parameters required by *printf()*. To use those definitions without having to type them into each of your source files, you can use the *#include* command to tell the macro preprocessor to include the complete text of the given header file in the current source file. Including header files in your source files allows you to specify such definitions conveniently and consistently in each source file that uses any of the library routines.

By convention, header file names have a *.h* suffix. Each programming language handles these files the same way, via the macro preprocessor.

Note: Do not put any code other than definitions in an include file, particularly if you intend to debug your program using *dbx*. The debugger recognizes an include file as only one line of source code, so source lines in an include file do not appear during debugging sessions.

Specifying a Header File

The *#include* command tells the preprocessor to replace the *#include* line with the text of the indicated header file. The usual way to specify a header file is with the line:

```
#include <filename>
```

where *filename* is the name of the header file to be included. The angle brackets (< >) surrounding the file name tell the macro preprocessor to search for the specified file only in directories specified by command−line options and in the default header−file directory (*/usr/include*).

Another specification format exists, in which the file name is given between double quotation marks:

```
#include "filename"
```

In this case, the macro preprocessor searches for the specified header file in the current directory first, then (if it doesn't find the requested file) goes on and searches in the other directories as in the angle−bracket specification.

Note: When you specify header files in your source files, the *#include* keyword should always start in column 1 (that is, the left−most column) to be recognized by the preprocessor.

Creating a Header File for Multiple Languages

A single header file can contain definitions for multiple languages; this setup allows you to use the same header file for all programs that use a given library, no matter what language those programs are in. To set up a shareable header file, create a *.h* file and enter the definitions for the various languages as follows:

```
#ifdef _LANGUAGE_C

C definitions

#endif

#ifdef _LANGUAGE_C_PLUS_PLUS

C++ definitions

#endif

#ifdef _LANGUAGE_FORTRAN

Fortran definitions

#endif

and so on for other language definitions
```

Note: To indicate C++ definitions you must use _LANGUAGE_C_PLUS_PLUS, not _LANGUAGE_C++.

You can specify the various language definitions in any order, but you must specify _LANGUAGE_ before the language name.

Compiler Drivers

The driver commands, such as *cc*, *f77*, and *pc*, call subsystems that compile, optimize, assemble, and link–edit your programs. This section describes driver options that you can use with any of the drivers.

Default Behavior for Compiler Drivers

At compilation time, you can select one or more options that affect a variety of program development functions, including debugging, optimization, and profiling facilities. You can also specify the names assigned to output files. However, some options have default values that apply if you do not specify the option.

When you invoke a compiler driver with source files as arguments, the driver calls other commands that compile your source code into object code. It then optimizes the object code (if requested to do so) and links together the object files, the default libraries, and any other libraries you specify.

Example

Given a source file *foo.c*, the default name for the object file is *foo.o*. The default name for an executable file is *a.out*. So the following example compiles source files *foo.c* and *bar.c* with the default options:

```
cc foo.c bar.c
```

This example produces two object files (*foo.o* and *bar.o*), then links those together with the default C library *libc* to produce an executable called *a.out*.

Note: If you compile a single source directly to an executable, the compiler does not create an object file.

General Options for Compiler Drivers

Table 2–3 describes some of the command–line options for IRIS–4D compiler drivers. Note that not all of the options work with every driver.

Note: **Table 2–3** lists only the most frequently used options, not all available options. See the *cc*(1), *pc* (1), and *f77*(1) manual pages for a complete list of options.

You can use the compiler system to generate profiled programs that, when executed, provide operational statistics. To perform this procedure, use the *-p* compiler option (for pc sampling information) and the *pixie* program (for profiles of basic block counts). Refer to **Chapter 4, "Improving Program Performance,"** for details.

In addition to the general options in **Table 2–3**, each driver has options that you typically won't use. These options primarily aid compiler development work. For information about nonstandard driver options, consult the appropriate driver manual page.

Table 2–3 General Driver Options

Option	Purpose
-c	Prevents the link editor from linking your program after assembly code generation. This option forces the driver to produce a *.o* file after the assembler phase, and prevents the driver from producing an executable file.
-C	(C driver only) Used with the *-P* or *-E* option. Prevents the macro preprocessor from stripping comments. Use this option when you suspect the preprocessor is not producing the intended code and you want to examine the code with its comments. Note that *-C* is an option to *cfe*; this option is passed along to *cfe* if you specify it with *cc*.
-C	(Pascal and Fortran drivers only) Generates code that invokes range checking for subscripts during program execution.
-Dname[*=def*]	Defines a macro *name* as if you had specified a *#define* in your program. If you do not specify a definition with *=def*, *name* is set to 1.
-E	(C driver only) Runs only the macro preprocessor and sends results to the standard output. To retain comments, use the *-C* option as well. Use *-E* when you suspect the preprocessor is not producing the intended code.
-Idirname	Adds *dirname* to the list of directories to be searched for specified header files.

	These directories are always searched before the default directory, */usr/include*.
-mips1	Generates code using the instruction set of the MIPS R2000/R3000 RISC architecture. This is the default.
-mips2	Generates code using the MIPS II instruction set (MIPS I + R4000 specific extensions). Note that code compiled with −mips2 does not run on R2000/R3000 based machines.
-nocpp	Suppresses running of the macro preprocessor on the source files prior to processing.
-non_shared	Turns off the default option, *-KPIC*, to produce non−shared code. This code can be linked to only a few standard libraries (such as *libc.a* and *libm.a*) that are provided in non−shared format, in the directory */usr/lib/nonshared*. You should use this option only when building device drivers.
-nostdinc	Suppresses searching of */usr/include* for the specified header files.
-o filename	Names the result of the compilation *filename*. If an executable is being generated, it is named *filename* rather than the default name, *a.out*. If a single source file is compiled with *-c*, the object is named *filename* (not, it should be noted, *filename.o*; if you want the object file name to end with *.o*, you should specify that in the argument to *-o*). Otherwise, this option is ignored.
-P	Runs only the macro preprocessor on the files and puts the result of each file in a *.i* file. Specify both *-P* and *-C* to retain comments.
-S	Similar to *-c*, except that it produces assembly code in a *.s* file instead of object code in a *.o* file.
-Uname	Overrides a definition of the macro *name* that you specified with the *-D* option, or that is defined automatically by the driver. Note that this option does not override a macro definition in a source file, only on the command line.
-v	Lists compiler phases as they are executed. Use this option to see the default options for each compiler phase along with the options you've specified.
-w	Suppresses warning messages.

Linking

The link editor, *ld*, combines one or more object files and libraries (in the order specified) into one executable file, performing relocation, external symbol resolutions, and all other required processing. Unless directed otherwise, the link editor names the executable file *a.out*.

This section summarizes the functions of the link editor. Also described here are how to link−edit a program manually (without using a compiler driver) and how to compile multilanguage programs. Refer to the *ld*(1) manual page for complete information on the link editor.

Invoking the Link Editor Manually

Usually the link editor is invoked by the compiler driver as the final step in compilation (as explained in **"Compiler Drivers"**). If you have object files produced by previous compilations that you want to link together, you can invoke the link editor using a compiler driver instead of calling *ld* directly; just pass the object−file names to the compiler driver in place of source−file names. If the original source files were in a single language, simply invoke the associated driver and specify the list of object files. (For information about linking together objects derived from several languages, see **"Linking Multilanguage Programs."**)

A few command−line options to *ld*, such as *-p*, have different meanings when used as command−line options to *cc*. To pass such options to *ld* through an invocation of a compiler driver, use the *-Wl* option to the driver (see the manual page for details).

Circumstances exist under which you need to invoke *ld* directly, such as when you're doing special linking not supported by compiler drivers (such as building an embedded system). But most of the time it's easier to call a compiler driver and let it invoke *ld* as necessary. A summary of *ld* syntax is provided here in case you need it.

Syntax

```
ld options   object1 [object2...objectn]
```

options	One or more of the options listed in **Table 2–4** .
object	Specifies the name of the object file to be link–edited.

Table 2–4 contains only a partial list of link editor options. Many options that apply only to creating shared objects are discussed in the next chapter. For complete information on options and libraries that affect link editor processing, refer to the *ld*(1) manual page.

Table 2–4 Link Editor Options

Option	**Purpose**
-kllibname	Similar to *-llibname*, but the library is a ucode library named *lib*libname*.b*.
-llibname	Specifies the name of a library, where *libname* is the library name. The link editor searches for a *lib*libname*.so* (and then *lib*libname*.a*) first in any directories specified by *-L dirname* options, and then in the standard directories: */lib*, */usr/lib*, and */usr/local/lib*.
-L *dirname*	Adds *dirname* to the list of directories to be searched for along with libraries specified by subsequent *-llibname* options.
––m	Produces a link editor memory map, listing input and output sections of the code, in System V format.
-M	Produces a link map in BSD format, listing the names of files to be loaded.
-nostdlib	This option must be accompanied by the *-L dirname* option. If the link editor does not find the library in *dirname*, then it does not search any of the standard library directories.
-o *filename*	Specifies a name for your executable. If you do not specify *filename*, the link editor names the executable *a.out*.
-s	Strips symbol table information from the program object, reducing its size. This option is useful for linking routines that are frequently linked into other program objects.
-v	Prints the name of each file as it is processed by the link editor.
-Xsortbss	Sorts bss symbols (this is the default in C but not in Fortran).
-Xnobsschange	Overrides defaults, eliminating all global bss reordering.
-y*symname*	Reports all references to, and definitions of, the symbol *symname*. Useful for locating references to undefined symbols.

Example

The following command tells the linker to search for the DSO *libcurses.so* in the directory */lib*. If it does not find that DSO, the linker then looks for *libcurses.a* in */lib*; then for *libcurses.so* in */usr/lib*, then in the same directory for *libcurses.a*. If it hasn't found an appropriate library by then, it looks in */usr/local/lib* for *libcurses.a*. (Note that the linker does not look for DSOs in */usr/local/lib*, so don't put shared objects there.) If found in any of those places, the DSO or library is linked with the objects *foiled.o* and *again.o*:

```
ld foiled.o again.o –lcurses
```

Note: The *-G* option, which formerly allowed you to specify which data items should be stored in the global pointer region, is no longer useful. *-KPIC*, the default, implies *-G 0*, and the compiler ignores any user attempts to specify otherwise. Compiling *-non_shared* (to avoid *-KPIC*) is primarily useful only for creating device drivers, in which case there is no direct linking step in which to specify a *-G* number. For more information, see the *cc* and *ld* manual pages.

Linking Assembly Language Programs

The assembler driver *as1* does not run the link editor. To link–edit a program written in assembly language, use one of these procedures:

- Assemble and link–edit using one of the other driver commands (*cc*, for example). The *.s* suffix of the assembly language source file causes the driver to invoke the assembler.
- Assemble the file using *as*; then link–edit the resulting object file with the *ld* command.

Specifying Libraries

The link editor *ld* processes its arguments from left to right as they appear on the command line. Arguments to *ld* can be DSOs, object files, or libraries.

When *ld* reads a DSO, it adds all the symbols from that DSO to a cumulative symbol table. If it encounters a symbol that's already in the symbol table, it does not change the symbol table entry. If you define the same symbol in more than one DSO, only the first definition is used.

When *ld* reads an archive, usually denoted by a file name ending in *.a*, it uses only the object files from that archive that can resolve currently unresolved symbol references. (When a symbol is referred to but not defined in any of the object files that have been loaded so far, it's called unresolved.) Once a library has been searched in this way, it is never searched again. Therefore, libraries should come after object files on the command line in order to resolve as many references as possible. Note that if a symbol is already in the cumulative symbol table from having been encountered in a DSO, its definition in any subsequent library is ignored.

Libraries and DSOs can be specified either by explicitly stating a pathname or by use of the library search rules. To specify a library or DSO by path, simply include that path on the command line (relative to the current directory, or else absolute):

```
ld myprog.o /usr/lib/libc.so.1 mylib.so
```

Note: *libc.so.1* is the name of the standard C DSO, replacing the older *libc.a*. Similarly, *libX11.so.1* is the X11 DSO. Most other DSOs are simply named *name.so*, without a *.1* extension.

To use the linker's library search rules, specify the library with the *-llibname* option:

```
ld myprog.o –lmylib
```

When the *-lmylib* argument is processed, *ld* searches for a file called *libmylib.so*. If it can't find *libmylib.so* in a given directory, it tries to find *libmylib.a* there; if it can't find that either, it moves on to the next directory in its search order. The default search order is to look first in */lib*, then in */usr/lib*. After looking in both of those directories, *ld* looks in */usr/local/lib* for archives only (DSOs should not be installed in */usr/local/lib*). You can modify these defaults by specifying the *-L dir* and/or *-nostdlib* options. Directories specified by *-L dir* before the *-llibname* argument are searched in the order they appear on the command line, before the default directories are searched. If *-nostdlib* is specified, then *-L dir* must also be specified because the default directories aren't searched at all.

If *ld* is invoked from one of the compiler drivers, all *-L* and *-nostdlib* options are moved up on the command line so that they appear before any *-llibname* option. For example:

```
cc file1.o –lm –L mydir
```

invokes, at the linking stage of compilation:

```
ld –L mydir file1.o –lm
```

Note: There are three different kinds of files that contain object code files: non–shared libraries, PIC archives, and DSOs. Non–shared libraries are the old–fashioned kind of library, built using *ar* from *.o* files that were compiled with *-non_shared*. These archives must also be linked *-non_shared*. PIC archives are the default in IRIX 5.0, built using *ar* from *.o* files compiled with *-KPIC* (a default option); they can be linked with other PIC files. DSOs are built from PIC *.o* files by using *ld -shared*; see **Chapter 3** for details.

When compiling multilanguage programs, be sure to specify any required runtime libraries using the *-l libname* option. For a list of the libraries that a language uses, see the corresponding compiler driver manual page.

If the link editor tells you that a reference to a certain function is unresolved, check that function's manual page to find out which library the function is in. If it isn't in one of the standard libraries (which *ld* links in by default), you may need to specify the appropriate library on the command line. For an alternative method of finding out where a function is defined, see **"Finding a symbol in an unknown library"**.

Note: Simply including the header file associated with a library routine is not enough; you also must specify the library itself when linking (unless it's a standard library). There is no magical connection between header files and libraries; header files only give prototypes for library routines, not the library code itself.

Examples

To link a sample program *foo.c* with the math DSO, *libm.so*, enter:

```
cc foo.c -lm
```

To specify the appropriate DSOs for a graphics program *foogl.c*, enter:

```
cc foogl.c -lgl -lX11
```

Linking to Dynamic Shared Objects

This section describes how to link your source files with previously built DSOs; for more information about how to build your own DSOs, see **Chapter 3, "Dynamic Shared Objects."**

Note: DSOs replace the older static shared libraries, which were named with the extension *_s.a*. The *_s.a* libraries are no longer shipped with IRIX; however, the runtime versions of those libraries, named with *_s* at the end (and no *.a*), are still present under IRIX 5.0 for backward compatibility with older executables that used static shared libraries.

To build an executable that uses a DSO, call a compiler driver just as you would for a non–shared library. For instance,

```
cc needle.c -lthread
```

links the resulting object file (*needle.o*) with the previously built DSO *libthread.so* (and the standard C DSO, *libc.so.1*), if available. If no *libthread.so* exists, but a PIC archive named *libthread.a* exists, that archive is used with *libc.so.1*, and you still get dynamic (runtime) linking. Note that even *.a* libraries now contain position–independent code by default, though it is also possible to build non–shared *.a* libraries that do not contain PIC.

Linking Multilanguage Programs

When the source language of the main program differs from that of a subprogram, use the following steps to link (refer to **Figure 2–2**):

1. Compile object files from the source files of each language separately by using the *-c* option.

   For example, if the source consists of a Fortran main program (*main.f*) and two files of C functions (*more.c* and *rest.c*), use the commands:

   ```
   cc -c more.c rest.c
   f77 -c main.f
   ```

   These commands produce the object files *main.o, more.o,* and *rest.o*.

2. Use the driver associated with the language of the main program to link the objects together:

   ```
   f77 main.o more.o rest.o
   ```

Note: The compiler drivers supply the default set of libraries necessary to produce an executable from the source of the associated language. However, when producing executables from source code in several languages, you may need to explicitly specify the default libraries for one or more of the languages used. For instructions on specifying libraries, see **"Specifying Libraries."**

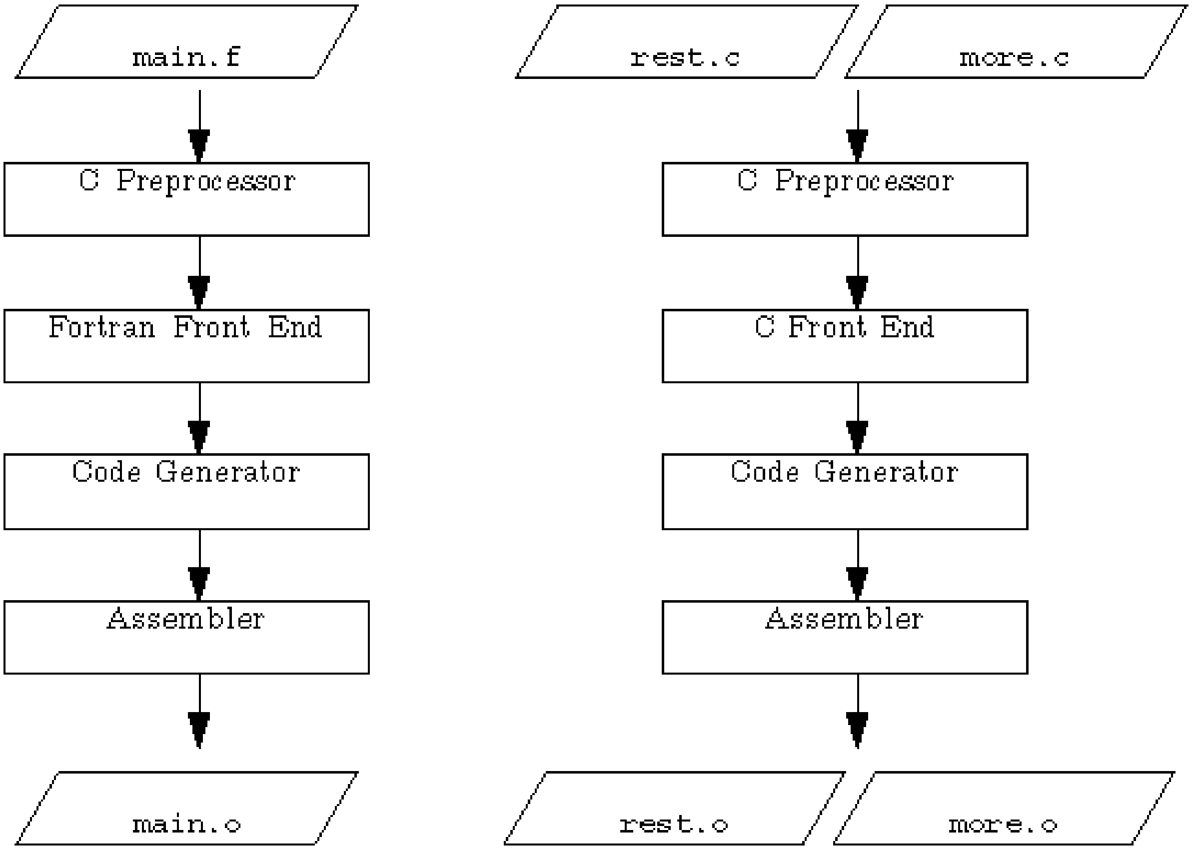

Figure 2–2 Compilation Control Flow for Multilanguage Programs

For specific details about compiling multilanguage programs, refer to the programming guides for the appropriate languages.

Debugging

The compiler system provides a debugging tool, *dbx*, which is explained in the *dbx User's Guide*. In addition, CASEVision/WorkShop™ contains debugging tools. For information about obtaining WorkShop for your computer, contact your dealer or sales representative.

Before using a debugging tool, you must use one of the standard driver options, listed in **Table 2–5** , to produce executables containing information that the debugger can use.

Table 2–5 Driver Options for Debugging

Option	Purpose
-g0	(Default) Produces a program object without debugging information. Reduces the size of the program object but retains optimizations. Use this option after you have finished debugging.
-g1	Specifies accurate, but limited, source–level debugging. This option performs most optimizations.
-g or -g2	Specifies full source–level debugging. These options suppress optimizations that might interfere with full debugging.
-g3	Specifies full, but inaccurate, debugging on fully optimized code. This level of debugger output can be confusing or misleading. Specify this option only for programs that malfunction after you optimize them.

Getting Information About Object Files

The following tools provide information on object files as indicated:

- *elfdump* lists the contents (including the symbol table and header information) of an ELF−format object file.
- *file* provides descriptive information on the general properties of the specified file.
- *nm* lists symbol table information.
- *odump* lists the contents of a COFF−format object file.
- *size* prints the size of each section of an object file (some such sections are named *text*, *data*, and *sbss*). The *a.out*(4) manual page describes the format of these sections.

Listing Selected Parts of Object Files and Libraries With *elfdump*

The *elfdump* tool lists headers, tables, and other selected parts of an ELF−format object file or archive file.

Syntax

```
elfdump options   filename1   [filename2..filenamen]
```

options	One or more of the options listed in **Table 2−6** .
filename	Specifies the name of one or more object files whose contents are to be dumped.

For more information, see the *elfdump*(1) manual page.

Table 2−6 *Elfdump* Options

Option	Dumps
-cr	Compact relocation information.
-Dc	Conflict list in Dynamic Shared Objects.
-Dg	Global Offset Table in Dynamic Shared Objects.
-Dl	Library list in Dynamic Shared Objects.
-Dt	String table entries of the dynamic symbol table in Dynamic Shared Objects.
-f	The file header.
-h	All section headers in the file.
-hash	Hash table entries.
-L	Dynamic section in Dynamic Shared Objects.
-o	Program header.
-r	Relocation information.
-reg	Register info.
-t	Symbol table entries.

Determining File Type With *file*

The *file* tool lists the properties of program source, text, object, and other files. This tool attempts to identify the contents of files using various heuristics. It is not exact and is occasionally fooled. For example, it often erroneously recognizes command files as C programs. For more information, see the *file*(1) manual page.

Syntax

```
file filename1   [filename2..filenamen]
```

where each *filename* is the name of a file to be examined.

Example

Information given by *file* is self−explanatory for most kinds of files, but using *file* on object files and executables gives somewhat cryptic output. In this example, "MSB" indicates Most Significant Byte, also called Big−Endian; "dynamic executable" means the executable was linked with DSO libraries; and "(not stripped)" means the executable contains at least some symbol table information. "Dynamic lib" indicates a DSO.

```
file test.o a.out /lib/libc.so.1
test.o:        ELF 32-bit MSB relocatable MIPS - version 1
a.out:         ELF 32-bit MSB dynamic executable (not stripped) MIPS -
 version 1
/lib/libc.so.1:   ELF 32-bit MSB dynamic lib MIPS - version 1
```

Listing Symbol Table Information: *nm*

The *nm* tool lists symbol table information for object files and archive files.

Syntax

`nm options   filename1 [filename2..filenamen]`

options	One or more of the options listed in **Table 2-7** .
filename	Specifies the object files or archive files from which symbol table information is to be extracted. If you do not specify a file name, *nm* assumes the file is called *a.out*.

Table 2-7 Symbol Table Dump Options

Option	Purpose
-a	Prints debugging information. If used with *-B*, uses BSD ordering with System V formatting.
-A	Prints the listing in System V format (default).
-b	Prints the value field in octal.
-B	Prints the listing in BSD format.
-d	Prints the value field in decimal (the default for System V output).
-e	Prints only external and static variables.
-h	Suppresses printing of headers.
-n	Sorts external symbols by name for System V format. Sorts all symbols by value for Berkeley format (by name is the BSD default output).
-o	Prints value field in octal (System V output). Prints the file name immediately before each symbol name (BSD output).
-p	Lists symbols in the order they appear in the symbol table.
-r	Reverses the sort that you specified for external symbols with the *-n* and *-v* options.
-T	Truncates characters in exceedingly long symbol names; inserts an asterisk as the last character of the truncated name. This option may make the listing easier to read.
-u	Prints only undefined symbols.
-v	Sorts external symbols by value (default for Berkeley format).
-V	Prints the version number of *nm*.
-x	Prints the value field in hexadecimal.

Table 2-8 defines the one-character codes shown in an *nm* listing. Refer to the example that follows the table for a sample listing.

Table 2-8 Character Code Meanings

Key	Description
a	Local absolute data
A	External absolute data
b	Local zeroed data
B	External zeroed data
C	Common data
d	Local initialized data
D	External initialized data
E	Small common data
G	External small initialized data
N	Nil storage class (avoids loading of unused external references)
r	Local read-only data
R	External read-only data
s	Local small zeroed data
S	External small zeroed data
t	Local text

T	External text
U	External undefined data
V	External small undefined data

Example

This example demonstrates how to obtain a symbol table listing. Consider the following program, *tnm.c*:

```
#include <stdio.h>
#include <math.h>
#define LIMIT 12
int unused_item = 14;
double mydata[LIMIT];

main()
{
      int i;
      for(i = 0; i < LIMIT; i++) {
          mydata[i] = sqrt((double)i);
      }
      return 0;
}
```

Compile the program into an object file by entering:

```
cc -c tnm.c
```

To obtain symbol table information for the object file *tnm.o* in BSD format, use the *nm -B* command:

```
nm -B tnm.o

0000000000 T main
0000000000 B mydata
0000000000 U sqrt
0000000000 D unused_item
00000000 N _bufendtab
```

To obtain symbol table information for the object file *tnm.o* in System V format use the *nm* command without any options:

```
nm tnm.o
Symbols from tnm.o:

[Index]   Value     Size    Class    Type         Section    Name

[0]     |        0|        |File    |ref=4       |Text      | tnm.c
[1]     |        0|        |Proc    |end=3  int  |Text      | main
[2]     |      116|        |End     |ref=1       |Text      | main
[3]     |        0|        |End     |ref=0       |Text      | tnm.c
[4]     |        0|        |File    |ref=6       |Text      |
 /usr/include/math.h
[5]     |        0|        |End     |ref=4       |Text      |
 /usr/include/math.h
[6]     |        0|        |Global  |            |Data      | unused_item
[7]     |        0|        |Global  |            |Bss       | mydata
[8]     |        0|        |Proc    |ref=1       |Text      | main
[9]     |        0|        |Proc    |            |Undefined| sqrt
[10]    |        0|        |Global  |            |Undefined| _gp_disp
```

Finding a symbol in an unknown library

When *ld* indicates that a symbol is undefined, you can use *nm* to figure out which DSO or library needs to be linked in by piping *nm*'s output through appropriate *grep*s.

Example

You're trying to compile a program, and *ld* tells you that you're trying to use an undefined symbol:

```
cc prog.c −lgl
ld:
Unresolved:
XGetPixel
```

But you don't know where XGetPixel is defined. Use *nm* to list the symbol tables for all of the available DSOs, and filter that output to find only the places where XGetPixel is mentioned. Then filter the result to find the places where XGetPixel is actually defined, as indicated by the T character code.

```
nm −Bo /usr/lib/lib*.so* | grep XGetPixel | grep T
/usr/lib/libX11.so.1: 0f790ff8 T XGetPixel
```

Some DSOs end in *.so*, while others end in *.so.1*, so we need to use multiple wildcards to get all of them. Also, this command line has to be modified to look in PIC archives or non−shared libraries; as written it only looks in DSOs.Now that XGetPixel is defined in */usr/lib/libX11.so.1*, the X11 DSO; use the *-l* option to tell *cc* to link in that library, and *ld* won't complain any more.

```
cc prog.c −lgl −lX11
```

Listing Selected Parts of COFF Files With *odump*

The *odump* tool lists headers, tables, and other selected parts of a COFF−format object or archive file. It is provided with this release of IRIX for compatibility; use *elfdump* for ELF−format files.

Syntax

```
odump options   filename1   [filename2..filenamen]
```

options	One or more of the options listed in **Table 2−9** .
filename	Specifies the name of one or more object files whose contents are to be dumped.

For more information, see the *odump*(1) manual page.

Table 2−9 *Odump* Options

Option	Dumps
-a	Archive header of each object file in the specified archive library file.
-c	String table.
-d *number*	The section numbered *number*, or a range of sections starting with *number* and ending with the last section number available (or the number you specify with the *+d* auxiliary option).
+d *number*	All sections starting with the first section (or with the section specified with the *-d* option) and ending with the section numbered *number*.
-f	File header for each object file in the specified file.
-F	File descriptor table for each object file in the specified file.
-g	Global symbols in the symbol table of an archive library file.
-h	Section headers.
-i	Symbolic information header.
-l	Line number information.
-n name	Information for section named *name* only. Use this option with the *-h*, *-s*, *-r*, *-l*, or *-t* option.
-o	Optional header for each object file.
-p	Suppresses the printing of headers.
-P	Procedure descriptor table.
-r	Relocation information.
-R	Relative file index table.
-s	Section contents.

-t	Symbol table entries.
-t index	Only the indexed symbol table entry. Use the *+t* option with the *-t* option to specify a range of table entries.
+t index	Symbol table entries in a range that ends with the indexed entry. The range begins with the first symbol table entry or with the section that you specify with the *-t* option.
-v	Information in symbolic rather than numeric representation. This option may be used with any *odump* option except *-s*.
-z name, number	Line number entry (or a range of entries starting at the specified number) for the named function.
+z number	Line number entries starting with the function name or line number specified by the *-z* option and ending with *number*.

Determining Section Sizes With *size*

The *size* tool prints information about the sections (such as *text*, *rdata*, and *sbss*) of the specified object or archive files.

Syntax

```
size options   [filename1   filename2..filenamen]
```

options	Specifies the format of the listing (see **Table 2–10**).
filename	Specifies the object or archive files whose properties are to be listed. If you do not specify a file name, the default is *a.out*.

Table 2–10 *Size* Options

Option	Action
-A	Prints data section headers in System V format.
-B	Prints data section headers in Berkeley format.
-d	Prints sizes in decimal (default).
-F	Prints data on loadable segments.
-n	Prints symbol table, global pointer, and more.
-o	Prints sizes in octal.
-s	Follows shared libraries, adding them as they're encountered to the list of files to be *size*d.
-V	Prints the version of size that you are using.
-x	Prints sizes in hexadecimal.

Example

Below are examples of the *size* command and the listings they produce:

```
size –B –o test.o

          text   data  bss    rdata  sdata  sbss  decimal  hex
test.o  31250  2010  40470  550    210    50    31232    7a00
```

```
size –B –d test.o

          text   data  bss    rdata  sdata sbss  decimal  hex
test.o  12968  1032  16696  360    136   40    31232    7a00
```

Using the Archiver to Create Libraries

An archive library is a file that includes the contents of one or more object (*.o*) files. When the link editor (*ld*) searches for a symbol in an archive library, it loads only the code from the object file where that symbol was defined (not the entire library) and links it with the calling program.

The archiver (*ar*) creates and maintains archive libraries and has the following main functions:

- Copying new objects into the library
- Replacing existing objects in the library
- Moving objects around within the library
- Extracting individual objects from the library

The following section explains the syntax of the *ar* command and lists some examples of how to use it. See the *ar*(1) manual page for details.

Note: *ar* simply strings together whatever object files you tell it to archive; thus, it can be used to build either non−shared or PIC libraries, depending on how the included *.o* files were built in the first place. If you do create a non−shared library with *ar*, remember to link it *-non_shared* with your other code. For information about building DSOs and converting libraries to DSOs, see **Chapter 3** .

Syntax

```
ar options [posObject] libName [object1...objectn]
```

options	Specifies the action that the archiver is to take. **Table 2−11** , **Table 2−12** , and **Table 2−13** list the available options. To specify more than one option, don't use a dash or put spaces between the options. For example, use *ar ts*, not *ar -t -s*.
posObject	Specifies the name of an object within an archive library. It specifies the relative placement (either before or after *posObject*) of an object that is to be copied into the library or moved within the library. This parameter is required when the *a*, *b*, or *i* suboptions are specified with the *m* or *r* option. The last example in **"Examples,"** shows the use of a *posObject* parameter.
libName	Specifies the name of the archive library you are creating, updating, or extracting information from.
object	Specifies the name(s) of the object file(s) to manipulate.

Archiver Options

When running the archiver, specify exactly one of the options *d*, *m*, *p*, *q*, *r*, *t*, or *x* (listed in **Table 2−11**). In addition, you can optionally specify any of the modifiers in **Table 2−12** , as well as any of the archiver suboptions listed in **Table 2−13** .

Table 2−11 Archiver Options

Option	Purpose
d	Deletes the specified objects from the archive.
m	Moves the specified files to the end of the archive. If you want to move the object to a specific position in the archive library, specify an *a*, *b*, or *i* suboption together with a *posObject* parameter.
p	Prints the specified objects in the archive on the standard output device (usually the terminal screen).
q	Adds the specified object files to the end of the archive. This option is similar to the *r* option (described below), but is faster and does not remove any older versions of the object files that may already be in the archive. Use the *q* option when creating a new library.
r	Adds the specified object files to the end of the archive file. If an object file with the same name already exists in the archive, the new object file overwrites it. If you want to add an object at a specific position in the archive library, specify an *a*, *b*, or *i* suboption together with a *posObject* parameter. Use the *r* option when updating existing libraries.
t	Prints a table of contents on the standard output (usually the screen) for the specified object or archive file.
x	Copies the specified objects from the archive and places them in the current directory. Duplicate files are overwritten. The last modified date is the current date (unless you specify the *o* suboption, in which case the date stamp on the archive file is the last modified date). If no objects are specified, x copies all the library objects into the current directory.

Table 2−12 Archiver Modifiers

Option	Purpose
c	Suppresses the warning message that the archiver issues when it discovers that the archive you specified does not already exist.
C	Makes an archive compatible with pre−SVR4 IRIX.
E	The default; creates an archive matching the specifications given by the SVR4 ABI.
l	Puts the archiver's temporary files in the current working directory. Ordinarily, the archiver puts those files in *%/tmp* (unless the STMDIR environment variable is set, in which case *ar* stores temporary files in the directory indicated by that variable). This option is useful when */tmp* (or STMDIR) is full.
s	Creates a symbol table in the archive. This modifier is rarely necessary since the archiver updates the symbol table of the archive library automatically. Options *m*, *p*, *q*, and *r*, in particular, create a symbol table by default and thus do not require *s* to be specified.
v	Lists descriptive information during the process of creating or modifying the archive. When specified with the *t* option, produces a verbose table of contents.

Table 2−13 Archiver Suboptions

Suboption	Use with Option	Purpose
a	*m* or *r*	Specifies that the object file being added should follow the already−archived object file specified by the *posObject* parameter on the command line.
b	*m* or *r*	Specifies that the object file precede the object file specified by the *posObject* parameter.
i	*m* or *r*	Same as *b*.
o	*x*	Forces the last modified date of the extracted object file to match that of the archive file.
u	*r*	Tells the archiver not to replace the existing object file in the archive if the last modified date indicates that the object file already in the archive is newer (more recently modified) than the one you're adding.

Note: The *a* and *b* suboptions are only useful if the same symbol is defined in two or more of the object files in the archive (in which case, the symbol table shows the first definition listed in the archive). Under other circumstances, order of object files in an archive is irrelevant (and the *a* and *b* suboptions are useless), since *ld* uses the archive symbol table rather than searching linearly through the file.

Examples

Create a new library, *libtest.a*, and add object files to it by entering:

```
ar cq libtest.a mcount.o mon1.o string.o
```

The *c* option suppresses an archiver message during the creation process. The *q* option creates the library and puts *mcount.o*, *mon1.o*, and *string.o* into it.

An example of replacing an object file in an existing library:

```
ar r libtest.a mon1.o
```

The *r* option replaces *mon1.o* in the library *libtest.a*. If *mon1.o* does not already exist in the library *libtest.a* , it is added.

Note: If you specify the same file twice in an argument list of files to be added to an archive, that file appears twice in the archive.

To add a new file immediately before *mcount.o* in this library, enter:

```
ar rb mcount.o libtest.a new.o
```

The *r* option adds *new.o* to the library *libtest.a*. The *b* option followed by *mcount.o* as the *posObject* causes the archiver to place *new.o* immediately before *mcount.o* in the archive.

Chapter 3

Dynamic Shared Objects

A dynamic shared object (DSO) is an object file that's meant to be used simultaneously—or *shared*—by multiple applications (*a.out* files) while they're executing. DSOs can be used in place of archive libraries, and they replace static shared libraries provided with earlier releases of IRIX.

Since DSOs contain shared components, using them provides several substantial benefits. First, overall memory usage is minimized because code is shared. Two executables that use the same DSO and that run simultaneously will have only one copy of the shared components loaded into memory. For example, if executable A and executable B both link with the same DSO C, and if A and B are both running at the same time, the total memory used is what's required for A, B, and C, plus some small overhead. If C was an unshared library, the memory used would be what's required for A, B, and two copies of C. A related second benefit is that executables linked with DSOs are smaller than those linked with unshared libraries because the shared objects aren't part of the executable file image, so disk usage is minimized.

Yet another feature of DSOs is that they are much easier to use, build, and debug than static shared libraries. Most of the libraries supplied by Silicon Graphics are available as DSOs. In the past, only a few static shared libraries have been available; most libraries were unshared.

Additionally, executables that use a DSO don't have to be relinked if the DSO changes; when the new DSO is installed, the executable automatically starts using it. This feature makes it easier to update end users with new software versions. It also allows you to create hardware−independent software packages more easily. You can design the hardware−dependent routines required by your application so that they have the same interface across all platforms. Then, you create different DSOs for each of the platforms, each DSO containing the implementation of those routines for that particular platform. The shrink−wrapped software package can then contain all the DSOs and is able to run on all the platforms.

Finally, DSOs and the executables that use them are mapped into memory by a run−time loader, *rld*, which resolves external references between objects and relocates objects at run time. (DSOs contain only position−independent code (PIC), so they can be loaded at any virtual address at run time.) With *rld*, the binding of symbols can be changed at run time at the request of the executing program. You could use this feature to dynamically change the feature set presented to a user of your application, for example, while minimizing start−up time. The application could be started quickly, with a subset of the features available and then, if the user needs other features, those can be loaded in under programmatic control.

Naturally, some costs are involved with using DSOs, and these are explained in the next section, **"Using DSOs."** The sections after that explain how to build and optimize DSOs and how *rld* works. The *dso*(5) reference page also contains more information about DSOs.

- **"Using DSOs"** tells you how to obtain the most benefit from using DSOs when creating your executable.
- **"Taking Advantage of QuickStart"** discusses an optimization you can use to make sure that DSOs you build load as quickly as possible.
- **"Building DSOs"** describes how to build a DSO.
- **"Runtime Linking"** discusses the runtime linker, and how it locates DSOs at runtime.
- **"Dynamic Loading Under Program Control"** explains the use of the *libdl* library to control runtime linking.
- **"Versioning of DSOs"** discusses a versioning mechanism for DSOs that allows binaries linked against different, incompatible versions of the same DSO to run correctly.

Using DSOs

Using DSOs is easy—the syntax is the same as for an archive (*.a*) library. The following compile line creates the executable *yourApp* by linking with the DSOs *libyours.so* and with *libc.so.1*:

```
cc yourApp.c –o yourApp –lyours
```

If *libyours.so* isn't available, but the archive version *libyours.a* is available, that archive version is used along with *libc.so.1*.

You should note that there's a significant difference between DSOs and archive libraries in terms of what gets loaded into memory when an application is executing. With an archive library, only the text portion of the library that the application actually requires (and the data associated with that text) gets loaded, not the entire library. In contrast, the entire DSO that's linked gets loaded. Thus, to conserve memory, don't link with DSOs unless your application actually needs them.

Also, you should avoid listing any archive libraries on the compile line after you list shared libraries; instead, list the archive libraries first and then the DSOs.

You may want to take advantage of the QuickStart optimization that minimizes start–up times for executables. You can use QuickStart when using or building DSOs. At link time, when an executable or a DSO is being created, the link editor *ld* assigns initial addresses to the object and attempts to resolve all references. Since DSOs are relocatable, these initial address assignments are really only guesses about where the object will be really loaded. At run time, *rld* verifies that the DSO being used is the same one that was linked with and what the real addresses are. If the DSOs are the same and if the addresses match the initial assignments, *rld* doesn't have to perform any relocation work, and the application starts up quickly (or QuickStarts). When an application QuickStarts, memory use is smaller since *rld* doesn't have to read in the information necessary to perform relocations.

To determine whether your application (or DSO) is able to do a QuickStart, use the *-quickstart_info* flag when building the executable (or DSO). If the application or DSO can't do a QuickStart, you'll be given information about how what to do. The next section goes into more detail about why an executable may not be able to use QuickStart.

In summary, when you use DSOs to build an executable:

- Link with only the DSOs that you need.
- Make sure that unshared libraries precede DSOs on the compile line.
- Use the *-quickstart_info* flag.

Taking Advantage of QuickStart

QuickStart is an optimization designed to reduce start–up times for applications that link with DSOs. Each time *ld* builds a DSO, it updates a registry of shared objects. The registry contains the preassigned QuickStart addresses of a group of DSOs that typically cooperate by having nonoverlapping locations. (See **"Using Registry Files"** for more information about how to use the registry when you're building a DSO.) If you compile your application by linking with registered DSOs, your application takes advantage of QuickStart: all the DSOs are mapped at their QuickStart addresses, and *rld* won't need to move any of them to an unused address and perform a relocation pass to resolve all references.

Suppose you compile your application using the *-quickstart_info* flag, and it fails. QuickStart may fail because:

- Your application has directly or indirectly linked with two different versions of the same DSO, as shown in **Figure 3–1** . In this example, *yourApp* links with *libyours.so*, *libmotif.so*, and *libc.so.1* on the compile line. When the DSO *libyours.so* was built, however, it linked with *libmalloc.so*, which in turn linked with *libc.so.1* when it was created. If the two versions of *libc.so.1* aren't identical, *yourApp* won't be able to QuickStart.

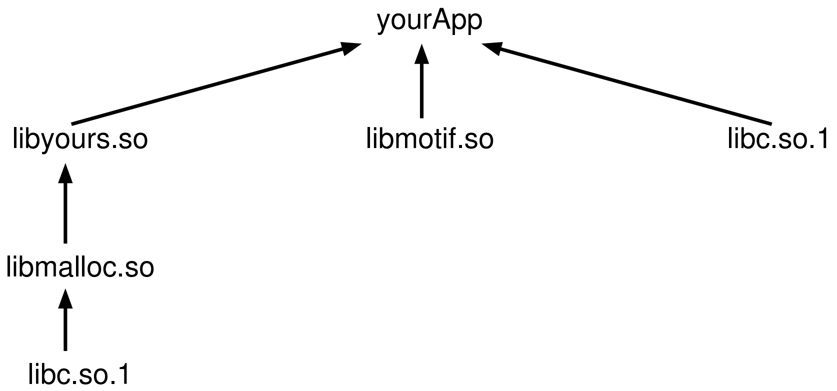

Figure 3–1 An Application Linked with DSOs

- You link with a DSO that can't QuickStart. This may occur because the DSO wasn't registered and therefore was assigned a location that overlaps with the location assigned another DSO.
- Your application pulls in incompatible shared objects (in a manner similar to the example shown in **Figure 3–1**).
- Your application contains an unresolved reference to a function (where it takes the address of the function).
- The DSO links with another DSO that can't QuickStart.

Even if QuickStart officially succeeds, your application may have name space collisions and therefore may not start up as fast as it should. This is because *rld* has to bring in more information to resolve the conflicts. In general, you should avoid having conflicts both because of the detrimental effect on start–up time and because conflicts make it difficult to ensure the correctness of an application over time.

In the example shown in **Figure 3–1** , you may have written your own functions to allocate memory in *libmalloc.so* for *libyours.so* to use. If you didn't use unique names for those functions (instead of *malloc()*, for example) the way this particular compile and link hierarchy is set up, the standard *malloc()* function defined in *libc.so.1* is used instead of the one defined in *libmalloc.so*. (Conflicts are resolved by proceeding through the hierarchy from left to right and then moving to the next level. See **"Searching for DSOs at Runtime"** for more information about how the runtime linker searches for DSOs.)

Thus, it's not a good idea to allow more than one DSO to define the same function. Even if the DSOs are synchronized for their first release, one of them may change the definition of the function in a subsequent release. Of course, you can use conflicts to override function definitions intentionally, but you should be sure you have control over what is overriding what over time.

If you use the *–quickstart_info* option, *ld* tells you if conflicts arise. It also tells you to run *elfdump* with the *–Dc* option to find the conflicts. See the *elfdump*(5) man page for more information about how to read the output produced by *elfdump*.

Building DSOs

In most cases, you can build DSOs as easily as archive libraries. If your library is written in a high–level language, such as C or Fortran, you won't have to make any changes to the source code. If your code is in assembly language, you must modify it to produce PIC, as described in **Appendix A, "Position–Independent Coding in Assembly Language."**

To create a DSO from a set of object files, use *ld* with the *-shared* option:

```
ld –shared stuff.o nonsense.o –o libdada.so
```

The above example creates a DSO, *libdada.so*, from two object files, *stuff.o* and *nonsense.o*. Note that DSO names should begin with "*lib*" and end with "*.so*", for ease of use with the compiler driver's *-l*ib argument. If you're already building an archive library (*.a* file), you can create a DSO from the library by using the *-shared* and *-all* arguments to *ld*:

```
ld –shared –all libdada.a –o libdada.so
```

The *-all* argument specifies that all of the object files from the library, *libdada.a*, should be included in the DSO.

Making DSOs Self−Contained

When building a DSO, be sure to include any archives required by the DSO on the link line so that the DSO is self−contained (that is, it has no unresolved symbols). If the DSO depends on libraries not explicitly named on the link line, subsequent changes to any of those libraries may result in name space collisions or other incompatibilities that can prevent any applications that use the DSO from doing a QuickStart. Such incompatibilities can also lead to unpredictable results over time as the libraries change asynchronously. To make the archive *libmine.a* into a DSO, for example, and *libmine.a* depends on routines in another archive, *libutil.a*, include *libutil.a* on the link line:

```
ld −shared −all −no_unresolved libmine.a −o libmine.so −none libutil.a
```

This causes the modules in *libutil.a* that are referenced in *libmine.a* to be included in the DSO, but these modules won't be exported. See **"Controlling Symbols to be Exported or Loaded"** for more information about exported symbols. The *-no_unresolved* option causes a list of unresolved symbols to be created; generally, this list should be empty to enable QuickStarting.

Similarly, if a DSO relies on another DSO, be sure to include that DSO on the link line. For example:

```
ld −shared −all −no_unresolved libbtree.a −o libtree.so −lyours
```

This example places *libyours.so* in the *liblist* of the new DSO, *libtree.so*. This ensures that *libyours.so* is loaded whenever an executable that uses *libtree.so* is launched. Again, symbols from *libyours.so* won't be exported for use by other libraries. (You can use the *-exports* flag to reverse this exporting behavior; the *-hides* flag specifies the default exporting behavior.)

Controlling Symbols to be Exported or Loaded

By default, to help avoid conflicts, symbols defined in an archive or a DSO that's used to build another DSO aren't externally visible. You can explicitly export or hide symbols with the *-exported_symbol* and *-hidden_symbol* options:

```
−exported_symbol name1, name2, name3
−hidden_symbol name4, name5
```

By default, if you explicitly export any symbols, all other symbols are hidden. If you both explicitly export and explicitly hide the same symbol on the link line, the first occurrence determines the behavior. You can also create a file of symbol names (delimited by white space) that you want explicitly exported or hidden, and then refer to the file on the link line with either the *−exports_file* or *−hiddens_file* option:

```
−exports_file yourFile
−hiddens_file anotherFile
```

These files can be used in addition to explicitly naming symbols on the link line.

Another useful option, *-delay_load*, prevents a library from being loaded until it's actually referenced. Suppose, for example, that your DSO contains several functions that are likely to be used in only a few instances. Furthermore, those functions rely on another library (archive or DSO). If you specify *-delay_load* for this other library when you build your DSO, the runtime linker loads that library only when those few functions that require it are used. Note that if you explicitly export any symbols defined in a library that the runtime linker is supposed to delay loading, the export behavior takes precedence and the library is automatically loaded at runtime.

Note: You can build DSOs using *cc*. However, if you want to export symbols/files or use *-delay_load*, use *ld* to build DSOs.

Using DSOs With C++

To make a DSO, build the C++ objects as you would normally:

```
CC -c
```

Then type:

```
CC -shared -o libmylib.so <list your objects here>
```

For example:

```
CC -shared -o libmylib.so a.o b.o c.o
```

In this instance, the *-l* and *-L* options to *ld* will work. However, most *ld* options won't work. If you want to specify other options, first determine the options that you must pass to *ld*. These options include:

```
-init _main
-fini _fini
-hidden_symbol _main
-hidden_symbol _fini
-hidden_symbol __head
-hidden_symbol __endlink
```

Finally, link in */usr/lib/c++init.o*.

Using Registry Files

You can make sure that your DSOs don't conflict with each other by using a QuickStart registry file. The registry files contain location information for shared objects. When creating a shared object, you can specify a registry file to *ld*, and *ld* ensures that your shared object doesn't conflict with any of the shared objects listed in the registry. A registry file containing the locations of all the shared objects provided with the system is supplied in */usr/lib/so_locations*.

You can use two options to *ld* to specify a registry file: *-check_registry* and *-update_registry*. When you invoke *ld* to build a shared object, with the argument *-check_registry file*, *ld* makes sure that the new shared object doesn't conflict with any of the shared objects listed in *file*. When invoked with *-update_registry file*, *ld* checks the registry in the same way, but when it's done, it writes an entry in *file* for the DSO being built. If *file* isn't writable, *-update_registry* acts like *-check_registry*. If *file* isn't readable, both *-update_registry* and *-check_registry* are ignored.

By exchanging registry files, providers of DSOs can avoid collisions between their shared objects. You should probably start out with a copy of */usr/lib/so_locations*, so that your shared objects won't conflict with any of the standard DSOs. However, you should remember that when collisions occur between shared objects, the only effect is slowing program startup.

Registry File Format

Three types of lines in the registry file include:

- comment lines, which begin with a pound sign (#)
- directive lines, which begin with a dollar sign ($)
- shared object specification lines, which begin with the name of a shared object

Comment lines are ignored by *ld*. Directive lines and shared object specification lines are described below.

Directive Lines

Directive lines specify global parameters that apply to all the DSOs listed in the registry.

```
$text_align_size=align padding=pad-size
$data_align_size=align padding=pad-size
```

These two directives specify the alignment and padding requirements for text and data segments, respectively. The current default segment alignment is 64K, which is the minimum permissible. The size value of a segment of a DSO appearing in the registry file is calculated based on the actual section size plus padding, and is aligned to the section align size (either the default or the one specified by the above directive). The align values for text and data as well as the padding values must be aligned to the minimum alignment size (64K). If not, *ld* generates a warning message and aligns these values to the minimum alignment.

```
$start_address=addr
```

This directive specifies where to start looking for addresses to put shared objects. The default start_address is 0x6000000.

```
$data_after_text={ 1 | 0 }
```

In this directive, a value of one instructs the linker to place data immediately after the text at specified text and data alignment requirements. A value of zero (the default) allows the linker to place these segments in different portions of the address space.

Shared Object Specification Lines

Shared object specification lines have the format:

```
so_name [ :st = {.text | .data | $range} base_addr,padded_size : ] *
```

where:

so_name	full path name (or trailing component) of a shared object
:st =	literal string indicating the beginning of the segment description
.text, .data	segment types: text or data
$range	range of addresses that can be used
base_addr	address where the segment starts
padded_size	padded size of the segment
:	literal string indicating the end of the segment description

A shared object specification can span several lines by "escaping" the newline character (using "\" as the last character on the line that is being continued). The following is an example of a shared object specification line:

```
libc.so.1 \
              :st = $range 0x5fc00000, 0x00400000:\
              :st = .text 0x5fe40000, 0x000a0000:\
              :st = .data 0x5fee0000, 0x00030000:
```

This specification instructs *ld* to relocate all segments of *libc.so.1* in the range 0x5fc00000 to 0x5fc00000+0x0040000, and, if possible, to place the text segment at 0x5fe40000 and the data segment at 0x5fee0000. The text segment should be padded to 0xa0000 bytes and the data segment to 0x3000 bytes. See */usr/lib/so_locations* for examples of shared object specifications.

When building a DSO with the *-check_registry* or *-update_registry* flag, if an entry corresponding to this DSO exists in the registry file, the linker tries to assign the indicated addresses for text and data. However, if the size of the DSO changes and no longer fits in the specified location, the linker searches for another location that fits. If the *$range* option is specified, the linker places the DSO only in the specified range of addresses. If there isn't enough room, an error is returned.

Runtime Linking

This section explains the search path followed by the run–time linker and how you can cause symbols to be resolved at run time rather than link time.

Searching for DSOs at Runtime

When you run a dynamically linked executable, the runtime linker, *rld*, identifies the DSOs required by the executable, loads the required DSOs, and if necessary relocates DSOs within the process's virtual address space, so that no two DSOs occupy the same location.The program header of a dynamically linked executable contains a field, the *liblist*, which lists the DSOs required by the executable.

When looking for a DSO, *rld* searches directories in the following sequence:

1. the path of the DSO in the *liblist* (if an explicit path is given)
2. RPATH if it's defined in the main executable
3. LD_LIBRARY_PATH if defined
4. the default path (*/usr/lib:/lib*)

RPATH is a colon−separated list of directories stored in the main executable. You can set RPATH by using the *-rpath* argument to *ld*:

```
ld −o myprog myprog.c −rpath /d/src/mylib libmylib.so −lc
```

This example links the program against *libmylib.so* in the current directory, and configures the executable such that *rld* searches the directory */d/src/mylib* when searching for DSOs.

The LD_LIBRARY_PATH environment variable is a colon−separated list of directories to search for DSOs. This can be very useful for testing new versions of DSOs before installing them in their final location. You can set the environment variable _RLD_ROOT to a colon−separated list of directories. The runtime linker prepends these to the paths in RPATH and the paths in the default search path.

In all of the colon−separated directory lists, an empty field is interpreted as the current directory. A leading or trailing colon counts as an empty field. Thus, if you set LD_LIBRARY_PATH to:

```
/d/src/lib1:/d/src/lib2:
```

the runtime linker searches the directory */d/src/lib1*, then the directory */d/src/lib2*, and then the current directory.

Note: For security reasons, if an executable has its set−user−ID or set−group−ID bits set, the runtime linker ignores the environment variables LD_LIBRARY_PATH and _RLD_ROOT. However, it still searches the directories in RPATH and the default path.

Runtime Symbol Resolution

Dynamically linked executables can contain symbol references that aren't resolved before run time. Any symbol references in your main program or in an archive must be resolved at link time, unless you specify the *-ignore_unresolved* argument to *cc*. DSOs may contain references that aren't resolved at link time. All data symbols must be resolved at run time. If *rld* finds an unresolvable data symbol at run time, it will cause the executable to exit with an error. Text symbols are resolved only when they're used; so a program can run with unresolved text symbols, as long as the unresolved symbols aren't used.

You can force *rld* to resolve text symbols at run time by setting the environment variable LD_BIND_NOW. If unresolvable text symbols exist in your executable and LD_BIND_NOW is set, the executable will exit with an error, just as if there were unresolvable data symbols.

Compiling with *-Bsymbolic*

<Default Para Font>When you compile a DSO with *-Bsymbolic*, the dynamic linker resolves referenced symbols from itself first. If the shared object fails to supply the referenced symbol, then the dynamic linker searches the executable file and other shared objects. For example:

main—defines *x*
x.so—defines and uses *x*

If you compile *x.so* with *-Bsymbolic* on, the linker tries to resolve the use of *x* by looking first for the definition in *x.so* and then by looking in *main*.

In FORTRAN programs, the linker allocates space for *COMMON* symbols and the compiler allocates space for *BLOCK DATA*. The first kind of symbol (with *COMMON* blocks present) appears in the symbol table as *SHN_MIPS_ACOMMON* (uninitialized *DATA*) whereas the second kind of symbol (with *BLOCK DATA* present) appears as *SHN_DATA* (initialized *DATA*). In general, initialized data takes precedence when the dynamic linker tries to resolve a symbol. However, with *-Bsymbolic*, whatever is defined in the current object takes precedence, whether it is initialized or uninitialized.

Variables that are declared at file scope in C with *-cckr* are also treated this way. For example:

```
int foo[100];
```

is *COMMON* if *-cckr* is used and *DATA* if *-xansi* or *-ansi* is used.

For example:

In *main*:

```
COMMON i, j /* definition of i, j with initial values */
DATA i/1/, j/1/
call junk
end
```

In *x.so*:

```
COMMON i, j
/* definition of i, j with NO initial values */
/* initialized by kernel to all zeros */
print *, i, j
end
```

When you build *x.so* using *-Bsymbolic*, this program prints:

```
0 0
```

When you build *x.so* without *-Bsymbolic*, this program prints:

```
1
```

Converting Libraries to DSOs

When you link a program with a DSO, all of the symbols in the DSO become associated with the executable. This can cause unexpected results if archives that contain unresolved externals are converted to DSOs. When linking with a PIC archive, the linker links in only those object files that satisfy unresolved references. If an object file in an archive contains an unresolved external reference, the linker tries to resolve the reference only when that object file is linked in to your program. In contrast, a DSO containing an external data reference that cannot be resolved at run time causes the program to fail. Therefore, you should exercise caution when converting archives with external data references to DSOs.

For example, suppose you have an archive, *mylib.a*, and one of the object files in the archive, *has_extern.o*, references an external variable, *foo*. As long as your program doesn't reference any symbols in *has_extern.o*, the program will link and run properly. If your program references a symbol in *has_extern.o* and doesn't define *foo*, then the link will fail. However, if you convert *mylib.a* to a DSO, then any program that uses the DSO and doesn't define *foo* will fail at run time, regardless of whether the program references any symbols from *has_extern.o*.

Two possible solutions exist for this problem. The first consists of simply adding a "dummy" definition of the data to the DSO. A data definition appearing in the main executable preempts one appearing in the DSO itself. This may, however, be misleading for executables that use the portion of the DSO that needs the data, but that failed to define it in the main program. The second solution consists of separating the routines that use the data definition into a second DSO, and placing dummy functions for them in the first DSO.

The second DSO can then be dynamically loaded the first time any of the dummy functions is accessed. Each of the dummy functions must verify that the second DSO was loaded before calling the real function (which must have a unique name). This way, programs run whether or not they supply the missing external data, as long as they don't call any of the functions that require the data. The first time one of the dummy functions is called, it tries to dynamically load the second DSO. Programs that do not supply the missing data fail at this point. For more information on dynamic loading, see **"Dynamic Loading Under Program Control"** below.

Dynamic Loading Under Program Control

IRIX provides a library interface to the runtime linker that allows programs to dynamically load and unload DSOs. This interface is called *libdl*, and it consists of four functions listed in **Table 3–1** .

Table 3–1 *libdl functions*

dlopen()	load a DSO
dlsym()	find a symbol in a loaded DSO
dlclose()	unload a DSO
dlerror()	report errors

To load a DSO, call *dlopen()*:

```
include <dlfcn.h>
void *dlhandle;
 ...
dlhandle = dlopen("/usr/lib/mylib.so", RTLD_LAZY);
if (dlhandle == NULL) {
 /* couldn’t open DSO */
 printf("Error: %s\n", dlerror());
}
```

The first argument to *dlopen()* is the pathname of the DSO to be loaded. This may be either an absolute or a relative pathname. When you call this routine, the runtime linker tries to load the specified DSO. If any unresolved references exist in the executable that are defined in the DSO, the runtime linker resolves these references on demand. You can also use *dlsym()* to access symbols in the DSO, whether or not the symbols are referenced in your executable.

When a DSO is brought into the address space of a process, it may contain references to symbols whose addresses are not known until the object is loaded. These references must be relocated before the symbols can be accessed. The second argument to *dlopen()* governs when these relocations take place.

This argument can have the following values:

RTLD_LAZY	Under this mode, only references to data symbols are relocated when the object is loaded. References to functions are not relocated until a given function is invoked for the first time. This mode should result in better performance, since a process may not reference all of the functions in any given shared object.
RTLD_NOW	Under this mode, all necessary relocations are performed when the object is first loaded. This may result in some wasted effort if relocations are performed for functions that are never referenced. However, this option is useful for applications that need to know as soon as an object is loaded that all symbols referenced during execution will be available.

To access symbols that are not referenced in your program, use *dlsym()*:

```
#include <dlfcn.h>
void *dlhandle;
int (*funcptr)(int);
int i,j;
 ... load DSO ...
funcptr = (int (*)(int)) dlsym(dlhandle, "factorial");
if (funcptr == NULL) {
 /* couldn’t locate the symbol */
}
i = (*funcptr)(j);
```

In this example, we look up the address of a function called *factorial()* and assign it to the function pointer *funcptr*.

If you encounter an error (*dlopen()* or *dlsym()* returns NULL), you can get diagnostic information by calling *dlerror().* The *dlerror()* function returns a string describing the cause of the latest error. You should only call *dlerror()* after an error has occurred; at other times, its return value is undefined.

To unload a DSO, call *dlclose()*:

```
#include <dlfcn.h>
void *dlhandle;
... load DSO, use DSO symbols ...
dlclose(dlhandle);
```

The *dlclose* function frees up the virtual address space *mmap*ed by the *dlopen* call of that file (similar to a *munmap* call). The difference, however, is that *dlclose* on a file that has been opened multiple times (either through *dlopen* or program startup) does not cause the file to be *munmap*ed until the file is no longer needed by the process.

Versioning of DSOs

In the 5.0.1 release, a mechanism for the versioning of shared objects was introduced for SGI–specific shared objects and executables. Note that this mechanism is outside the scope of the ABI, and, thus, must not be relied on for code that must be ABI–compliant and run on non–SGI platforms. Currently, all executables produced on SGI systems are marked SGI_ONLY to allow use of the versioning mechanism.

Versioning is mainly of interest to developers of shared objects. It may not be of interest to you if you simply *use* shared objects. Versioning allows a developer to update a shared object in a way that may be incompatible with executables previously linked against the shared object. This is accomplished by renaming the original shared object and providing it along with the (incompatible) new version.

What Is a Version?

A version is part or all of an identifying *version_string* that can be associated with a shared object by using the *-set_version version_string* option to *ld*(1) when the shared object is created.

A *version_string* consists of one or more versions separated by colons (:). A single version has the form:

[*comment***#**]**sgi***major***.***minor*

where:

comment	is a comment string, which is ignored by the versioning mechanism. It consists of any sequence of characters followed by a pound sign (#). The comment is optional.
sgi	is the literal string *sgi*.
major	is the major version number, which is a string of digits [0–9].
.	is a literal period.
minor	is the minor version number, which is a string of digits [0–9].

Follow these instructions when building your shared library:

When you first build your shared library, give it an initial version, for example, *sgi1.0*. Add the option *-set_version sgi1.0* to the command to build your shared library (*cc -shared*, *ld -shared*).

Whenever you make a *compatible* change to the shared object, create another version by changing the minor version number (for example, *sgi1.1*) and add it to the end of the *version_string*. The command to set the version of the shared library now looks like *-set_version "sgi1.0:sgi1.1"*.

When you make an *incompatible* change to the shared object:

1. Change the file name of the old shared object by adding a dot followed by the major number of one of the versions to the file name of the shared object. Do not change the *soname* of the shared object or its contents. Simply rename the file.
2. Update the major version number and set the *version_string* of the shared object (when you create it) to this new version; for example, *-set_version sgi2.0*.

This versioning mechanism affects executables in the following ways:

- When an executable is linked against a shared object, the last version in the shared object's *version_string* is recorded in the executable as part of the *liblist*. You can examine this using *elfdump -Dl*.
- When you run an executable, *rld* looks for the proper file name in its usual search routine.
- If a file is found with the correct name, the version specified in the executable for this shared object

is compared to each of the versions in the *version_string* in the shared object. If one of the versions in the *version_string* matches the executable's version exactly (ignoring comments), then that library is used.

- If no proper match is found, a new file name for the shared object is built by combining the *soname* specified in the executable for this shared object and the *major* number found in the version specified in the executable for this shared object (*soname.major*). Remember that you did *not* change the *soname* of the object, only the file name. The new file is searched for using *rld*'s usual search procedure.

For example, suppose you have a shared object *foo.so* with initial version *sgi10.0*. Over time, you make two compatible changes for *foo.so* that result in the following final *version_string* for *foo.so*:

```
initial_version#sgi10.0:upgrade#sgi10.1:new_devices#sgi10.2
```

You then link an executable that uses this shared object, *useoldfoo*. This executable specifies version *sgi10.2* for *soname* *foo.so*. (Remember that the executable inherits the last version in the *version_string* of the shared object.)

The time comes to upgrade *foo.so* in an incompatible way. Note that the *major* version of *foo.so* is 10, so you move the existing *foo.so* to the file name *foo.so.10* and create a new *foo.so* with the *version_string*:

```
efficient_interfaces#sgi11.0
```

New executables linked with *foo.so* use it directly. Older executables, like *useoldfoo*, attempt to use *foo.so*, but find that its version (*sgi11.0*) is not the version they need (*sgi10.2*). They then attempt to find a *foo.so* in the file name *foo.so.10* with version *sgi10.2*.

Chapter 4

Improving Program Performance

This chapter explains how to reduce the execution time of your programs using the compiler system. The techniques described here comprise only a minor part of performance tuning. Other areas that can be tuned, but which are outside the scope of this chapter, include graphics, I/O, the kernel, memory, and *REACT* (real−time) system calls. This chapter contains the following sections:

- **"Profiling With prof"** describes how to use the profiler, *prof*.
- **"Optimization"** describes the compiler optimization facility and how to use it. The section also contains examples demonstrating optimization techniques.

Although it may be possible to obtain short−term speed increases by relying on unsupported or undocumented quirks of the compiler system, it's a bad idea to do so. Any such "features" may well break in future releases of the operating system. The best way to produce efficient code that can be trusted to remain efficient is to follow good programming practices; in particular, choose good algorithms and leave the details to the compiler.

Profiling With *prof*

Profiling produces detailed information about program execution. Use profiling tools to find the areas of code where most of the execution time is spent. In a typical program, a large part of the execution time is spent in relatively few sections of code. It is a good idea to concentrate on improving code efficiency in those sections first.

Overview of Profiling

Profiling is a three−step process that consists of compiling the source program, executing the output, and running the profiler, *prof*, to analyze the data.

The compiler system provides two kinds of profiling:

- Program counter (pc) sampling—measures the amount of execution time spent in various parts of the program.
- Basic block counting—measures the execution of basic blocks (a basic block is a sequence of

instructions that is entered only at the beginning and exits only at the end). Basic block counting provides statistics on individual lines.

Running the Profiler

The profiler program, *prof*, converts raw profiling information to either a printed listing or an output file for use by the compiler.

Syntax

```
prof [options] [pname] [profile_filename ...]
```

options	One of the keywords or keyword abbreviations shown in **Table 4–1** . (Specify either the entire name or the initial character of the option, as indicated in the table.)
pname	Specifies the name of the program to be profiled. The default file is *a.out*.
profile_filename	Specifies one or more files containing the profile data gathered when the profiled program executed (defaults are explained below). If you specify more than one file, *prof* sums the statistics in the resulting profile listings.

The *prof* program has these defaults:

- If you do not specify *profile_filename*, the profiler looks for the *mon.out* file; if that file does not exist, *prof* looks for profile input data files in the directory specified by the PROFDIR environment variable (refer to **"Creating Multiple Profile Data Files"**).
- If you do not specify *profile_filename*, but do specify *-pixie*, then *prof* looks for *pname.Counts* and provides basic block count information if this file is present.

Note: *prof* and *pixie* now work on programs linked with shared libraries. For details refer to the *pixie*(1) manual page.

Consider using the *-merge* option when you have more than one profile data file. This option merges the data from several profile files into one file. See **Table 4–1** for information on the *-merge* option.

Table 4–1 Options for *prof*

Name	Result
-c[lock] n	A basic–block–counting option. Lists the number of seconds spent in each routine, based on the CPU clock frequency *n*, expressed in megahertz.
-dso [*dso_name*]	Applies *prof* analysis to only the named DSO. If you don't specify *dso_name*, *prof* prints a list of applicable dso names.
-e[xclude] *procedure_name*	Excludes information on the procedures (and their descendants) specified by *procedure_name*. If you specify uppercase *-E*, *prof* also omits that procedure from the base upon which it calculates percentages. This option overrides the *-include* option.
-h[eavy]	A basic–block–counting option. Same as the *-lines* option, but sorts the lines by their frequency of use.
-i[nvocations]	A basic–block–counting option. Lists the number of times each procedure is invoked. The *-exclude* and *-only* options described below apply to callees, but not to callers.
-l[ines]	A basic–block–counting option. Lists statistics for each line of source code.
-m[erge] filename	Merges the input files into *filename*, allowing you to specify the name of the merged file (instead of several file names) on subsequent profiler runs. This option is useful when using multiple input files of profile data (normally in *mon.out*).
-o[nly] *procedure_name*	Reports information on only the procedure specified by *procedure_name* rather than the entire program. You can specify more than one *-o* option. If you specify uppercase *-O*, *prof* uses only the named procedures, rather than the entire program, as the base upon which it calculates percentages.
-pixie	Indicates that information is to be generated on basic block counting, and that the *pname.Counts* files produced by *pixie* are to be used by default.
-p[rocedures]	Lists the time spent in each procedure.
-q[uit] n	Condenses output listings by truncating unwanted lines. You can specify *n* in three different ways: *n*, an integer, truncates everything after *n* lines; *n%*, an integer

	followed by a percent sign, truncates everything after the line containing *n%* calls in the %calls column; and an integer followed by "cum%" truncates everything after the line containing *ncum%* calls in the cum% column.
-*t*[*estcoverage*]	A basic–block–counting option. Lists line numbers that contain code that is never executed.
-*z*[*ero*]	A basic–block–counting option. Lists the procedures that are never invoked.

pc Sampling

Program counter (pc) sampling reveals the amount of execution time spent in various parts of a program. The count includes:

- CPU time
- Memory access time
- Time spent in user routines
- Time spent in system routines

Pc sampling does not count time spent swapping or time spent accessing external resources.

This section explains how to obtain pc sampling and provides an example for clarification. It also explains how to create multiple profile data files.

Obtaining pc Sampling

Obtain pc sampling information by link–editing the desired source modules using the *-p* option and then executing the resulting program object, which generates raw profile data.

Use the procedure below to obtain pc sampling information (refer to **Figure 4–1**).

1. Compile the program using the appropriate compiler driver. For example, to compile a C program *myprog.c*:

   ```
   IRIS% cc –c myprog.c
   ```

2. Link–edit the object file created in Step 1.

   ```
   IRIS% cc –p –o myprog myprog.o
   ```

   Note: You must specify the *-p* profiling option during this step to obtain pc sampling information.

3. Execute the profiled program. (During execution, profiling data is saved in the file *mon.out*.)

   ```
   IRIS% myprog
   ```

   You can run the program several times, altering the input data, to create multiple profile data files. You can also use the environment variable PROFDIR as explained later in this section.

4. Run the profile formatting program *prof*.

   ```
   IRIS% prof myprog mon.out
   ```

   prof extracts information from *mon.out* and prints it in an easily readable format. For more information see the *prof*(1) manual page.

Include or exclude information on specific procedures within your program by using the *-only* or *-exclude* profiler options (refer to **Table 4–1**).

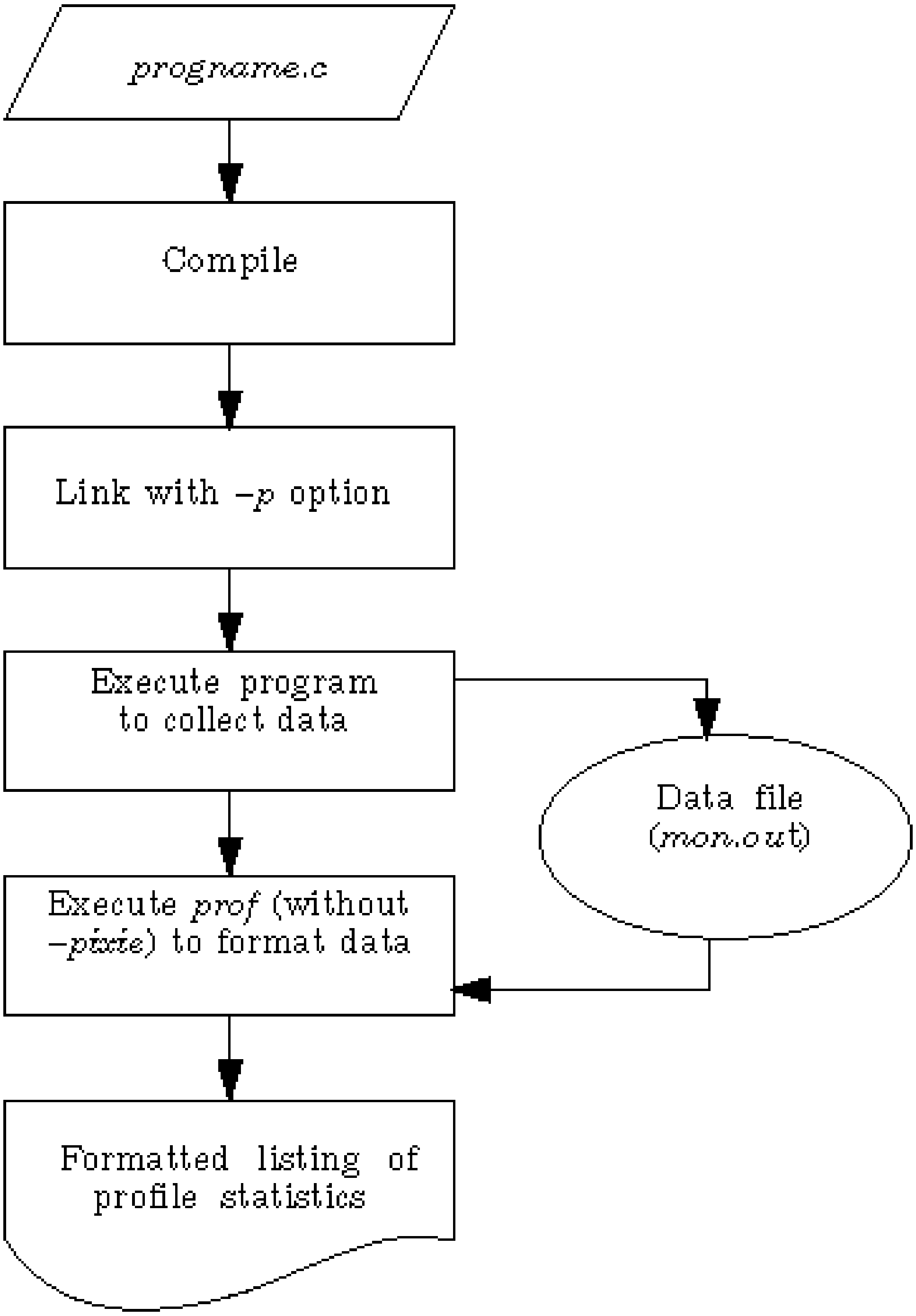

Figure 4–1 How pc Sampling Works

Example

The following listing is an example of pc sampling output from a profiled version of the ANSI C preprocessor, *acpp*:

```
Profile listing generated Sat Feb 23 15:00:10 1991 with:
    prof acpp
-----------------------------------------------------------
*  –p[rocedures] using pc–sampling sorted in descending   *
```

```
*  -p[rocedures] using pc-sampling sorted in descending   *
*  order by total time spent in each procedure;           *
*  unexecuted procedures excluded                         *
-----------------------------------------------------------

Each sample covers 8.00 byte(s) for 1.1% of 0.9000 seconds

%time  seconds  cum %  cum sec  procedure (file)

   28.9  0.2600   28.9   0.26    rescan (cccp.c)
   27.8  0.2500   56.7   0.51    write (sys/write.s)
   10.0  0.0900   66.7   0.60    handle_directive (cccp.c)
    8.9  0.0800   75.6   0.68    read (sys/read.s)
    5.6  0.0500   81.1   0.73    malloc (malloc.c)
    5.6  0.0500   86.7   0.78    collect_expansion (cccp.c)
    3.3  0.0300   90.0   0.81    open (sys/open.s)
    2.2  0.0200   92.2   0.83    hashf (cccp.c)
    2.2  0.0200   94.4   0.85    skip_if_group (cccp.c)
    1.1  0.0100   95.6   0.86    main (cccp.c)
    1.1  0.0100   96.7   0.87    do_define (cccp.c)
    1.1  0.0100   97.8   0.88    macroexpand (cccp.c)
    1.1  0.0100   98.9   0.89    skip_to_end_ofcomment (cccp.c)
    1.1  0.0100  100.0   0.90    strncmp (strncmp.c)
```

In the above listing:

- 0.09 seconds (10.0% of execution time) were spent in *handle_directive*.
- 0.6 seconds (66.7% of total execution time) were spent cumulatively in the *rescan*, *write*, and *handle_directive* routines.
- The name of the source file containing the *handle_directive* routine is *cccp.c*.

Creating Multiple Profile Data Files

When you run a program using pc sampling, raw data is collected and saved in the profile data file *mon.out*. To collect profile data in several files, or to specify a different name for the profile data file, set the environment variable PROFDIR, using the appropriate method from **Table 4–2** :

Table 4–2 Setting the PROFDIR environment variable

C Shell	**Bourne Shell**
setenv PROFDIR *dirname*	PROFDIR=*dirname*; export PROFDIR

Setting this environment variable places the raw profile data of each invocation of *progname* in files named *dirname/progname.mon.pid*. (You must create a directory called *dirname* before you run the program.) *pid* is the process ID of the executing program, and *progname* is the name of the program when invoked.

Basic Block Counting

Basic block counting, obtained using the program *pixie*, measures the execution of basic blocks. A basic block is a sequence of instructions that is entered only at the beginning and exits only at the end.

Using pixie

Use *pixie* to measure the frequency of code execution. *Pixie* reads an executable program, partitions it into basic blocks, and writes an equivalent program containing additional code that counts the execution of each basic block.

Syntax

```
pixie file [options]
```

file	Name of the input program.
options	One of the keywords listed in **Table 4–3** .

For a complete list of options refer to the *pixie*(1) manual page.

Table 4–3 Options for *pixie*

Name	Result
-pixie_file out_file	Specifies a name for the output file. Default: remove any leading directory names from the input file name and append *.pixie*.
-counts_file file	Specifies a name for the counts file generated while running the program with *pixie*. Default: remove any leading directory names from the input file name and append *.Counts*.
-[*no*]*autopixie*	[Prevents] or permits a recursive instrumenting all dynamic shared libraries used by the input file during runtime. *pixie* cannot detect libraries opened with *dlopen*. This option works only when invoked on the main *a.out*. This option is not useful if the input file is non–shared. Default: *-autopixie*.
-[*no*]*liblist*	[Prevents] or permits printing the names and paths of dynamic shared libraries used by the input file during runtime. This uses the same default search path strategy as *rld* and *prof*. This list is useful to build a dependency list for *makefiles* and shell scripts. *pixie* removes any leading directory names from the input file name and appends *.liblist*. *pixie* cannot detect libraries opened with *dlopen*. Default: *-noliblist*.
-[*no*]*verbose*	[Permits] or suppresses messages summarizing the binary–to–binary translation process. Default: *-verbose*.

Obtaining Basic Block Counts

Use this procedure to obtain basic block counts (refer to **Figure 4–2**).

1. Compile and link edit your program. Do not use the *-p* option. This example uses the input file *myprog.c*.

   ```
   IRIS%   cc –o myprog myprog.c
   ```

   The *cc* compiler compiles *myprog.c* into an executable called *myprog*.

2. Run *pixie* to generate the equivalent program containing basic–block–counting code.

   ```
   IRIS%   pixie myprog
   ```

   Pixie takes *myprog* and writes an equivalent program, *myprog.pixie*, containing additional code that counts the execution of each basic block. *Pixie* also writes an equivalent program for each shared object used by the program, *libname.so.pixie*, containing additional code that counts the execution of each basic block. For example, if *myprog* uses *libc.so.1*, pixie generates *libc.so.1.pixie*.

3. Set the path for your *.pixie* files. *Pixie* uses the *rld* search path for libraries (see *rld*(1) for the default paths). If the *.pixie* files are in your local directory, set the path as:

   ```
   IRIS% setenv LD_LIBRARY_PATH .
   ```

4. Execute the files generated by *pixie* (*myprog.pixie*) in the same way you executed the original program.

   ```
   IRIS%   myprog.pixie
   ```

   This program generates a list of basic block counts in files named *myprog.Counts*. If the program executes a fork/sproc, a process ID is appended to the end of the file name (for example, *myprog.Counts.345*) for each process.

5. Run the profile formatting program *prof* specifying the *-pixie* option and the name of the original program.

   ```
   IRIS%   prof –pixie myprog myprog.Counts
   ```

   prof extracts information from *myprog.Counts* and prints it in an easily readable format. If multiple *.Counts* files exist, you can use the wildcard character (*) to specify all of the files.

```
IRIS%    prof -pixie myprog myprog.Counts*
```

Note: Specifying *myprog.Counts* is optional; *prof* searches by default for names having the form *program_name.Counts*.

You can run the program several times, altering the input data, to create multiple profile data files. See **"Summing Basic Block Count Results"** later in this section for an example.

The complete output of the *-pixie* option is often extremely large. Use the *-quit* option with *prof* to restrict the size of the output. Refer to **"Running the Profiler"** for details about *prof* options.

Include or exclude information on specific procedures within your program by using the *prof* options *-only* or *-exclude* (explained in **Table 4–3**). *prof* timings reflect only time spent in a specific procedure, not time spent including procedures called by that procedure. To show the time spent including called procedures specify the *-g* option to *prof*.

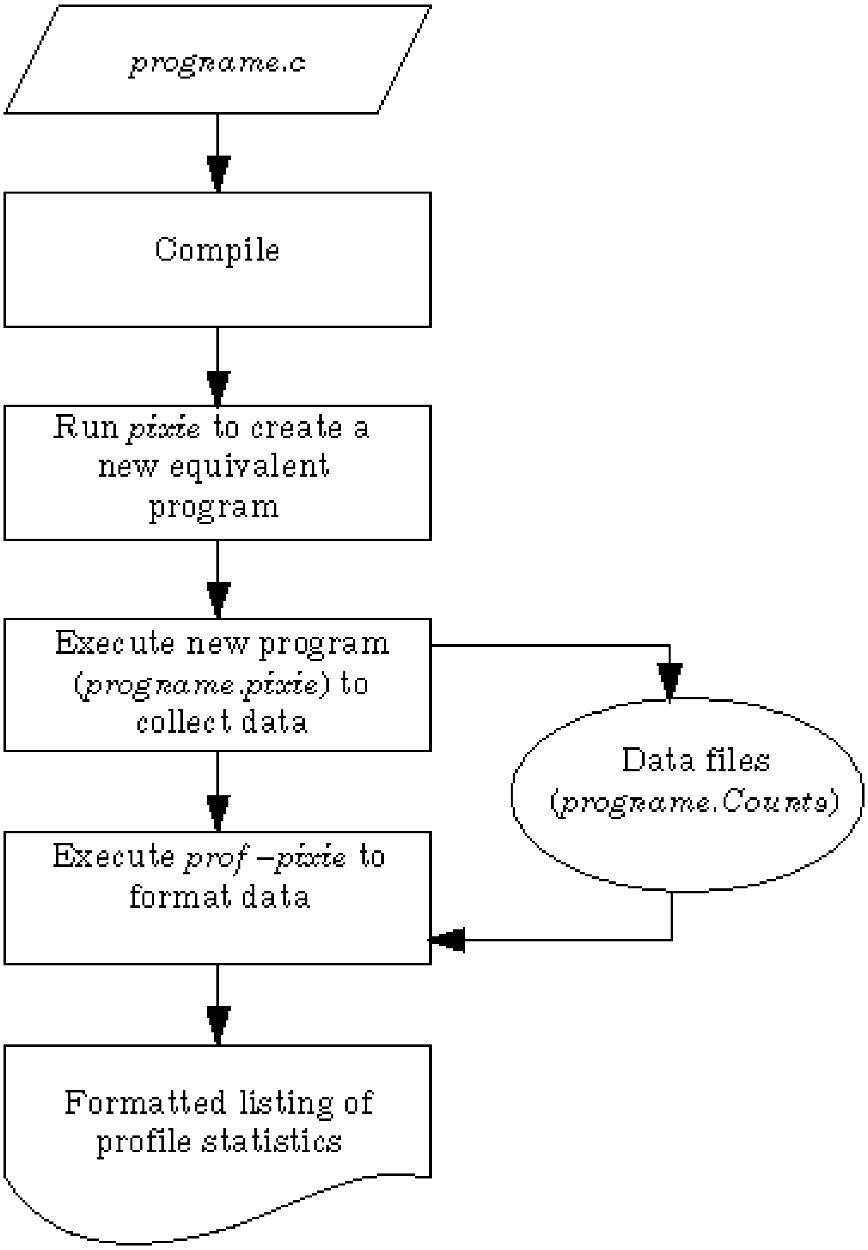

Figure 4–2 How Basic Block Counting Works

Examples

The following listings are examples of basic block counting output using *prof -pixie*, from a profiled version of a C file, *espresso*.

The first listing illustrates the use of the -*i*[*nvocations*] option. For each procedure invoked, *prof* produces a list of the procedures that called it:

```
IRIS% prof -pixie -i espresso
```

```
-----------------------------------------------------------------------------
--
Profile listing generated Tue Jan 18 19:31:53 1994
      with:       prof -pixie -i espresso
Assumption: All instructions take one cycle.
-----------------------------------------------------------------------------
--

Total cycles    Total Time  Instructions Cycles/inst    Clock
    1183248220        11.83s    1183248220       1.000 100.0MHz

-----------------------------------------------------------------------------
-
*  -i[nvocations] using basic block invocation counts.
                       *
*     Sorted in descending order by number of calls per procedure.
           *
*     Unexecuted procedures are excluded.
                                    *
*     The inst column is a static number of instructions.
                    *
*     %coverage column contains the percent inst executed.
                   *
-----------------------------------------------------------------------------
-

Total procedure invocations: 8580657

           calls(%)    cum%      inst     %coverage procedure (file)

     1632626(19.03)   19.03       108     22.22 full_row (espresso:setc.c)
      528962( 6.16)   25.19       828     24.76 __malloc
 (./libc.so.1:malloc.c)
      528945( 6.16)   31.36       172     25.00 __free (./libc.so.1:malloc.c)
      526081( 6.13)   37.49       176     14.77 _malloc
 (./libc.so.1:malloc.c)
      526080( 6.13)   43.62       168     14.88 _free (./libc.so.1:malloc.c)
      485880( 5.66)   49.28       104     22.12 set_or (espresso:set.c)
      447933( 5.22)   54.50       672     24.85 realfree
 (./libc.so.1:malloc.c)
      407124( 4.74)   59.25       200     24.00 cleanfree
 (./libc.so.1:malloc.c)
      336135( 3.92)   63.16       288     24.31 _smalloc
 (./libc.so.1:malloc.c)
      333396( 3.89)   67.05       964     11.51 cdist01 (espresso:setc.c)
      316901( 3.69)   70.74        92     21.74 setp_implies (espresso:set.c)
      274750( 3.20)   73.94        76     21.05 set_clear (espresso:set.c)
```

```
      235410( 2.74)   76.69       768     13.54 force_lower (espresso:setc.c)
      219841( 2.56)   79.25       108     22.22 set_diff (espresso:set.c)
      107445( 1.25)   80.50       104     22.12 set_and (espresso:set.c)
      105270( 1.23)   81.73       540     20.93 t_delete
 (./libc.so.1:malloc.c)
       97346( 1.13)   82.86        64     20.31 set_copy (espresso:set.c)
       96713( 1.13)   83.99       492     23.98 scofactor
 (espresso:cofactor.c)
       95491( 1.11)   85.10       292     18.84 sf_addset (espresso:set.c)
       81875( 0.95)   86.06       520     24.04 t_splay
 (./libc.so.1:malloc.c)
```

The following listing shows the source code lines responsible for the largest portion of execution time produced with the *-heavy* option:

```
IRIS% prof -pixie -heavy acpp


Profile listing generated Sat Feb 23 15:00:03 1991 with:
    prof -pixie -heavy acpp
-------------------------------------------------------------
*  -h[eavy] using basic-block counts;                        *
*  sorted in descending order by the number of cycles       *
*  executed in each line; unexecuted lines are excluded     *
-------------------------------------------------------------

procedure (file)            line bytes     cycles    %    cum %

rescan (cccp.c)             1431    40     940652  12.80  12.80
handle_directive (cccp.c)   2291    68     428379   5.83  18.63
rescan (cccp.c)             1678    16     369193   5.02  23.65
skip_if_group (cccp.c)      4067    68     280109   3.81  27.46
rescan (cccp.c)             1679    28     277812   3.78  31.24
rescan (cccp.c)             1429     8     189746   2.58  33.82
rescan (cccp.c)             1659     8     186960   2.54  36.36
rescan (cccp.c)             2014    92     149390   2.03  38.40
rescan (cccp.c)             1720    40     143226   1.95  40.34
rescan (cccp.c)             1721    12     139773   1.90  42.25
rescan (cccp.c)             1723    12     139773   1.90  44.15
handle_directive (cccp.c)   2375    72     137623   1.87  46.02
rescan (cccp.c)             1832    16     114445   1.56  47.58
handle_directive (cccp.c)   2372    20     110966   1.51  49.09
bzero (cccp.c)              5595    20     100660   1.37  50.46
collect_expansion (cccp.c)  3309    44      97100   1.32  51.78
collect_expansion (cccp.c)  3357    72      95715   1.30  53.08
rescan (cccp.c)             1428     8      94988   1.29  54.37
```

In the above listing:

- The most heavily used line is line 1431 in procedure *rescan*, compiled from the source file *cccp.c*.
- Lines 1431, 2291, and 1678 executed 23.65% of the total program cycles.
- Line 1431 of *cccp.c* has 40 bytes of code and used 940652 cycles, which is 12.80% of the total program cycles.

The following listing, produced using the *-lines* option, shows the execution time spent on each line of code, grouped by procedure:

```
IRIS% prof -pixie -lines acpp

Profile listing generated Sat Feb 23 15:00:05 1991 with:
    prof -pixie -lines acpp
-------------------------------------------------------------
```

```
* -l[ines] using basic-block counts; grouped by procedure, *
* sorted by cycles executed per procedure;  '?' means that *
* line number information is not available.               *
-------------------------------------------------------------
procedure (file)                  line bytes    cycles %cycles
rescan (cccp.c)                   1355   52      1495  0.02
                                   1360    4       115  0.00
                                   1363    4       115  0.00
                                   1383    4       115  0.00
                                   1415   52       230  0.00
                                   1416    8         0  0.00
                                   1418   32       920  0.01
                                   1419   64      1612  0.02
                                   1420    4       115  0.00
                                   1424   16       460  0.00
                                   1425   24       460  0.00
                                   1428    8     94988  1.29
                                   1429    8    189746  2.58
                                   1431   40    940652 12.80
                                   1433   12         0  0.00
                                   1434   12         0  0.00
                                   1435   16         0  0.00
                                   1438    4         0  0.00
                                   1439   12         0  0.00
                                   1440    4         0  0.00
                                   1441    8         0  0.00
                                   1446   16         0  0.00
                                   1447   12         0  0.00
```

In the above listing:

- The statistics describe the lines of code in procedure *rescan* compiled from the source file *cccp.c*.
- Line 1355 in *rescan* contains 52 bytes of code, executed 1,495 times, using 0.02% of the total program cycles.
- Line 1360 in *rescan* contains 4 bytes of code and did not execute any recorded cycles.

You can limit the output of *prof* to information on only the most time−consuming parts of the program by specifying the *-quit* option. You can instruct *prof* to quit after a particular number of lines of output, after listing the elements consuming more than a certain percentage of the total, or after the portion of each listing whose cumulative use is a certain amount.

Consider the following sample listing:

```
calls     %call    cum%
48071708  32.45    32.45  6.0090
42443503  28.65    61.10  5.3054
26457936  17.86    78.96  3.3072
20662326  13.95    92.91  2.5828
4307932    2.91    95.82  0.5385
3678408    2.48    98.30  0.4598
1573858    1.06    99.36  0.1967
362700     0.24    99.61  0.0453
279002     0.19    99.80  0.0349
251152     0.17    99.97  0.0314
30283      0.02    99.99  0.0038
13391      0.01    100.0  0.0017
2923       0.00    100.00 0.0017
```

Any one of the following commands eliminates everything from the line starting with 4307932 to the end of the listing:

```
prof −quit 4
```

```
prof -quit 13%
prof -quit 92cum%
```

The following listing, produced with the *-procedures* option, shows the percentage of execution time spent in each procedure:

```
IRIS% prof -pixie -procedures acpp

Profile listing generated Sat Feb 23 15:00:01 1991 with:
    prof -pixie -procedures acpp
-------------------------------------------------------------
* -p[rocedures] using basic-block counts; sorted in         *
* descending order by the number of cycles executed in      *
* each procedure; unexecuted procedures are excluded        *
-------------------------------------------------------------
7350305 cycles
cycles %cycles cum % cycles bytes procedure (file)
                      /call /line
3536769 48.12  48.12 30755   17   rescan (cccp.c)
1231455 16.75  64.87  1671   18   handle_directive (cccp.c)
684976   9.32  74.19  1616   19   collect_expansion (cccp.c)
453805   6.17  80.36  4777   17   skip_if_group (cccp.c)
258087   3.51  83.88   118   14   malloc (malloc.c)
234150   3.19  87.06   445   17   skip_to_end_of_comment (cccp.c)
111560   1.52  88.58    46   21   strncmp (strncmp.c)
109313   1.49  90.07   258   18   do_define (cccp.c)
101035   1.37  91.44  1348   14   bzero (cccp.c)
   83969   1.14  92.58   124   13   skip_quoted_string (cccp.c)
   71584   0.97  93.56   309   21   macroexpand (cccp.c)
   52005   0.71  94.26    91   27   hashf (cccp.c)
   50685   0.69  94.95   114   17   install (cccp.c)
   49438   0.67  95.63   634   18   _doprnt (doprnt.c)
   43520   0.59  96.22    75   13   xcalloc (cccp.c)
   30143   0.41  96.63    63   16   bcopy (cccp.c)
```

In the above listing:

- The total number of program cycles is 7,350,305.
- *rescan* used 3,536,769 cycles, which is 48.12% of the total number of program cycles.
- The cumulative total of all cycles used by *rescan*, *handle_directive*, and *collect_expansion* is 74.19% (see column 3 of the third row).
- *rescan* used an average of 30,755 cycles per call and consisted of 17 bytes of generated code per line of source text.
- The procedure *rescan* is in the source file *cccp.c*.

You can add absolute time information to the output by specifying the clock rate, in megahertz, with the *-clock* option. The output looks like this:

```
IRIS% prof -pixie -procedures -clock 20 acpp

Profile listing generated Sat Feb 23 15:00:01 1991 with:
    prof -pixie -procedures -clock acpp
-------------------------------------------------------------
*  -p[rocedures] using basic-block counts; sorted in        *
*  descending order by the number of cycles executed in     *
*  each procedure; unexecuted procedures are excluded.      *
-------------------------------------------------------------
7350305 cycles (0.3675 seconds at 20.00 megahertz)
cycles %cycles cum % seconds cycles bytes procedure (file)
                              /call  /line
3536769 48.12  48.12 0.176 30755 17  rescan (cccp.c)
1231455 16.75  64.87 0.0616 1671 18  handle_directive (cccp.c)
```

```
684976  9.32  74.19  0.0342 1616  19  collect_expansion (cccp.c)
453805  6.17  80.36  0.022  4777  17  skip_if_group (cccp.c)
258087  3.51  83.88  0.0129  118  14  malloc (malloc.c)
111560  1.52  88.58  0.0056   46  21  strncmp (strncmp.c)
109313  1.49  90.07  0.0055  258  18  do_define (cccp.c)
101035  1.37  91.44  0.00   1348  14  bzero (cccp.c)
   71584  0.97  93.56  0.0036   30  21  macroexpand (cccp.c)
   52005  0.71  94.26  0.0026   91  27  hashf (cccp.c)
   50685  0.69  94.95  0.0025  114  17  install (cccp.c)
   49438  0.67  95.63  0.0025  634  18  _doprnt (doprnt.c)
   43520  0.59  96.22  0.0022   75  13  xcalloc (cccp.c)
   30143  0.41  96.63  0.0015   63  16  bcopy (cccp.c)
   26577  0.36  96.99  0.0013   55  15  alloca (alloca.c)
```

The above listing contains the same information as the previous listing except that this listing also contains the number of seconds spent in each procedure. The profiler computes the time, in seconds, based on the machine speed specified with the *-clock* option (in megahertz). In this example, the speed specified is 20 megahertz.

Summing Basic Block Count Results

Sometimes a single run of a program does not produce the results you require. You can repeatedly run the version of a program created by *pixie* and vary the input with each run, then use the resulting *.Counts* files to produce a consolidated report.

Use the following procedure to average *prof* results:

1. Compile and link−edit the input file. Do not use the *-p* option. For example, using the input file *myprog.c*:

   ```
   IRIS% cc −o myprog myprog.c
   ```

   The *cc* compiler compiles *myprog.c* and saves the object file as *myprog.o*. The second command link−edits *myprog.o* and saves the executable as *myprog*.

2. Run the profiling program *pixie*.

   ```
   IRIS% pixie myprog
   ```

   pixie generates the modified program *myprog.pixie*.

3. Run the profiled program as many times as desired. Each time you run the program, *pixie* creates a *myprog.Counts* file. Rename this file before executing the next sample run.

   ```
   IRIS% myprog.pixie < input1 > output1
   IRIS% mv myprog.Counts myprog1.Counts
   IRIS% myprog.pixie < input2 > output2
   IRIS% mv myprog.Counts myprog2.Counts
   IRIS% myprog.pixie < input3 > output3
   IRIS% mv myprog.Counts myprog3.Counts
   ```

4. Create the report.

   ```
   IRIS% prof −pixie myprog myprog[123].Counts
   ```

   prof takes an average of the basic block data in the *myprog1.Counts*, *myprog2.Counts*, and *myprog3.Counts* files to produce the profile report.

   You can also instrument the program by using the *-pids* option, which produces files with unique names. Produce a profile report by typing:

   ```
   IRIS% prof −pixie myprog myprog.Counts*
   ```

Using pixstats

Use the *pixstats* command to analyze a program's execution characteristics. *prof*–generated reports ignore time spent in floating point operations and time spent accessing memory. *pixstats* provides accurate floating point information by analyzing the *.Counts* files created through *pixie*.

The disadvantages to using *pixstats* are that it:

- Does not provide a line–by–line count
- Profiles only one *.Counts* file at a time (no averaging)
- Provides very little documentation
- Does not show time spent in floating point exceptions

You can also use *pixstats* to look for write buffer stalls and to produce disassembled code listings.

Syntax

```
pixstats program [options]
```

program	Specifies the name of the program to be analyzed.
options	One of the keywords shown in **Table 4–4**

Table 4–4 Options for *pixstats*

Name	**Result**
-cycle ns	Assumes an *ns* cycle time when converting cycle counts to seconds.
-r2010	Uses r2010 floating point chip operation times and overlap rules. This option is the default.
-r2360	Uses r2360 floating point board operation times and overlap rules.
-dis	Disassembles and annotates the analyzed object code.

Other options are explained in *pixstats*(1).

Use the following procedure to run *pixstats*:

1. Compile and link edit the input file *myprog.c*. Do not use the *-p* option. For example, using the input file *myprog.c*:

   ```
   IRIS% cc –c myprog.c
   IRIS% cc –o myprog myprog.o
   ```

   The *cc* compiler driver compiles *myprog.c* and saves the object file as *myprog.o*. The second command link–edits *myprog.o* and saves the executable as *myprog*.

2. Run the profiling program *pixie*.

   ```
   IRIS% pixie myprog
   ```

   pixie generates the modified program *myprog.pixie*.

3. Set the path, so *pixie* knows where to find the *.pixie* files.

   ```
   IRIS% setenv LD_LIBRARY_PATH .
   ```

4. Execute the file generated by *pixie*, *myprog.pixie*, in the same way you would execute the original program.

   ```
   IRIS% myprog.pixie
   ```

   This program generates the file *myprog.Counts* which contains the basic block counts.

5. Run *pixstats* to generate a detailed report.

   ```
   IRIS% pixstats myprog
   ```

Example

The following example shows the default listing generated by *pixstats*:

```
pixstats espresso:
```

```
1588254395 (1.357/inst) cycles (15.88s @ 100.0MHz)
1170355761 (1.000/inst) instructions

        2397 (0.000/inst) floating point ops (0.000151 MFLOPS @ 100.0MHz)
Procedures ordered by execution time:
      cycles %cycles  cum%     instrs  cycles   calls     cycles procedure
                                       /inst               /call
   382093989  24.1%  24.1%  278174631    1.4    77888       4906
 massive_count
   194452825  12.2%  36.3%  130750578    1.5    11265      17262 cofactor
   146765915   9.2%  45.5%  104525532    1.4     3599      40780
 elim_lowering
   144704109   9.1%  54.7%  113501194    1.3     3163      45749 essen_parts
    65043668   4.1%  58.7%   51198838    1.3    96713        673 scofactor
    57256404   3.6%  62.4%   41920736    1.4    34564      1657
 taut_special_cases
    54258762   3.4%  65.8%   43594130    1.2  1632626         33 full_row
    43947988   2.8%  68.5%   32692126    1.3    72095      610
 sccc_special_cases
    42611632   2.7%  71.2%   27971390    1.5     2370      17980 setup_BB_CC
    35769668   2.3%  73.5%   26776245    1.3   528962         68 __malloc
    29107500   1.8%  75.3%   24012582    1.2   333396         87 cdist01
    28840766   1.8%  77.1%   23333068    1.2   235410        123 force_lower
    27458158   1.7%  78.8%   21150937    1.3   447933         61 realfree
    26682338   1.7%  80.5%   21303304    1.3   407124         66 cleanfree
    21207623   1.3%  81.9%   16338599    1.3   528945         40 __free
    19991078   1.3%  83.1%   13678106    1.5   526081         38 _malloc
    19464960   1.2%  84.3%   13152000    1.5   526080         37 _free
    17434271   1.1%  85.4%   15501189    1.1   485880         36 set_or
    14574313   0.9%  86.4%    9466949    1.5      725      20103 expand1_gasp
    13115606   0.8%  87.2%    9753942    1.3   336135         39 _smalloc
    10646822   0.7%  87.9%    7928278    1.3   316901         34 setp_implies
     9812911   0.6%  88.5%    7858695    1.2    50103      196
 binate_split_select
     9487312   0.6%  89.1%    7085744    1.3      908      10449 compl_lift
     9342972   0.6%  89.7%    6531388    1.4      567      16478 cb_consensus
     8825424   0.6%  90.2%    7330700    1.2    55166        160 consensus
     8353958   0.5%  90.7%    7474594    1.1   219841         38 set_diff
     7858360   0.5%  91.2%    6439566    1.2    32735
        240cb_consensus_dist0
     7670120   0.5%  91.7%    5122330    1.5    72095        106 sccc
     7606139   0.5%  92.2%    6589200    1.2    66552        114 cactive
     6833384   0.4%  92.6%    4754352    1.4   105270         65 t_delete
     6561606   0.4%  93.0%    4923904    1.3   274750         24 set_clear
     6065768   0.4%  93.4%    4007453    1.5    44414        137 sm_insert
     5738712   0.4%  93.8%    4397726    1.3    95491         60 sf_addset
     5477719   0.3%  94.1%    3639951    1.5   117592         47 t_splay
     5162590   0.3%  94.5%    3351656    1.5      789       6543
 essen_raising
     5134827   0.3%  94.8%    3968016    1.3        1    5134827 rm_contain
     4737968   0.3%  95.1%    4006370    1.2    34838        136 sccc_merge
     4611671   0.3%  95.4%    3079225    1.5     2120       2175 intcpy
     3868020   0.2%  95.6%    3438240    1.1   107445         36 set_and
     3862752   0.2%  95.9%    3338484    1.2    66552         58 sccc_cube
```

In the above listing, the first line shows the total cycles used and the second line shows the total instructions. The third line shows the number of floating point operations.

You can use *pixstats -op* to generate a detailed listing about instructions and operations. Information includes instruction distribution, stall distribution, basic block size distribution, and register usage.

The following example shows a partial listing generated by running *pixstats -op* on the C file, *espresso*.

```
IRIS% pixstats -op espresso
1588254395 (1.357/inst) cycles (15.88s @ 100.0MHz)
1170355761 (1.000/inst) instructions
    12892539 (0.011/inst) instructions annulled by untaken branch likely
   250767706 (0.214/inst) cycles lost on non-sequential fetch
   152077341 (0.130/inst) alu/shift/load result interlock cycles
      689848 (0.001/inst) multiply interlock cycles (12 cycles)
      692925 (0.001/inst) divide interlock cycles (75 cycles)
      513369 (0.000/inst) variable shift extra issue cycles (2 total issue)
       29852 (0.000/inst) floating point result interlock cycles
          48 (0.000/inst) floating point compare interlock cycles
       71323 (0.000/inst) interlock cycles due to basic block boundary

    42595822 (0.036/inst) nops
   570540263 (0.487/inst) alu (including logicals, shifts, traps)
   185728252 (0.159/inst) logicals (including moves and li)
    72264550 (0.062/inst) shifts
   294103332 (0.251/inst) loads
    70309957 (0.060/inst) stores
   364413289 (0.311/inst) loads+stores
   120292374 (0.103/inst) load followed by load
   364415843 (0.311/inst) data bus use
       77171 (0.000/inst) partial word references
   117123216 (0.100/inst) sp+gp load/stores

        2397 (0.000/inst) floating point ops (0.000151 MFLOPS @ 100.0MHz)
          48 (0.000/inst) floating point compares
           0 (0.000/inst) overlapped floating point cycles
   168207521 (0.144/inst) conditional branches
    35852721 (0.031/inst) branch likelies
    32916066 (0.028/inst) load in branch delay slot
    20834011 (0.018/inst) branch to branch
    13630671 (0.012/inst) branch to branch taken
     5836780 (0.005/inst) branch to branch untaken
    41296177 (0.035/inst) branch delay filled with target-1 instruction
    67419786 (0.058/inst) untaken conditional branches
   100787735 (0.086/inst) taken conditional branches
    17365744 (0.015/inst) taken conditional branches with bnop
     7598403 (0.006/inst) untaken conditional branches with bnop
    15749049 (0.013/inst) direction-predicted conditional branches with bnop
   125383991 (0.107/inst) non-sequential fetches
   236116938 (0.202/inst) basic blocks
     8578595 (0.007/inst) calls
     8594643 (0.007/inst) non-R31 JR
     3511926 (0.003/inst) addui opportunities
           0 (0.000/inst) fp multiply/add opportunities
     2220801 (0.002/inst) skip
```

You can use *pixstats* to disassemble and annotate the analyzed object code. The next example shows *pixstats -dis*[*assemble*]. The file, *espresso*, was executed on an R4000 CPU; results will differ on other CPUs.

```
IRIS% pixstats -dis espresso
     0   43 404618 8fbc0020       lw      gp,32(sp)
    ^---11265 total cycles. Executed 11265 times, avg 1 (0.00107% of
 inst.)---^
```

```
     0   43 40461c 0040f825       move    ra,v0
     1   43 404620 afa20030       sw      v0,48(sp)
     2   43 404624 8f858574       lw      a1,-31372(gp)
                      << 2 cycle interlock >>
     5   43 404628 8ca50000       lw      a1,0(a1)
                      << 2 cycle interlock >>
     8   46 40462c 28a10021       slti    at,a1,33
     9   46 404630 10200003       beq     at,zero,0x404640
    10   46 404634 24a2ffff       addiu   v0,a1,-1
                      << branch taken 11265 times (100%) >>
                      << possible 2 cycles branch penalty, total 22530, avg
 2 >>
   ^---146445 total cycles. Executed 11265 times, avg 13 (0.0075% of
 inst.)---^
     0   46 404638 10000003       b       0x404648
     1   46 40463c 24020002       li      v0,2
   ^---    0 total cycles. Executed  0 times, avg 4 (0% of inst.)---^
     0   46 404640 00021143       sra     v0,v0,5
     1   46 404644 24420002       addiu   v0,v0,2
   ^---22530 total cycles. Executed 11265 times, avg 2 (0.00214% of
 inst.)---^
```

The second line lists the total number of cycles for basic block **8fbc0020**, (listed in the previous line). Line six shows a 2−cycle interlock because of a load of register **a1** (referenced in line seven).

Line 12 shows that a branch was taken 11,265 times, and that branches were taken 100% of the time. A possible branch penalty exists for every branch. Line 13 shows an average of 2 penalties occurred for a total of 22,530 penalties (a large number because the branch was always taken).

Profiling Multiprocessed Executables

You can gather both *pixie* and pc sampling profile data from executables that use the *sproc* and *sprocsp* system calls, such as POWER Fortran and POWER C executables. Prepare and run the job using the same process as for uniprocessed executables. For multiprocessed executables, each thread of execution writes its own separate profile data file. View these data files with *prof* like any other profile data files.

The only difference between multiprocessed and regular executables is the way in which the data files are named. When using pc sampling, the data files for multiprocessed executables are named *process_id.program_name*. When using *pixie*, the data files are named *program_name.Countsprocess_id*. This naming convention avoids the potential conflict of all the threads attempting to write simultaneously to the same file.

Optimization

This section describes the compiler optimization facilities and explains their benefits, the implications of optimizing and debugging, and the major optimizing techniques.

Overview

This section contains an overview of optimization. It explains the global optimizer, the benefits of optimization, and other general topics.

Global Optimizer

The global optimizer is a single program that improves the performance of object programs by transforming existing code into more efficient coding sequences. The optimizer distinguishes between C, Pascal, and Fortran programs to take advantage of the various language semantics involved.

Silicon Graphics compilers perform both machine−independent and machine−dependent optimizations.

Silicon Graphics machines and other machines with reduced instruction set computing (RISC) architectures provide a better target for machine−dependent optimizations. The low−level instructions of RISC machines provide more optimization opportunities than the high−level instructions in other machines. Even optimizations that are machine−independent have been found to be effective on machines with RISC architectures. Although most optimizations performed by the global optimizer are machine−independent, they have been specifically tailored to the Silicon Graphics environment.

Benefits

The primary benefits of optimization are faster running programs and smaller object code size. However, the optimizer can also speed up development time. For example, your coding time can be reduced by leaving it up to the optimizer to relate programming details to execution time efficiency. You can focus on the more crucial global structure of your program. Moreover, programs often yield optimizable code sequences regardless of how well you write your source program.

Optimization and Debugging

Optimize your programs only when they are fully developed and debugged. The optimizer may move operations around so that the object code does not correspond to the source code. These changed sequences of code can create confusion when using a debugger.

Loop Optimization

Optimizations are most useful in program areas that contain loops. The optimizer moves loop−invariant code sequences outside loops so that they are performed only once instead of multiple times. Apart from loop−invariant code, loops often contain loop−induction expressions that can be replaced with simple increments. In programs composed of mostly loops, global optimization can often reduce the running time by half.

Consider the source code below.

```
void left (a, distance)
      char a[];
      int distance;
{
      int j, length;
      length = strlen(a) - distance;
      for (j = 0; j < length; j++)
          a[j] = a[j + distance];
}
```

The following code samples show the unoptimized and optimized code produced by the compiler. The optimized version (compiled with the *-O* option) contains fewer total instructions and fewer instructions that reference memory. Wherever possible, the optimizer replaces load and store instructions (which reference memory) with the faster computational instructions that perform operations only in registers.

Unoptimized Code

The loop is 13 instructions long and uses eight memory references.

```
#   8           for (j=0; j<length; j++)
           sw     $0, 36($sp)   # j = 0
           ble    $24, 0, $33   # length >= j
$32:
#   9           a[j] = a[j+distance];
           lw     $25, 36($sp)  #  j
           lw     $8, 44($sp)   # distance
           addu   $9, $25, $8   # j+ distance
           lw     $10, 40(4sp)  # address of a
           addu   $11, $10, $25 # address of a[j+distance]
           lbu    $12, 0($11)   # a[j+distance]
```

```
           addu   $13, $10, $25 # address of a[j]
           sb     $12, 0($13)   # a[j]
           lw     $14, 36($sp)  # j
           addu   $15, $14, 1   # j+1
           sw     $15, 36($sp)  # j++
           lw     $3, 32($sp)   #  length
           blt    $15, $3, $32  # j < length
$33:
```

Optimized Code

The loop is six instructions long and uses two memory references.

```
#   8           for (j=0; j<length; j++)
           move   $5,$0         # j = 0
           ble    $4, 0, $33    # length >= j
           move   $2, $16       # address of a[j]
           addu   $6, $16, $1   # address of a[j+distance]
$32:
#   9           a[j] = a[j+distance];
           lbu    $3, 0($6)     # a[j+distance]
           sb     $3, 0($2)     # a[j]
           addu   $5, $5, 1     # j++
           addu   $2, $2, 1     # address of next a[j]
           addu   $6, $6, 1     # address of next a[j+distance]
           blt    $5, $4, $32   # j < length
$33:                          # address of next a[j+distance]
```

Loop Unrolling

The optimizer performs loop unrolling to improve performance in two ways:

- Reduces loop overhead.
- Increases work performed in the loop body allowing more opportunity for optimization and register usage.

For example, the Fortran loop:

```
do i=1,100
     sum = sum + a(i)*b(i)
enddo
```

when unrolled four times looks like

```
do i=1,100,4
     sum = sum + a(i)*b(i)
     sum = sum + a(i+1)*b(i+1)
     sum = sum + a(i+2)*b(i+2)
     sum = sum + a(i+3)*b(i+3)
enddo
```

The unrolled version runs much faster than the original. Most of the increase in execution speed is because the multiplication and addition operations are overlapped.

The optimizer performs a similar sort of transformation, although the optimizer does this on its own internal representation of the program, not by rewriting the original source code.

Note: If the number of iterations of the loop is not an exact multiple of the unrolling factor (or if the number of iterations is unknown), the optimizer still performs this transformation even though the result is more complicated than the original code.

Register Allocation

The Silicon Graphics architecture emphasizes the use of registers. Therefore, register usage has a significant impact on program performance. For example, fetching a value from a register is significantly faster than fetching a value from RAM. The optimizer must therefore make the best possible use of registers.

The optimizer allocates registers for the most suitable data items, taking into account their frequency of use and their locations in the program structure. In addition, the optimizer assigns values to registers in such a way as to minimize shifting around within loops and during procedure invocations.

Optimizing Separate Compilation Units

The optimizer processes one procedure at a time. Large procedures offer more opportunities for optimization, since more interrelationships are exposed in terms of constructs and regions. However, large procedures require more time to optimize than smaller ones.

The *uld* and *umerge* phases of the compiler permit global optimization among code from separate files (or "modules") in the same compilation. Traditionally, program modularity restricted the optimization of code to a single compilation unit at a time rather than over the full breadth of the program. For example, it was impossible to fully optimize calling code along with the procedures called if those procedures resided in other modules.

The *uld* and *umerge* phases of the compiler system overcome this deficiency. The *uld* phase links multiple modules into a single unit. Then, *umerge* orders the procedures for optimal processing by the global optimizer, *uopt*.

Optimization Options

Invoke the optimizer by specifying a compiler driver, such as *cc*(1), with any of the options listed in **Table 4–5**.

Table 4–5 Optimization Options

Option	Result
-O0	No optimization. Prevents all optimizations, including the minimal optimization normally performed by the code generator and assembler. *uld*, *umerge*, and *uopt* are bypassed, and the assembler bypasses certain optimizations it normally performs.
-O1	(Default) The assembler and code generator perform as many optimizations as possible without affecting compile time performance. Bypasses *uld*, *umerge*, and *uopt*. However, the code generator and the assembler perform basic optimizations in a more limited scope.
-O2	Specifies global optimization. Optimizes within the bounds of individual compilation units. This option executes the global optimizer (*uopt*) phase. *uld* and *umerge* are bypassed, and only the *uopt* phase executes. It performs optimization only within the bounds of individual compilation units.
-O3	This option cannot be used to compile with DSOs; it is only available for non–shared programs. The *-non_shared* option must therefore be used whenever *-O3* is used. *-O3* specifies using all optimizations, including procedure inlining. This option must precede all source file arguments. It creates a ucode object file, which remains a *.u* file, for each source file. The runtime start–up routine, runtime libraries, and ucode versions of the runtime libraries are linked, as well as newly created ucode object files and any ucode object files specified on the command line. Procedure inlining is done on the resulting linked file. This file is then compiled as usual into an executable. The *uld* and *umerge* phases process the output from the compilation phase of the

compiler, which produces symbol table information and the program text in an internal format called *ucode*. The *uld* phase combines all the ucode files and symbol tables, and passes control to *umerge*. *umerge* reorders the ucode for optimal processing by *uopt*. Upon completion, *umerge* passes control to *uopt*, which performs global optimizations on the program.

Note: Refer to the applicable *cc*(1), *pc*(1), or *f77*(1) manual pages for details on the *-O3* option and the input and output files related to this option.

Figure 4–3 shows the major processing phases of the compiler and how the compiler *-On* option determines the execution sequence.

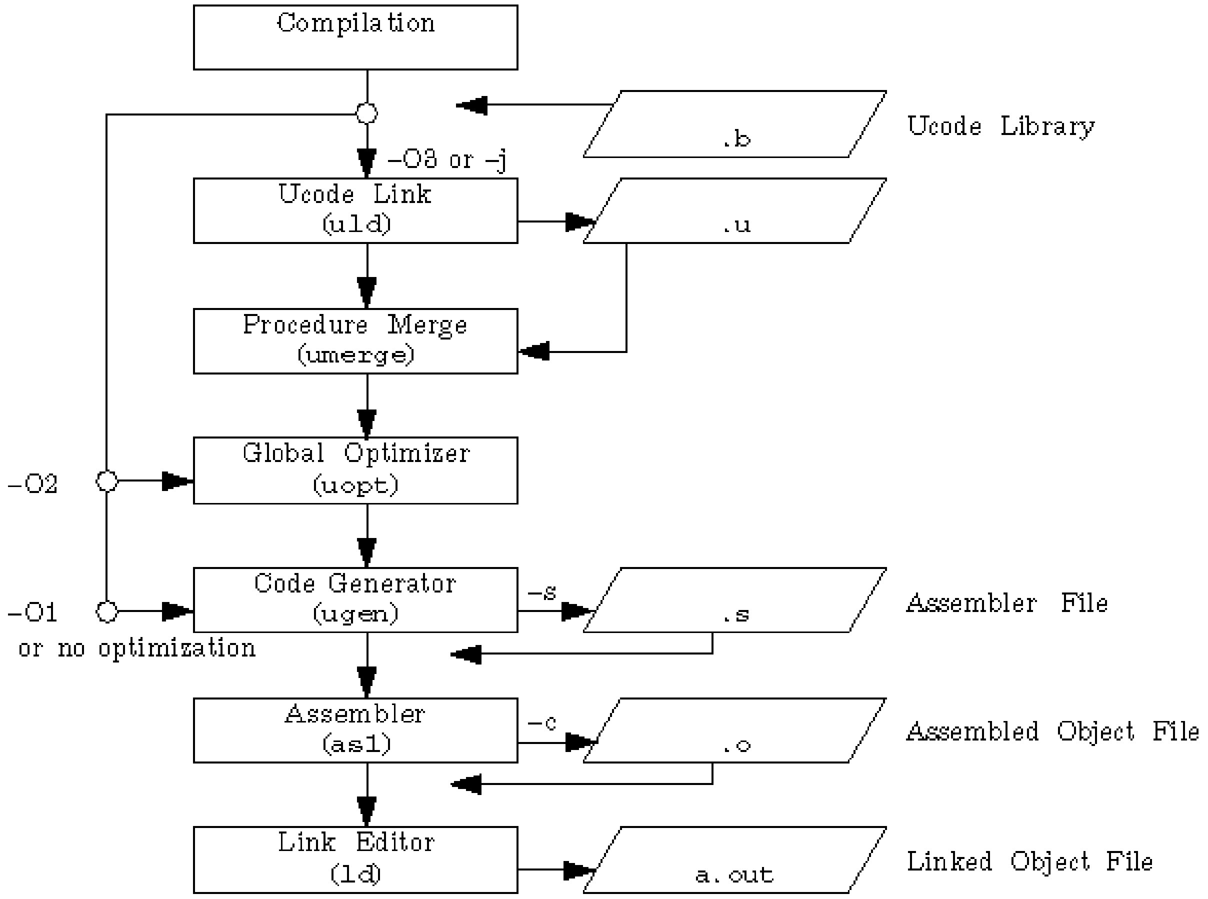

Figure 4–3 Optimization Phases of the Compiler

Full Optimization

This section provides examples of full optimization using the *-O3* option. Although the examples are in C, you can substitute the C files and driver command for another source language. The following examples assume that the program *foo* consists of three files: *a.c*, *b.c*, and *c.c*.

To perform procedure merging optimizations (*-O3*) on all three files, enter the following:

```
IRIS% cc –O3 –non_shared –o foo a.c b.c c.c
```

If you normally use the *-c* option to compile the *.o* object file, follow these steps:

1. Compile each file separately using the *-j* option by typing in the following:

```
IRIS% cc -j a.c
IRIS% cc -j b.c
IRIS% cc -j c.c
```

The *-j* option produces a *.u* file (the standard compiler front−end output made up of ucode; ucode is an internal language used by the compiler). None of the remaining compiling phases are executed, as illustrated in **Figure 4−4** .

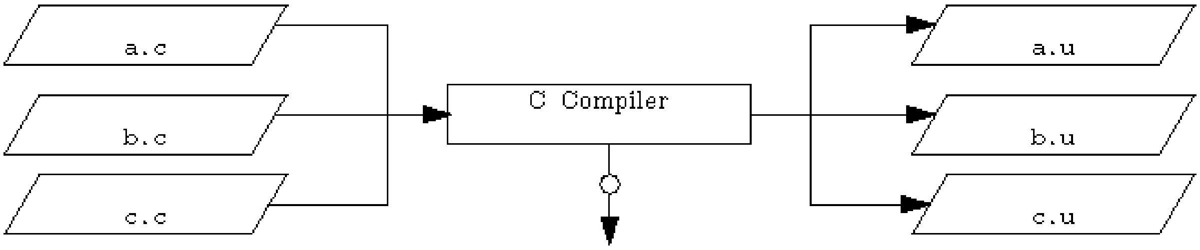

Figure 4−4 Compiling with the *-j* Option

2. Enter the following statement to perform optimization and complete the compilation process.

```
IRIS% cc -O3 -non_shared -o foo a.u b.u c.u
```

Figure 4−5 illustrates the results of executing the above statement.

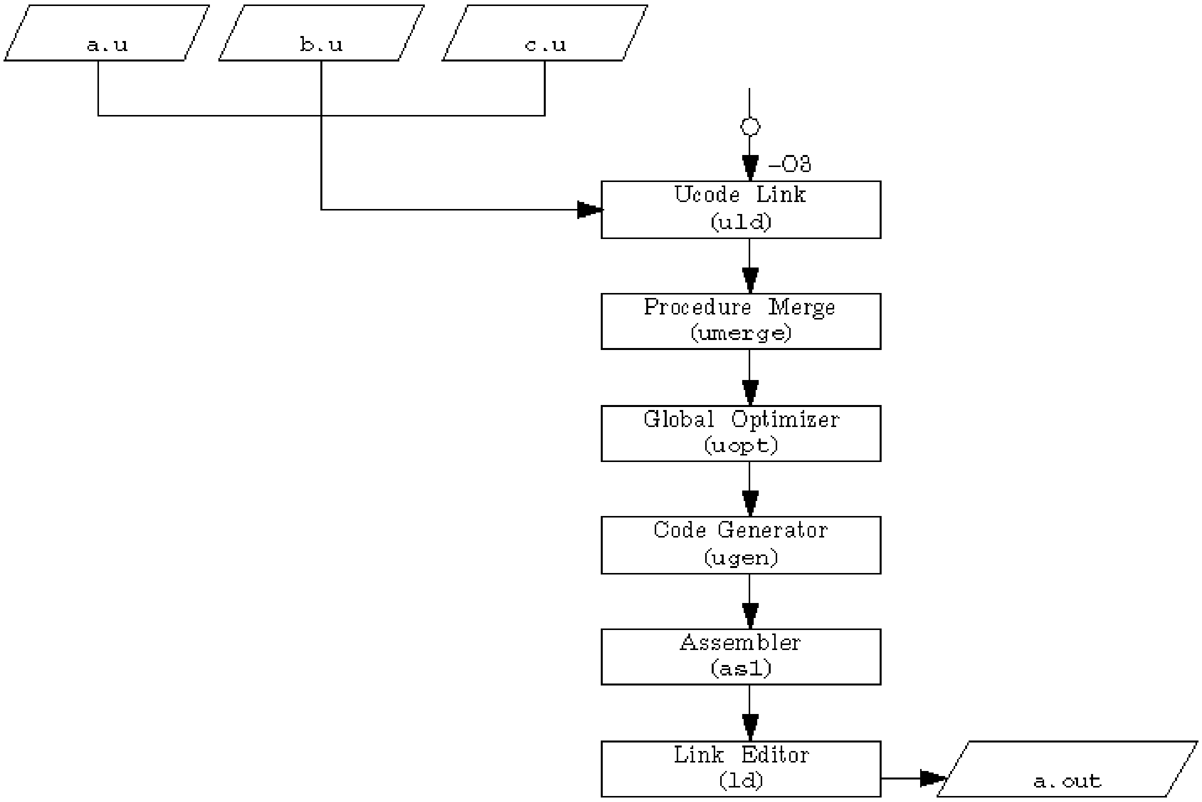

Figure 4−5 Executing Full Optimization

Optimizing Frequently Used Modules

Compiling and optimizing frequently used modules reduces the compile and optimization time required when the modules are called.

The following procedure explains how to compile two frequently used modules, *b.c* and *c.c*, while retaining all the necessary information to link them with future programs; *future.c* represents one such program.

1. Compile *b.c* and *c.c* separately by entering the following statements:

   ```
   IRIS% cc -j b.c
   IRIS% cc -j c.c
   ```

   The *-j* option causes the front end (first phase) of the compiler to produce two ucode files: *b.u* and *c.u* .

2. Using an editor, create a file containing the external symbols in *b.c* and *c.c* to which *future.c* will refer. Each symbolic name must be separated by at least one blank. Consider the skeletal contents of *b.c* and *c.c*:

   ```
   File b.c               File c.c
   foo()                  x()
      {                      {
       .                      .
       .                      .
      }                      }

   bar()                  help()
      {                      {
       .                      .
       .                      .
       zot()                 }
        {
          .                struct
          .                  {
        }                      .
                               .
       struct                } ddata;
        {
          .                y()
          .                  {
        } work;                .
      }                        .
                             }
   ```

   In this example, *future.c* calls or references only *foo*, *bar*, *x*, *ddata*, and *y* in the *b.c* and *c.c* procedures. A file (named *extern* for this example) must be created containing the following symbolic names:

   ```
   foo bar x ddata y
   ```

   The structure *work*, and the procedures *help* and *zot* are used internally only by *b.c* and *c.c*, and thus are not included in *extern*.

   If you omit an external symbolic name, an error message is generated (see Step 4 below).

3. Optimize the *b.u* and *c.u* modules (created in Step 1) using the *extern* file (created in Step 2) as follows:

   ```
   IRIS% cc -O3 -non_shared -kp extern b.u c.u -o keep.o
   ```

   In the *-kp* option, *k* indicates that the link editor option *-p* is to be passed to the ucode loader.

   Figure 4–6 illustrates Step 3.

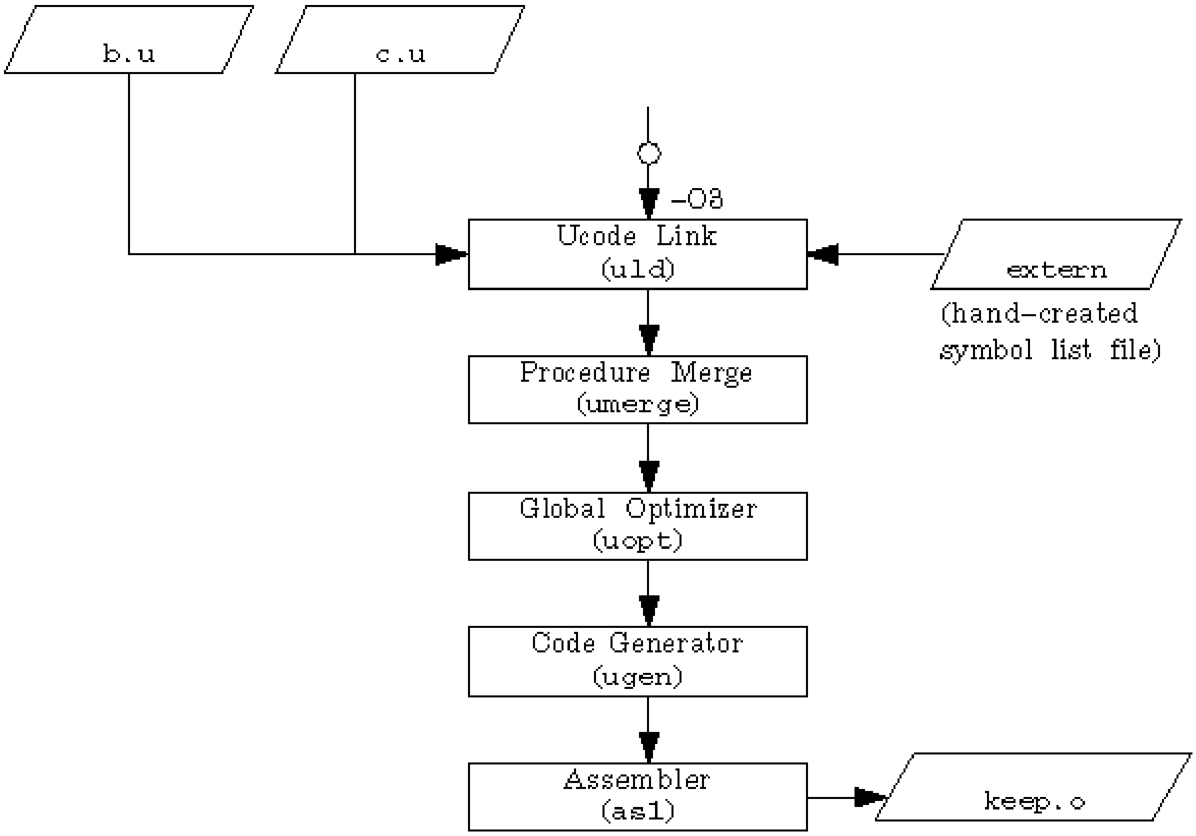

Figure 4–6 Optimization Process

4. Create a ucode file and an optimized object code file (*foo*) for *future.c* as follows:

```
IRIS% cc -j future.c
IRIS% cc -O3 -non_shared future.u keep.o -o foo
```

If the following message appears it means that the code in *future.c* is using a symbol from the code in *b.c* or *c.c* that was not specified in the file *extern* (go to Step 5 if this message appears.)

```
zot: multiply defined hidden external (should have been preserved)
```

5. Include *zot*, which the message indicates is missing, in the file *extern* and recompile as follows:

```
IRIS% cc -O3 -non_shared -kp extern b.u c.u -o keep.o
IRIS% cc -O3 -non_shared future.u keep.o -o foo
```

Building a Ucode Object Library

Building a ucode object library is similar to building a *coff*(5) object library. First, compile the source files into ucode object files using the compiler driver option *-j* and using the archiver just as you would for *coff* object libraries.

Using the above example, to build a ucode library (*libfoo.b*) of a source file, enter the following:

```
IRIS% cc -j a.c
IRIS% cc -j b.c
IRIS% cc -j c.c
IRIS% ar crs libfoo.b a.u b.u c.u
```

Conventional names exist for ucode object libraries (*lib*name.*b*) just as they do for *coff* object libraries (*lib* name.*a*).

Using Ucode Object Libraries

Using ucode object libraries is similar to using *coff*(5) object files. To load from a ucode library, specify a *-klname* option to the compiler driver or the ucode loader. For example, to load the file created in the previous example from the ucode library (assuming *libfoo.a* was placed in the */usr/lib* directory), enter the following:

```
IRIS% cc -O3 -non_shared file1.u file2.u -klfoo -o output
```

Remember that libraries are searched as they are encountered on the command line, so the order in which you specify them is important. If a library is made from both assembly and high−level language routines, the ucode object library contains code only for the high−level language routines. The library does not contain all the routines, as does a *coff* object library or a DSO. In this case, specify to the ucode loader first the ucode object library and then the *coff* object library or DSO to ensure that all modules are loaded from the proper library.

If the compiler driver is to perform both a ucode load step and a final load step, the object file created after the ucode load step is placed in the position of the first ucode file specified or created on the command line in the final load step.

Improving Global Optimization

This section contains coding hints recommended to increase optimizing opportunities for the global optimizer (*uopt*). Apply these recommendations to your code whenever possible.

C and Fortran Programs

The following suggestion applies to both C and Fortran programs:

Do not use indirect calls (calls that use routines or pointers to functions as arguments). Indirect calls cause unknown side effects (that is, they change global variables) that can reduce the amount of optimization possible.

C Programs Only

The following suggestions apply to C programs only:

Return values. Use functions that return values instead of pointer parameters.

Do while. When possible, use *do while* instead of *while* or *for*. For *do while*, the optimizer does not have to duplicate the loop condition in order to move code from within the loop to outside the loop.

Unions. Avoid unions that cause overlap between integer and floating point data types. The optimizer does not assign such fields to registers.

Use local variables. Avoid global variables. In C programs, declare any variable outside of a function as static, unless that variable is referenced by another source file. Minimizing the use of global variables increases optimization opportunities for the compiler.

Value parameters. Pass parameters by value instead of passing by reference (pointers) or using global variables. Reference parameters have the same degrading effects as the use of pointers (see below).

Pointers and aliasing. You can often avoid aliases by introducing local variables to store the values obtained from dereferenced pointers. Indirect operations and calls affect dereferenced values, but do not affect local variables. Therefore, local variables can be kept in registers. The following example shows how the proper placement of pointers and the elimination of aliasing produces better code.

Example

In this example, because the statement **p++ = 0* might modify *len*, the compiler cannot place *len* in a register for optimal performance. Instead, the compiler must load it from memory on each pass through the loop.

Source Code

```
int len = 10;
char a[10];
void
zero()
{
     char *p;
     for (p= a; p != a + len; ) *p++ = 0;
}
```

Generated Assembly Code

```
#8  for (p = a; p!= a + len; )  # p++ = 0;
         move   $2, $4
         lw     $3, len
         addu   $24, $4, $3
         beq    $24  $4  $33      # a + len != a
$32:
         sb     $0, 0($2)         # *p = 0
         addu   $2, $2, 1         # p++
         lw     $25, len
         addu   $8, $4, $25
         bne    $8, $2, $32       # len + a != p
$33:
```

You can increase the efficiency of this example by using subscripts instead of pointers, or by using local variables to store unchanging values.

Using subscripts instead of pointers. Using subscripts in the procedure *azero* (as shown below) eliminates aliasing. The compiler keeps the value of *len* in a register, saving two instructions. It still uses a pointer to access *a* efficiently, even though a pointer is not specified in the source code.

Source Code

```
void azero()
{
      int i;
      for ( i = 0; i != len; i++ )
      a[i] = 0; }
```

Generated Assembly Code

```
         for (i = 0; i != len; i++ ) a[i] = 0;
         move   $2, $0          # i = 0
         beq    $4, 0, $37      # len != a
         la     $5, a
$36:
         sb     $0, 0($5)       # *a = 0
         addu   $2, $2, 1       # i++
         addu   $5, $5, 1       # a++
         bne    $2, $4, $36     # i != len
$37:
```

Using local variables. Specifying *len* as a local variable or formal argument (as shown below) prevents aliasing and lets the compiler to *len* in a register.

Source Code

```
char a[10];
```

```
void lpzero(len)
      int len;
{
      char *p;
      for (p = a; p != a + len; ) *p++ = 0;
}
```

Generated Assembly Code

```
#8  for (p = a; p!= a + len; )  # p++ = 0;
         move   $2, $6
         addu   $5, $6, $4
         beq    $5, $6  $33       # a + len != a
$32 :
         sb     $0, 0($2)         # *p = 0
         addu   $2, $2, 1         # p++
         bne    $5, $2, $32       # a + len != p
$33:
```

In the previous example, the compiler generates slightly more efficient code for the second method.

Write straightforward code. For example, do not use ++ and –– operators within an expression. Using these operators for their values, rather than for their side–effects, often produces bad code. For example, the following code uses the value of *n––* as a condition for the *while* loop, which is a convoluted way of performing the loop while *n* is non–zero:

```
while (n--) {
      ...
}
```

In the following code the loop is performed when *n* is non–zero:

```
while (n != O) {
      n--;
      ...
}
```

Use register declarations liberally. The compiler automatically assigns variables to registers. However, specifically declaring a *register* type lets the compiler make more aggressive assumptions when assigning register variables.

Addresses. Avoid taking and passing addresses (& values). Using addresses creates aliases, makes the optimizer store variables from registers to their home storage locations, and significantly reduces optimization opportunities that would otherwise be performed by the compiler.

VARARG/STDARG. Avoid functions that take a variable number of arguments. The optimizer saves all parameter registers on entry to VARARG or STDARG functions.

Ada® Programs

This suggestion applies to Ada programs:

Use of pragma inline. Use pragma inline to inline short subroutines and avoid the overhead associated with procedure calls.

Improving Other Optimization

The global optimizer processes programs only when you specify the *-O2* or *-O3* option at compilation. However, the code generator and assembler phases of the compiler always perform certain optimizations (certain assembler optimizations are bypassed when you specify the *-O0* option at compilation).

This section contains coding hints that increase optimizing opportunities for the other passes of the compiler.

C and Fortran Programs

The following suggestions apply to both C and Fortran programs:

- Use tables rather than *if–then–else* or *switch* statements. For example:

  ```
  /* OK: */
  if ( i == 1 ) c = '1';
        else c = '0';
  /* More efficient: */
  c = "01"[i];
  ```

- As an optimizing technique, the compiler puts the first four parameters of a parameter list into registers where they remain during execution of the called routine. Therefore, always declare as the first four parameters those variables that are most frequently manipulated in the called routine.
- Use word–size variables instead of smaller ones if enough space is available. This practice can take more space, but it is more efficient.

C Programs Only

The following suggestions apply to C programs only:

- Rely on *libc.so* functions (for example, *strcpy*, *strlen*, *strcmp*, *bcopy*, *bzero*, *memset*, and *memcpy*). These functions were carefully coded for efficiency.
- Use the *unsigned* data type for variables wherever possible (see next bulleted item for an exception to this rule, though). There are two reasons for this. First: since the compiler knows such a variable is always greater than or equal to zero, it performs optimizations that would not otherwise be possible. Second: the compiler generates fewer instructions for multiplying and dividing unsigned numbers by a power of two. Consider the following example:

  ```
  int i;
  unsigned j;
  ...
  return i/2 + j/2;
  ```

  The compiler generates six instructions for the signed *i/2* operation:

  ```
  000000   20010002 li       r1,2
  000004   0081001a div      r4,r1
  000008   14200002 bne      r1,r0,0x14
  00000c   00000000 nop
  000010   03fe000d break    1022
  000014   00001812 mflo     r3
  ```

  The compiler generates only one instruction for the unsigned *j/2* operation:

  ```
  000018   0005c042 srl      r24,r5,1   # j / 2
  ```

  In this example, *i/2* is an expensive expression, while *j/2* is inexpensive.
- Use a signed data type, or cast to a signed data type, for any variable which must be converted to floating–point.

  ```
  double d;
  unsigned int u;
  int i;
  /* fast */ d = i;
  /* fast */ d = (int)u;
  /* slow */ d = u;
  ```

  Converting an unsigned type to floating–point takes significantly longer than converting signed types to floating–point; additional software support must be generated in the instruction stream for the former case.

Chapter 5
Inter–Process Communication

The term "*Inter–Process Communication*" (*IPC*) describes any method of sending data from one running process to another. IPC is commonly used to allow processes to cooperate—for instance, to let two subprograms use the same data areas in memory without interfering with each other—or to make data acquired by one process available to others. There are a wide variety of IPC mechanisms in common use, each intended for a different purpose.

Types of Inter–Process Communication Available

IRIX supports several types of IPC. Standard AT&T System V Release 4 IPC is available for making code portable. However, its implementation is fundamentally different from (and slower than) that of the IRIX–specific IPC also provided with IRIX. BSD socket–based IPC is supported for compatibility and, to allow IPC across a network, between processes running on different machines.

Do not mix the various types of IPC in a given program. Use System V IPC—based on a mechanism called keys—for code that must comply with the MIPS ABI, code that needs to be portable, or code that you're porting from another System V operating system. Use arena–based IRIX IPC for applications that require speed, ease of implementation, or multiprocessing ability. Socket–based IPC is necessary only for code being ported from or to a BSD system, and for network IPC.

This chapter describes the available types of System V and IRIX IPC, and provides examples of each. Since there are many ways to accomplish any given task or requirement, keep in mind that the example programs were written for clarity and not for program efficiency.

System V IPC

System V IPC Overview

System V IPC comprises three inter–process communication mechanisms:

- *Messages* allow processes to send and receive buffers full of data.
- *Semaphores* allow processes to turn on and off a set of flags.
- *Shared memory* gives multiple processes access to the same data area in memory.

System V Messages

The message mechanism allows processes to exchange data stored in buffers. This data is transmitted between processes in discrete units called messages. Processes using this type of IPC can perform two operations: sending messages and receiving messages.

Before a process can send or receive messages, the process must request that the operating system generate a new *message queue* (the mechanism used to control and keep track of messages), and an associated data structure. A process makes this request by using the *msgget()* system call. The requesting process becomes the owner and creator of the resulting message queue, and specifies the initial operation permissions for all processes that might use that queue (including itself). Subsequently, the owning process can relinquish ownership or change the operation permissions using the *msgctl()* system call. However, the creator remains the creator as long as the queue exists. Other processes with permission can use *msgctl()* to perform various other control functions, as described in **"Controlling Message Queues: msgctl()."**

A process that is attempting to send a message can suspend execution temporarily in order to wait until the process that is to receive the message is ready; similarly, a receiving process can suspend execution until the sending process is ready. Message operations that suspend execution in this fashion are called "blocking message operations." A process that specifies that its execution is not to be suspended—that is, a process that does not wait for communication if such is not immediately available—is performing a "nonblocking message operation."

A blocking message operation can be told to suspend a calling process until one of three conditions occurs:

- The operation is successful.
- The operation receives a signal.
- The message queue is removed.

To request a message operation, the calling process passes arguments to a system call, and the system call attempts to perform its function. If the system call is successful, it returns its results. Otherwise, it returns a known error code (−1), and an external error variable, *errno*, is set accordingly.

Message Queues

Before a message can be sent or received, a uniquely identified message queue and data structure must be created. The unique identifier created is called the message queue identifier (*msqid*); it is used to reference the associated message queue and data structure.

The message queue is used to store header information about each message that is being sent or received. This information includes the following for each message:

- pointer to the next message on queue
- message type
- message text size
- message text address

Each message queue has a data structure associated with it. This data structure contains the following information for its message queue:

- operation permissions data
- pointer to first message on the queue
- pointer to last message on the queue
- current number of bytes on the queue
- number of messages on the queue
- maximum number of bytes allowed on the queue
- process identification (PID) of last message sender
- PID of last message receiver
- last message send time
- last message receive time
- last change time
- and some padding space reserved for future expansion.

The definition for the message queue structure is:

```
struct msg
{
      struct msg   *msg_next; /* ptr to next msg on queue */
      long         msg_type;  /* message type             */
      short        msg_ts;    /* message text size        */
      caddr_t      msg_spot;  /* message text map address */
};
```

It is located in the */usr/include/sys/msg.h* header file. The definition for the associated data structure, *msqid_ds*, is also located in that header file. The permissions structure that's part of the *msqid_ds* structure is based on another structure called *ipc_perm*, which is also used as a template for permissions for other forms of IPC. The *ipc_perm* data structure format can be found in the */usr/include/sys/ipc.h* header file.

Message Operations Overview

The *msgget()* system call receives an argument *msgflg*, which can be set to indicate various flags. When only the IPC_CREAT flag is set in *msgflg*, *msgget()* performs one of two tasks:

- It gets a new *msqid* and creates an associated message queue and data structure for it; or
- It returns an existing *msqid* that already has an associated message queue and data structure.

The task performed is determined by the value of the *key* argument passed to the *msgget()* system call. For the first task, if the *key* is not already in use for an existing *msqid*, a new *msqid* is returned with an associated message queue and data structure created for the *key*, unless some system−tunable parameter (such as the maximum allowable number of message queues) would be exceeded.

Instead of requesting a specific *key* number, you may indicate a *key* of value zero, which is known as the private *key* (the constant IPC_PRIVATE is defined to be zero). When you specify IPC_PRIVATE for the *key* value, a new *msqid* is always returned with an associated message queue and data structure created for it unless a system−tunable parameter would be exceeded. When the *ipcs* command is performed, the KEY field for the *msqid* is all zeros, for security reasons.

For the second task, if a *msqid* exists for the *key* specified, the value of the existing *msqid* is returned. If you do not desire to have an existing *msqid* returned, a control command (IPC_EXCL) can be specified in the *msgflg* argument passed to the system call. The details of using this system call are discussed in the **"Getting Message Queues with msgget()"** section of this chapter.

When performing the first task, the process that calls *msgget()* becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed but the creating process always remains the creator; see **"Controlling Message Queues: msgctl()."** The creator of the message queue also determines the initial operation permissions for it.

Once a uniquely identified message queue and data structure are created, message operations and message control can be used.

The available message operations are sending and receiving. System calls are provided for these operations: *msgsnd()* and *msgrcv()*, respectively. Refer to **"Operations for Messages: msgsnd() and msgrcv(),"** for details of these system calls.

Message control is done by using the *msgctl()* system call. It permits you to control the message facility in the following ways:

- to determine the associated data structure status for a message queue identifier (*msqid*)
- to change operation permissions for a message queue
- to change the size (*msg_qbytes*) of the message queue for a particular *msqid*
- to remove a particular *msqid* from the UNIX operating system along with its associated message queue and data structure

Refer to **"Controlling Message Queues: msgctl()"** for details of the *msgctl()* system call.

Getting Message Queues with msgget()

This section details the *msgget()* system call and provides an example program illustrating its use.

The synopsis in the *msgget(2)* manual page looks like this:

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/msg.h>

int msgget (key, msgflg)
key_t key;
int msgflg;
```

All of the listed include−files are located in the */usr/include/sys* directory of the UNIX operating system.

The type of the *key* parameter, *key_t*, is defined by a *typedef* in the *types.h* header file to be equivalent to an integer.

Upon successful completion, *msgget()* returns a message queue identifier. A new *msqid* with an associated message queue and data structure is provided if either of the following is true:

- *key* is equal to IPC_PRIVATE
- *key* is a unique hexadecimal integer, and IPC_CREAT is set in *msgflg*

The value passed to the *msgflg* argument must be an integer type octal value and must specify access permissions, execution modes, and control fields (commands). Access permissions determine the read/write attributes, while execution modes determine the user/group/other attributes of the *msgflg* argument. The permissions and modes are collectively referred to as "operation permissions." **Table 5−1** shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 5−1 Operation Permissions Codes

Operation Permissions	Octal Value

Read by Owner (MSG_R)	00400
Write by Owner (MSG_W)	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. For instance, if you want a message queue to be readable by its owner and both readable and writable by others, use the code value 00406 (00400 plus 00004 plus 00002). The constants MSG_R and MSG_W, defined in the *msg.h* header file, can be used instead of 00400 and 00200, respectively.

Control commands are constants defined in the *ipc.h* header file. **See Table 5−2** , which contains the names of the constants that apply to the *msgget()* system call and their values.

Table 5−2 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for *msgflg* is therefore a combination of operation permissions and control commands. To accomplish this combination, perform a bitwise OR (|) on the flags with the operation permissions. The bit positions and values for the control commands in relation to those of the operation permissions make this possible.

Two examples:

```
msqid = msgget (key, (IPC_CREAT | MSG_R));
msqid = msgget (key, (IPC_CREAT | IPC_EXCL | 0400));
```

The *msgget()* system call attempts to return a new *msqid* if either of the following conditions is true:

- *key* is equal to IPC_PRIVATE
- *key* does not already have a *msqid* associated with it, and IPC_CREAT is set in *msgflg*

To satisfy the first condition, simply pass IPC_PRIVATE as the *key* argument when calling *msgget()*:

```
msqid = msgget (IPC_PRIVATE, msgflg);
```

The second condition is satisfied if the value for *key* is not already associated with a *msqid* and a bitwise AND of *msgflg* and IPC_CREAT gives "true" (1). This means that the given *key* is not currently being used to refer to any message queue on the computer the program is running on, and that the IPC_CREAT flag is set in *msgflg*.

Note: The system−tunable parameter MSGMNI determines the maximum number of unique message queues (*msqids*) in the UNIX operating system. Attempting to exceed MSGMNI always causes a failure.

IPC_EXCL is another control flag used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a *msqid* exists for the specified *key* provided. This is necessary to prevent the process from thinking that it has received a new (unique) *msqid* when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new *msqid* is returned if the system call is successful.

Refer to the *msgget(2)* manual page for specific information about the associated data structures. The specific failure conditions, with error names, are listed there as well.

Example Program

The example program in this section (**Example 5−1**) is a menu−driven program that exercises all possible combinations of the *msgget()* system call.

From studying this program, you can observe the method of passing arguments and receiving return values. The user−written program requirements are pointed out.

This program begins (lines 4−8) by including the required header files as specified by the *msgget(2)* manual page. Note that the *errno.h* header file is included instead of declaring *errno* as an external variable; either method works.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self−explanatory. This choice of names makes the program more readable, and it is perfectly legal since the variables are local to this program. The variables in this program and their purposes are as follows:

- *key*: used to pass the value for the desired *key*
- *opperm*: used to store the desired operation permissions
- *flags*: used to store the desired control commands
- *opperm_flags*: used to store the combination from the logical ORing of the *opperm* and *flags* variables; it is then used in the system call to pass the *msgflg* argument
- *msqid*: used for returning the message queue identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and finally for the control command combinations (flags), which are selected from a menu (lines 15−32). All possible combinations are allowed even though they might not be viable. This allows observing the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the *opperm_flags* variable (lines 36−51).

The system call is made next, and the result is stored in the *msqid* variable (line 53).

Since the *msqid* variable now contains a valid message queue identifier or the error code (−1), it is tested to see if an error occurred (line 55). If *msqid* equals −1, a message indicates that an error resulted, and the external *errno* variable is displayed (lines 57, 58).

If no error occurred, the returned message queue identifier is displayed (line 62).

The example program for the *msgget()* system call follows in **Example 5−1** .

Example 5−1 *msgget()* System Call Example

```
1  /*This is a program to illustrate the capabilities of
2   *the msgget() (message−get) system call.
3   */

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/msg.h>
8  #include <errno.h>

9  /*Start of main C program*/
10 main()
11 {
12     key_t key;
13     int opperm, flags;
14     int msqid, opperm_flags;
15     /*Enter the desired key*/
16     printf("Enter the desired key in hex = ");
17     scanf("%x", &key);
18     /*Enter the desired octal operation
19       permissions.*/
20     printf("\nEnter the operation ");
21     printf("permissions in octal = ");
22     scanf("%o", &opperm);
23     /*Set the desired flags.*/
24     printf("\nEnter corresponding number to\n");
25     printf("set the desired flags:\n");
26     printf("No flags = 0\n");
27     printf("IPC_CREAT = 1\n");
```

```
    printf("IPC_EXCL = 2\n");
    printf("IPC_CREAT and IPC_EXCL = 3\n");
    printf(" Flags = ");

    /*Get the flag(s) to be set.*/
    scanf("%d", &flags);

    /*Check the values.*/
    printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
        key, opperm, flags);

    /*Incorporate the control fields (flags) with
      the operation permissions*/
    switch (flags)
    {
    case 0: /*No flags are to be set.*/
        opperm_flags = (opperm | 0);
        break;
    case 1: /*Set the IPC_CREAT flag.*/
        opperm_flags = (opperm | IPC_CREAT);
        break;
    case 2: /*Set the IPC_EXCL flag.*/
        opperm_flags = (opperm | IPC_EXCL);
        break;
    case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
        opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
    }
    /*Call the msgget() system call.*/
    msqid = msgget (key, opperm_flags);

    /*Perform the following if the call failed.*/
    if(msqid == -1)
    {
        printf ("\nThe msgget system call failed!\n");
        printf ("The error number was %d.\n", errno);
    }
    /*Return the msqid upon successful completion.*/
    else
        printf ("\nThe msqid is %d.\n", msqid);

    exit(0);
}
```

Controlling Message Queues: msgctl()

This section gives a detailed description of using the *msgctl()* system call and an example program that exercises its capabilities.

The synopsis given in the *msgctl(*2*)* manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgctl (msqid, cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

The *msgctl()* system call requires three arguments to be passed to it, and it returns an integer value.

On successful completion, it returns zero; when unsuccessful, it returns a −1.

The *msqid* variable must be a valid, non−negative, integer value. In other words, it must have already been created by using the *msgget()* system call.

The *cmd* argument can be replaced by one of the following control commands (flags):

IPC_STAT	return the status information contained in the associated data structure for the specified *msqid*, and place it in the data structure pointed to by the **buf* pointer in the user memory area.
IPC_SET	for the specified *msqid*, set the effective user and group identification, operation permissions, and the number of bytes for the message queue.
IPC_RMID	remove the specified *msqid* along with its associated message queue and data structure.

A process must have an effective user identification of OWNER/CREATOR or super user to perform an IPC_SET or IPC_RMID control command. Read permission is required to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have trouble understanding the logic manipulations in this program, read the **"Getting Message Queues with msgget()"** section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (**Example 5−2**) is a menu−driven program that exercises all possible combinations of the *msgctl()* system call.

From studying this program, you can observe the method of passing arguments and receiving return values. The user−written program requirements are pointed out.

This program begins (lines 5−9) by including the required header files as specified by the *msgctl(2)* manual page. Note in this program that *errno* is declared as an external variable, and therefore, the *errno.h* header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self−explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

uid	used to store the IPC_SET value for the effective user identification
gid	used to store the IPC_SET value for the effective group identification
mode	used to store the IPC_SET value for the operation permissions
bytes	used to store the IPC_SET value for the number of bytes in the message queue (*msg_qbytes*)
rtrn	used to store the return integer value from the system call
msqid	used to store and pass the message queue identifier to the system call
command	used to store the code for the desired control command so that subsequent processing can be performed on it
choice	used to determine which member is to be changed for the IPC_SET control command
msqid_ds	used to receive the specified message queue identifier's data structure when an IPC_STAT control command is performed
**buf*	a pointer passed to the system call; it locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set

Note that the *msqid_ds* data structure in this program (line 16) uses the data structure located in the *msg.h* header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the **buf* pointer is declared to be a pointer to a data structure of the *msqid_ds* type, it must also be initialized to contain the address of the user memory area data structure (line 17). Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid message queue identifier, which is stored in the *msqid* variable (lines 19, 20). This is required for every *msgctl()* system call.

Then the code for the desired control command must be entered (lines 21−27), and it is stored in the command variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 37, 38) and the status information returned is printed out (lines 39−46); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful (line 106), the status information of the last successful call is printed out. In addition, an error message is displayed and the *errno* variable is printed out (lines 108, 109). If the system call is successful, a message indicates this along with the message queue identifier used (lines 111−114).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 50−52). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 53−59). This code is stored in the *choice* variable (line 60). Now, depending upon the member picked, the program prompts for the new value (lines 66−95). The value is placed in the appropriate member in the user memory area data structure, and the system call is made (lines 96−98). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 100−103), and the *msqid* along with its associated message queue and data structure are removed from the UNIX operating system. Note that the **buf* pointer is not required as an argument to perform this control command, and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the *msgctl()* system call follows in **Example 5−2** .

Example 5−2 *msgctl()* System Call Example

```
1  /*This is a program to illustrate
2  *the message control, msgctl(),
3  *system call capabilities.
4  */

5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/msg.h>

10  /*Start of main C language program*/
11  main()
12  {
13    extern int errno;
14   int uid, gid, mode, bytes;
15   int rtrn, msqid, command, choice;
16   struct msqid_ds msqid_ds, *buf;
17   buf = &msqid_ds;
18   /*Get the msqid, and command.*/
19   printf("Enter the msqid = ");
20   scanf("%d", &msqid);
21   printf("\nEnter the number for\n");
22   printf("the desired command:\n");
23   printf("IPC_STAT = 1\n"); 24 printf("IPC_SET = 2\n");
25   printf("IPC_RMID = 3\n");
26   printf("Entry = ");
27   scanf("%d", &command);
28   /*Check the values.*/
29   printf (\nmsqid =%d, command = %\n", 30 msqid, command);
```

```
 switch (command)
 {
 case 1: /*Use msgctl() to duplicate
    the data structure for
    msqid in the msqid_ds area pointed
    to by buf and then print it out.*/
  rtrn = msgctl(msqid, IPC_STAT,
   buf);
  printf ("\nThe USER ID = %d\n",
   buf->msg_perm.uid);
  printf ("The GROUP ID = %d\n",
   buf->msg_perm.gid);
  printf ("The operation permissions = 0%o\n",
   buf->msg_perm.mode);
  printf ("The msg_qbytes = %d\n",
   buf->msg_qbytes);
  break;
 case 2: /*Select and change the desired
    member(s) of the data structure.*/
  /*Get the original data for this msqid
    data structure first.*/
  rtrn = msgctl(msqid, IPC_STAT, buf);
  printf("\nEnter the number for the\n");
  printf("member to be changed:\n");
  printf("msg_perm.uid = 1\n");
  printf("msg_perm.gid = 2\n");
  printf("msg_perm.mode = 3\n");
  printf("msg_qbytes = 4\n");
  printf("Entry = ");
  scanf("%d", &choice);
  /*Only one choice is allowed per
  pass as an illegal entry will
   cause repetitive failures until
  msqid_ds is updated with
   IPC_STAT.*/
  switch(choice){
case 1:
   printf("\nEnter USER ID = ");
   scanf ("%d", &uid);
   buf->msg_perm.uid = uid;
   printf("\nUSER ID = %d\n",
    buf->msg_perm.uid);
   break;
  case 2:
   printf("\nEnter GROUP ID = ");
   scanf("%d", &gid);
   buf->msg_perm.gid = gid;
   printf("\nGROUP ID = %d\n",
    buf->msg_perm.gid);
   break;
  case 3:
   printf("\nEnter MODE = ");
   scanf("%o", &mode);
   buf->msg_perm.mode = mode;
   printf("\nMODE = 0%o\n",
    buf->msg_perm.mode);
   break;
  case 4:
```

```
    printf("\nEnter msq_bytes = ");
    scanf("%d", &bytes);
    buf->msg_qbytes = bytes;
    printf("\nmsg_qbytes = %d\n",
     buf->msg_qbytes);
    break;
   }
   /*Do the change.*/
   rtrn = msgctl(msqid, IPC_SET,
    buf);
   break;
   case 3: /*Remove the msqid along with its
       associated message queue
       and data structure.*/
    rtrn = msgctl(msqid, IPC_RMID, NULL);
   }
   /*Perform the following if the call is unsuccessful.*/
   if(rtrn == -1)
   {
    printf ("\nThe msgctl system call failed!\n");
    printf ("The error number = %d\n", errno);
   }
   /*Return the msqid upon successful completion.*/
   else
    printf ("\nMsgctl was successful for msqid = %d\n",
     msqid);
   exit (0);
}
```

Operations for Messages: msgsnd() and msgrcv()

This section describes the *msgsnd()* and *msgrcv()* system calls and presents an example program that exercises all of their capabilities.

The synopsis found in the *msgop(2)* manual page, which describes both *msgsnd()* and *msgrcv()*, is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgsnd(int msqid, const void *msgp,
   size_t msgsz, int msgflg);

int msgrcv(int msqid, void *msgp,
   size_t msgsz, long msgtyp, int msgflg);
```

Sending a Message

The *msgsnd()* system call requires four arguments to be passed to it. It returns an integer value.

Upon successful completion, a zero value is returned; when unsuccessful, *msgsnd()* returns a −1.

The *msqid* argument must be a valid, non−negative, integer value. In other words, it must have already been created by using the *msgget()* system call.

The *msgp* argument is a pointer to a structure in the user memory area that contains the type of the message and the message to be sent.

The *msgsz* argument specifies the length of the character array in the data structure pointed to by the *msgp* argument. This is the length of the message. The maximum size of this array is determined by the MSGMAX system tunable parameter.

The *msg_qbytes* data structure member can be lowered from MSGMNB by using the *msgctl()* IPC_SET control command, but only the super user can raise it afterwards.

The *msgflg* argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (*msgflg* & IPC_NOWAIT = 0). This occurs if the total number of bytes allowed on the specified message queue are in use (*msg_qbytes* or MSGMNB), or the total system−wide number of messages on all queues is equal to the system imposed limit (MSGTQL). If the IPC_NOWAIT flag is set, the system call fails and returns −1.

Further details of this system call are discussed in the example program for it. If you don’t understand the logic manipulations in this program, read the **"Getting Message Queues with msgget()"** section of this chapter; it goes into more detail than would be practical to do for every system call.

Receiving Messages

The *msgrcv()* system call requires five arguments to be passed to it, and it returns an integer value.

Upon successful completion, a value equal to the number of bytes received is returned and when unsuccessful a −1 is returned.

The *msqid* argument must be a valid, non−negative, integer value. In other words, it must have already been created by using the *msgget()* system call.

The *msgp* argument is a pointer to a structure in the user memory area that receives the message type and the message text.

The *msgsz* argument specifies the length of the message to be received. If its value is less than the message in the array, an error can be returned if desired; see the *msgflg* argument.

The *msgtyp* argument is used to pick the first message on the message queue of the particular type specified. If it is equal to zero, the first message on the queue is received; if it is greater than zero, the first message of the same type is received; if it is less than zero, the lowest type that is less than or equal to its absolute value is received.

The *msgflg* argument allows the "blocking message operation" to be performed if the IPC_NOWAIT flag is not set (*msgflg* & IPC_NOWAIT = 0); this would occur if there is not a message on the message queue of the desired type (*msgtyp*) to be received. If the IPC_NOWAIT flag is set, the system call fails immediately when a message of the desired type is not on the queue. *msgflg* can also specify that the system call fails if the message is longer than the size to be received; this is done by not setting the MSG_NOERROR flag in the *msgflg* argument (*msgflg* & MSG_NOERROR = 0). If the MSG_NOERROR flag is set, the message is truncated to the length specified by the *msgsz* argument of *msgrcv()*.

Further details of this system call are discussed in the example program for it. If you don’t understand the logic manipulations in this program, read the **"Getting Message Queues with msgget()"** section of this chapter; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (**Example 5−3**) is a menu−driven program that exercises all possible combinations of the *msgsnd()* and *msgrcv()* system calls.

From studying this program, you can observe the method of passing arguments and receiving return values. The user−written program requirements are pointed out.

This program begins (lines 5−9) by including the required header files as specified by the *msgop(2)* manual page. Note that in this program, *errno* is declared as an external variable. Therefore, the *errno.h* header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self−explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program.

The variables declared for this program and their purposes are as shown in **Table 5−3** :

Table 5−3 Variables Used in the *msgop()* Example Program

Variable	Purpose
sndbuf	used as a buffer to contain a message to be sent

	(line 13); it uses the *msgbuf1* data structure as a template (lines 10−13) The *msgbuf1* structure (lines 10−13) is almost an exact duplicate of the *msgbuf* structure contained in the *msg.h* header file. The only difference is that the character array for *msgbuf1* contains the maximum message size (MSGMAX) for the workstation where in *msgbuf* it is set to one to satisfy the compiler. For this reason *msgbuf* cannot be used directly as a template for the user−written program. It is there so you can determine its members.
rcvbuf	used as a buffer to receive a message (line 13); it uses the *msgbuf1* data structure as a template (lines 10−13)
**msgp*	used as a pointer (line 13) to both the *sndbuf* and *rcvbuf* buffers
i	used as a counter for inputting characters from the keyboard, storing them in the array, and keeping track of the message length for the *msgsnd()* system call; it is also used as a counter to output the received message for the *msgrcv()* system call
c	used to receive the input character from the *getchar()* function (line 50)
flag	used to store the code of IPC_NOWAIT for the *msgsnd()* system call (line 61)
flags	used to store the code of the IPC_NOWAIT or MSG_NOERROR flags for the *msgrcv()* system call (line 117)
choice	used to store the code for sending or receiving (line 30)
rtrn	used to store the return values from all system calls
msqid	used to store and pass the desired message queue identifier for both system calls
msgsz	used to store and pass the size of the message to be sent or received
msgflg	used to pass the value of flag for sending or the value of flags for receiving
msgtyp	used for specifying the message type for sending, or used to pick a message type for receiving

Note that a *msqid_ds* data structure is set up in the program (line 21) with a pointer that is initialized to point to it (line 22); this allows the data structure members that are affected by message operations to be observed. They are observed by using the *msgctl()* system call (with IPC_STAT) to get them so the program can print them out (lines 80−92 and lines 161−168).

The first thing the program prompts for is whether to send or receive a message. A corresponding code must be entered for the desired operation, and it is stored in the choice variable (lines 23−30). Depending upon the code, the program proceeds as in the following *msgsnd()* or *msgrcv()* sections.

msgsnd

When the code is to send a message, the *msgp* pointer is initialized (line 33) to the address of the send data structure, *sndbuf*. Next, a message type must be entered for the message; it is stored in the variable *msgtyp* (line 42), and then (line 43) it is put into the *mtype* member of the data structure pointed to by *msgp*.

The program now prompts for a message to be entered from the keyboard and enters a loop of getting and storing into the *mtext* array of the data structure (lines 48−51). This continues until an end of file is reached (for the *getchar()* function, this is a control−D immediately following a carriage return). When this happens, the size of the message is determined by adding one to the *i* counter (lines 52, 53) as it stored the message beginning in the zero array element of *mtext*. Keep in mind that the message also contains the terminating characters; therefore, the message appears to be three characters short of *msgsz*.

The message is immediately echoed from the mtext array of the *sndbuf* data structure to provide feedback (lines 54−56).

The next and final thing that must be decided is whether to set the IPC_NOWAIT flag. The program does this by requesting that 1 be entered for yes or anything else for no (lines 57−65). The result is stored in the flag variable. If a 1 is entered, IPC_NOWAIT is logically ORed with *msgflg*; otherwise, *msgflg* is set to zero.

The *msgsnd()* system call is then performed (line 69). If it is unsuccessful, a failure message is displayed along with the error number (lines 70−72). If it is successful, the returned value (which should be zero) is printed (lines 73−76).

Every time a message is successfully sent, three members of the associated data structure are updated. They are:

msg_qnum	represents the total number of messages on the message queue; it is incremented by one
msg_lspid	contains the Process Identification (PID) number of the last process sending a message; it is set accordingly
msg_stime	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) of the last message sent; it is set accordingly

These members are displayed after every successful message send operation (lines 79−92).

msgrcv

If the code specifies that a message is to be received, the program continues execution as in the following paragraphs.

The *msgp* pointer is initialized to the *rcvbuf* data structure (line 99).

Next, the message queue identifier of the message queue from which to receive the message is requested, and it is stored in *msqid* (lines 100−103).

The message type is requested, and it is stored in *msgtyp* (lines 104−107).

The code for the desired combination of control flags is requested next, and it is stored in *flags* (lines 108−117). Depending upon the selected combination, *msgflg* is set accordingly (lines 118−133).

Finally, the number of bytes to be received is requested, and it is stored in *msgsz* (lines 134−137).

The *msgrcv()* system call is performed (line 144). If it is unsuccessful, a message and error number is displayed (lines 145−148). If successful, a message indicates success, and the number of bytes returned is displayed followed by the received message (lines 153−159).

When a message is successfully received, three members of the associated data structure are updated; they are described as follows:

msg_qnum	contains the number of messages on the message queue; it is decremented by one
msg_lrpid	contains the process identification (PID) of the last process receiving a message; it is set accordingly
msg_rtime	contains the time in seconds since January 1, 1970, Greenwich Mean Time (GMT) that the last process received a message; it is set accordingly

The example program for the *msgop()* system calls is shown in **Example 5−3**.

Example 5−3 *msgsnd()* and *msgrcv()* System Call Example

```
1  /*This is a program to illustrate
2   * the message operations, msgop(),
3   * system call capabilities: msgsnd() and msgrcv().
4   */
5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
```

```
#include <sys/ipc.h>
#include <sys/msg.h>
 struct msgbuf1 {
  long mtype;
  char mtext[8192];
} sndbuf, rcvbuf, *msgp;
 /*Start of main C language program*/
 main()
 {
  extern int errno;
  int i, c, flag, flags, choice;
  int rtrn, msqid, msgsz, msgflg;
  long mtype, msgtyp;
  struct msqid_ds msqid_ds, *buf;
  buf = &msqid_ds;

  /*Select the desired operation.*/
  printf("Enter the corresponding\n");
  printf("code to send or\n");
  printf("receive a message:\n");
  printf("Send = 1\n");
  printf("Receive = 2\n");
  printf("Entry = ");
  scanf("%d", &choice);

  if(choice == 1) /*Send a message.*/
  {
   msgp = &sndbuf; /*Point to user send structure.*/

   printf("\nEnter the msqid or\n");
   printf("the message queue to\n");
   printf("handle the message = ");
   scanf("%d", &msqid);

   /*Set the message type.*/
   printf("\nEnter a positive integer\n");
   printf("message type (long) for the\n");
   printf("message = ");
   scanf("%d", &msgtyp);
   msgp->mtype = msgtyp;

   /*Enter the message to send.*/
   printf("\nEnter a message: \n");

   /*A control-d (^d) terminates as
     EOF.*/

   /*Get each character of the message
     and put it in the mtext array.*/
   for(i = 0; ((c = getchar()) != EOF); i++)
    sndbuf.mtext[i] = c;

   /*Determine the message size.*/
   msgsz = i + 1;
   /*Echo the message to send.*/
   for(i = 0; i < msgsz; i++)
    putchar(sndbuf.mtext[i]);
```

```
 /*Set the IPC_NOWAIT flag if
   desired.*/
 printf("\nEnter a 1 if you want the\n");
 printf("the IPC_NOWAIT flag set: ");
 scanf("%d", &flag);
 if(flag == 1)
  msgflg |= IPC_NOWAIT;
 else
  msgflg = 0;

 /*Check the msgflg.*/
 printf("\nmsgflg = 0%o\n", msgflg);

 /*Send the message.*/
 rtrn = msgsnd(msqid, msgp, msgsz, msgflg);
 if(rtrn == -1)
 printf("\nMsgsnd failed. Error = %d\n",
    errno);
 else {
  /*Print the value of test which
   should be zero for successful.*/
  printf("\nValue returned = %d\n", rtrn);

  /*Print the size of the message
  sent.*/
  printf("\nMsgsz = %d\n", msgsz);

  /*Check the data structure update.*/
  msgctl(msqid, IPC_STAT, buf);

  /*Print out the affected members.*/

  /*Print the incremented number of
    messages on the queue.*/
  printf("\nThe msg_qnum = %d\n",
     buf->msg_qnum);
  /*Print the process id of the last sender.*/
  printf("The msg_lspid = %d\n",
    buf->msg_lspid);
  /*Print the last send time.*/
  printf("The msg_stime = %d\n",
    buf->msg_stime);
 }
}

if(choice == 2) /*Receive a message.*/
{
 /*Initialize the message pointer
 to the receive buffer.*/
 msgp = &rcvbuf;

  /*Specify the message queue which contains
     the desired message.*/
  printf("\nEnter the msqid = ");
  scanf("%d", &msqid);

  /*Specify the specific message on the queue
   by using its type.*/
  printf("\nEnter the msgtyp = ");
```

```
 printf("\nEnter the msgtyp = ");
 scanf("%d", &msgtyp);

 /*Configure the control flags for the
  desired actions.*/
 printf("\nEnter the corresponding codeen");
 printf("to select the desired flags: \n");
 printf("No flags = 0\n");
printf("MSG_NOERROR = 1\n");
 printf("IPC_NOWAIT = 2\n");
 printf("MSG_NOERROR and IPC_NOWAIT = 3\n");
 printf(" Flags = ");
 scanf("%d", &flags);

 switch(flags) {
  /*Set msgflg by ORing it with the appropriate
     flags (constants).*/
 case 0:
  msgflg = 0;
  break;
 case 1:
  msgflg |= MSG_NOERROR;
  break;
 case 2:
  msgflg |= IPC_NOWAIT;
  break;
 case 3:
  msgflg |= MSG_NOERROR | IPC_NOWAIT;
  break;
 }

 /*Specify the number of bytes to receive.*/
 printf("\nEnter the number of bytes\n");
 printf("to receive (msgsz) = ");
 scanf("%d", &msgsz);

 /*Check the values for the arguments.*/
 printf("\nmsqid =%d\n", msqid);
 printf("\nmsgtyp = %d\n", msgtyp);
 printf("\nmsgsz = %d\n", msgsz);
 printf("\nmsgflg = 0%o\n", msgflg);

 /*Call msgrcv to receive the message.*/
 rtrn = msgrcv(msqid, msgp, msgsz, msgtyp, msgflg);

 if(rtrn == -1) {
  printf("\nMsgrcv failed. ");
  printf("Error = %d\n", errno);
 }
 else {
  printf ("\nMsgctl was successful\n");
  printf("for msqid = %d\n",
   msqid);

  /*Print the number of bytes received,
    it is equal to the return
    value.*/
  printf("Bytes received = %d\n", rtrn);
```

```
   /*Print the received message.*/
   for(i = 0; i<=rtrn; i++)
    putchar(rcvbuf.mtext[i]);
  }
  /*Check the associated data structure.*/
  msgctl(msqid, IPC_STAT, buf);
  /*Print the decremented number of messages.*/
  printf("\nThe msg_qnum = %d\n", buf->msg_qnum);
  /*Print the process id of the last receiver.*/
  printf("The msg_lrpid = %d\n", buf->msg_lrpid);
  /*Print the last message receive time*/
  printf("The msg_rtime = %d\n", buf->msg_rtime);
 }
}
```

System V Semaphores

Semaphore IPC allows processes to communicate through the exchange of data items called semaphores. A single semaphore is represented as a positive integer (0 through 32,767). Semaphores are usually used to manage resources; each semaphore indicates whether or not a specific data item is currently in use (and by how many different processes). Since many applications require the use of more than one semaphore, the UNIX operating system has the ability to create sets or arrays of semaphores at one time. A semaphore set can contain one or more semaphores, up to a limit set by the system administrator. (This limit, a tunable parameter called SEMMSL, has a default value of 25.) To create a set of semaphores, use the *semget()* system call.

The process performing the *semget()* system call becomes the owner/creator of the semaphore set, determines how many semaphores are in the set, and sets the operation permissions for the set. This process can subsequently relinquish ownership of the set or change the operation permissions using the *semctl()* (semaphore control) system call.The creating process remains the creator as long as the semaphore set exists, but other processes with permission can use *semctl()* to perform other control functions.

Provided a process has alter permission, it can manipulate the semaphore set. Each semaphore within a set can be either increased or decreased with the *semop(2)* system call (see the IRIX manual pages for more information).

To increase a semaphore, pass a positive integer value of the desired magnitude to the *semop()* system call. To decrease a semaphore, pass a negative integer value of the desired magnitude.

The UNIX operating system ensures that only one process can manipulate a semaphore set at any given time. Simultaneous requests are performed sequentially in an arbitrary manner.

A process can test for a semaphore value to be greater than a certain value by attempting to decrement the semaphore by one more than that value. If the process is successful, then the semaphore value is greater than that certain value. Otherwise, the semaphore value is not. While doing this, the process can have its execution suspended (IPC_NOWAIT flag not set) until the semaphore value would permit the operation (other processes increment the semaphore), or the semaphore facility is removed.

The ability to suspend execution is called a "blocking semaphore operation." This ability is also available for a process that is testing for a semaphore to become zero or equal to zero; only read permission is required for this test, and it is accomplished by passing a value of zero to the *semop()* system call.

On the other hand, if the process is not successful and the process does not request to have its execution suspended, it is called a "nonblocking semaphore operation." In this case, the process is returned a known error code (–1), and the external *errno* variable is set accordingly.

The blocking semaphore operation allows processes to communicate based on the values of semaphores at different points in time. Remember also that IPC facilities remain in the UNIX operating system until removed by a permitted process or until the system is reinitialized.

Operating on a semaphore set is done by using the *semop()*, semaphore operation, system call.

When a set of semaphores is created, the first semaphore in the set is semaphore number zero. The last semaphore in the set is numbered one less than the number of semaphores in the set.

An array of these "blocking/nonblocking operations" can be performed on a set containing more than one semaphore. When performing an array of operations, the "blocking/nonblocking operations" can be applied to any or all of the semaphores in the set. Also, the operations can be applied in any order of semaphore number. However, no operations are done until they can all be done successfully. This requirement means that preceding changes made to semaphore values in the set must be undone when a "blocking semaphore operation" on a semaphore in the set cannot be completed successfully; no changes are made until they can all be made. For example, if a process has successfully completed three of six operations on a set of ten semaphores but is "blocked" from performing the fourth operation, no changes are made to the set until the fourth and remaining operations are successfully performed. Additionally, any operation preceding or succeeding the "blocked" operation, including the blocked operation, can specify that when all operations can be performed successfully, that the operation should be undone. Otherwise, the operations are performed and the semaphores are changed, or one "nonblocking operation" is unsuccessful and none are changed. This is commonly referred to as being "atomically performed."

The ability to undo operations requires the UNIX operating system to maintain an array of "undo structures" corresponding to the array of semaphore operations to be performed. Each semaphore operation that is to be undone has an associated adjust variable used for undoing the operation, if necessary.

Remember, any unsuccessful "nonblocking operation" for a single semaphore or a set of semaphores causes immediate return with no operations performed at all. When this occurs, a known error code (–1) is returned to the process, and the external variable *errno* is set accordingly.

System calls make these semaphore capabilities available to processes. The calling process passes arguments to a system call, and the system call attempts to perform its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (–1) is returned to the process, and the external variable *errno* is set accordingly.

Using Semaphores

Before you can use semaphores, you must request a uniquely identified semaphore set and its associated data structure with a system call. The unique identifier is called the semaphore identifier (*semid*); it is used to reference a particular semaphore set and data structure.

The semaphore set contains a predefined number of structures in an array, one structure for each semaphore in the set. The user may select the number of semaphores (*nsems*) in a set. The following members are in each instance of the structure within a semaphore set:

- semaphore text map address
- process identification (PID) performing last operation
- number of processes waiting for the semaphore value to become greater than its current value
- number of processes waiting for the semaphore value to equal zero

Each semaphore set has an associated data structure. This data structure contains information related to the semaphore set:

- operation permissions data
- pointer to first semaphore in the set
- number of semaphores in the set
- last semaphore operation time
- last semaphore change time

The definition for the semaphore set is:

```
struct sem
{
      ushort semval; /* semaphore text map address */
      short sempid; /* pid of last operation */
      ushort semncnt; /* # awaiting semval > cval */
      ushort semzcnt; /* # awaiting semval = 0 */
};
```

This definition is located in the */usr/include/sys/sem.h* header file. The structure definition for the associated semaphore data structure is also located in the same header file. Note that the *sem_perm* member of this structure uses *ipc_perm* as a template. The *ipc_perm* data structure is the same for all IPC facilities, and it is located in the */usr/include/sys/ipc.h* header file.

The *semget()* system call is used to perform two tasks when only the IPC_CREAT flag is set in the *semflg* argument that it receives:

- to get a new *semid* and create an associated data structure and semaphore set for it
- to return an existing *semid* that already has an associated data structure and semaphore set

The task performed is determined by the value of the *key* argument passed to the *semget()* system call. For the first task, if the *key* is not already in use for an existing *semid*, a new *semid* is returned with an associated data structure and semaphore set created for it, provided no system−tunable parameter would be exceeded.

Also, a provision exists for specifying a *key* of value zero, known as the private *key* (IPC_PRIVATE = 0). When specified, a new *semid* is always returned with an associated data structure and semaphore set created for it, unless a system tunable parameter would be exceeded. When the *ipcs* command is performed, the KEY field for the *semid* is all zeros.

When performing the first task, the process that calls *semget()* becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see **"Controlling Semaphores with semctl()"** for more information. The creator of the semaphore set also determines the initial operation permissions for the facility.

For the second task, if a *semid* exists for the *key* specified, the value of the existing *semid* is returned. If you do not desire to have an existing *semid* returned, a control command (IPC_EXCL) can be specified in the *semflg* argument passed to the system call. The system call fails if it is passed a value for the number of semaphores (*nsems*) that is greater than the number actually in the set. If you do not know how many semaphores are in the set, use 0 for *nsems*. The details of the *semget()* system call are discussed in **"Getting Semaphores with semget()."**

Once a uniquely−identified semaphore set and data structure are created, use semaphore operations (*semop()*) and semaphore control (*semctl()*).

Semaphores can be increased, decreased, or tested for zero. A single system call, *semop()*, is used to perform these operations. Refer to **"Operations on Semaphores: semop()"** for details on this system call.

Semaphore control is done by using the *semctl()* system call. Use control operations to control the semaphore facility in the following ways:

- to return the value of a semaphore
- to set the value of a semaphore
- to return the process identification (PID) of the last process performing an operation on a semaphore set
- to return the number of processes waiting for a semaphore value to become greater than its current value
- to return the number of processes waiting for a semaphore value to equal zero
- to get all semaphore values in a set and place them in an array in user memory
- to set all semaphore values in a semaphore set from an array of values in user memory
- to place all data structure member values and status of a semaphore set into user memory area
- to change operation permissions for a semaphore set
- to remove a particular *semid* from the UNIX operating system along with its associated data structure and semaphore set

Refer to **"Controlling Semaphores with semctl()"** for details of the *semctl()* system call.

Getting Semaphores with semget()

This section contains a detailed description of using the *semget()* system call along with an example program illustrating its use.

The synopsis found in the *semget(2)* manual page is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

key_t is declared by a *typedef* in the *types.h* header file to be an integer.

The integer returned from this system call upon successful completion is the semaphore set identifier (*semid*) that was discussed above.

As declared, the process calling the *semget()* system call must supply three actual arguments to be passed to the formal *key*, *nsems*, and *semflg* arguments.

A new *semid* with an associated semaphore set and data structure is provided if either

- *key* is equal to IPC_PRIVATE, or
- *key* is passed a unique hexadecimal integer, and *semflg* ANDed with IPC_CREAT is TRUE

The value passed to the *semflg* argument must be an integer type octal value and must specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/alter attributes and execution modes determine the user/group/other attributes of the *semflg* argument. They are collectively referred to as "operation permissions."

Table 5–4 shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 5–4 Operation Permissions Codes

Operation Permissions	Octal Value
Read by User	00400
Alter by User	00200
Read by Group	00040
Alter by Group	00020
Read by Others	00004
Alter by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/alter by others is desired, the code value would be 00406 (00400 plus 00006). There are constants *#define*d in the *sem.h* header file that can be used for the owner. They are:

```
SEM_A 0200 /* alter permission by owner */
SEM_R 0400 /* read permission by owner */
```

Control commands are predefined constants (represented by all uppercase letters). **Table 5–5** contains the names of the constants that apply to the *semget()* system call along with their values. They are also referred to as flags and are defined in the *ipc.h* header file.

Table 5–5 Control Commands (Flags)

Control Command	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for *semflg* is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This specification is accomplished by using a bitwise OR (|) with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The *semflg* value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
semid = semget (key, nsems, (IPC_CREAT | 0400));
semid = semget (key, nsems, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the *semget(2)* manual page, success or failure of this system call depends upon the actual argument values for *key*, *nsems*, *semflg* or system tunable parameters. The system call attempts to return a new *semid* if one of the following conditions is true:

- *key* is equal to IPC_PRIVATE (0)
- *key* does not already have a *semid* associated with it, and (*semflg* & IPC_CREAT) is "true"

The *key* argument can be set to IPC_PRIVATE in the following ways:

```
semid = semget (IPC_PRIVATE, nsems, semflg);
```

or

```
semid = semget (0, nsems, semflg);
```

which causes the system call to be attempted because it satisfies the first condition specified.

Exceeding the SEMMNI, SEMMNS, or SEMMSL system−tunable parameters always causes a failure. The SEMMNI system tunable parameter determines the maximum number of unique semaphore sets (*semids*) in the UNIX operating system. The SEMMNS system tunable parameter determines the maximum number of semaphores in all semaphore sets system wide. The SEMMSL system tunable parameter determines the maximum number of semaphores in each semaphore set.

The second condition is satisfied if the value for *key* is not already associated with a *semid*, and the bitwise ANDing of *semflg* and IPC_CREAT is "true" (not zero). This means that the *key* is unique (not already in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag has been set (using *semflg* | IPC_CREAT). SEMMNI, SEMMNS, and SEMMSL apply here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a *semid* exists for the specified key provided. This is necessary to prevent the process from thinking that it has received a new (unique) *semid* when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a new *semid* is returned if the system call is successful. Any value for *semflg* returns a new *semid* if the *key* equals zero (IPC_PRIVATE) and no system tunable parameters are exceeded.

Refer to the *semget(2)* manual page for specific associated data structure initialization for successful completion.

Example Program

The example program in this section (**Example 5−4**) is a menu−driven program that exercises all possible combinations of the *semget()* system call.

From studying this program, you can observe the method of passing arguments and receiving return values. The user−written program requirements are pointed out.

This program begins (lines 4−8) by including the required header files as specified by the *semget(2)* manual page. Note that the *errno.h* header file is included as opposed to declaring *errno* as an external variable; either method works.

Variable names have been chosen to be as close as possible to those in the synopsis. Their declarations are self−explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purpose are as follows:

- *key*: used to pass the value for the desired key
- *opperm*: used to store the desired operation permissions
- *flags*: used to store the desired control commands (flags)
- *opperm_flags*: used to store the combination from the logical ORing of the *opperm* and *flags* variables; it is then used in the system call to pass the *semflg* argument
- *semid*: used for returning the semaphore set identification number for a successful system call or the error code (−1) for an unsuccessful one.

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and the control command combinations (flags), which are selected from a menu (lines 15−32). All possible combinations are allowed, even though they might not be viable. This allows you to observe the errors combinations are allowed, even though they might not be viable. This allows you to observe the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the *opperm_flags* variable (lines 36–52).

Then, the number of semaphores for the set is requested (lines 53–57), and its value is stored in *nsems*.

The system call is made next, and the result is stored in the *semid* variable (lines 60, 61).

Since the *semid* variable now contains a valid semaphore set identifier or the error code (–1), it is tested to see if an error occurred (line 63). If *semid* equals –1, a message indicates that an error resulted and the external *errno* variable is displayed (lines 65, 66). Remember that the external *errno* variable is only set when a system call fails; it should only be tested immediately following system calls.

If no error occurred, the returned semaphore set identifier is displayed (line 70).

The example program for the *semget()* system call is shown in **Example 5–4** .

Example 5–4 *semget()* System Call Example

```
1  /*This is a program to illustrate
2   *the semaphore get, semget(),
3   *system call capabilities.*/

4  #include <stdio.h>
5  #include <sys/types.h>
6  #include <sys/ipc.h>
7  #include <sys/sem.h>
8  #include <errno.h>

9  /*Start of main C language program*/
10  main()
11  {
12   key_t key; /*declare as long integer*/
13   int opperm, flags, nsems;
14   int semid, opperm_flags;

15   /*Enter the desired key*/
16   printf("\nEnter the desired key in hex = ");
17   scanf("%x", &key);

18   /*Enter the desired octal operation
19   permissions.*/
20   printf("\nEnter the operation\n");
21   printf("permissions in octal = ");
22   scanf("%o", &opperm);

23   /*Set the desired flags.*/
24   printf("\nEnter corresponding number to\n");
25   printf("set the desired flags:\n");
26   printf("No flags = 0\n");
27   printf("IPC_CREAT = 1\n");
28   printf("IPC_EXCL = 2\n");
29   printf("IPC_CREAT and IPC_EXCL = 3\n");
30   printf(" Flags = ");
31   /*Get the flags to be set.*/
32   scanf("%d", &flags);

33   /*Error checking (debugging)*/
34   printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
35    key, opperm, flags);
36   /*Incorporate the control fields (flags) with
37      the operation permissions.*/
```

```
  switch (flags)
  {
  case 0: /*No flags are to be set.*/
   opperm_flags = (opperm | 0);
   break;
  case 1: /*Set the IPC_CREAT flag.*/
   opperm_flags = (opperm | IPC_CREAT);
   break;
  case 2: /*Set the IPC_EXCL flag.*/
   opperm_flags = (opperm | IPC_EXCL);
   break;
  case 3: /*Set the IPC_CREAT and IPC_EXCL
    flags.*/
   opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
  }
  /*Get the number of semaphores for this set.*/
  printf("\nEnter the number of\n");
  printf("desired semaphores for\n");
  printf("this set (25 max) = ");
  scanf("%d", &nsems);
  /*Check the entry.*/
  printf("enNsems = %d\n", nsems);
  /*Call the semget system call.*/
  semid = semget(key, nsems, opperm_flags);
  /*Perform the following if the call is unsuccessful.*/
  if(semid == -1)
  {
   printf("The semget system call failed!\n");
   printf("The error number = %d\n", errno);
  }
  /*Return the semid upon successful completion.*/
  else
   printf("\nThe semid = %d\n", semid);
  exit(0);
 }
```

Controlling Semaphores with semctl()

This section details the *semctl()* system call and provides an example program that exercises all of its capabilities.

The synopsis found in the *semctl(*2*)* manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semctl (semid, semnum, cmd, arg)
int semid, semnum, cmd;
union semun

{
        int val;
        struct semid_ds *bu;
        ushort array[];
} arg;
```

The *semctl()* system call requires four arguments to be passed to it, and it returns an integer value.

The *semid* argument must be a valid, non−negative, integer value that has already been created by using the *semget()* system call.

The *semnum* argument is used to select a semaphore by its number. This relates to array (atomically performed) operations on the set. When a set of semaphores is created, the first semaphore is number 0, and the last semaphore has the number of one less than the total in the set.

The *cmd* argument can be replaced by one of the following control commands (flags):

- GETVAL: return the value of a single semaphore within a semaphore set
- SETVAL: set the value of a single semaphore within a semaphore set
- GETPID: return the process identifier (PID) of the process that performed the last operation on the semaphore within a semaphore set
- GETNCNT: return the number of processes waiting for the value of a particular semaphore to become greater than its current value
- GETZCNT: return the number of processes waiting for the value of a particular semaphore to be equal to zero
- GETALL: return the values for all semaphores in a semaphore set
- SETALL: set all semaphore values in a semaphore set
- IPC_STAT: return the status information contained in the associated data structure for the specified *semid*, and place it in the data structure pointed to by the **buf* pointer in the user memory area; *arg.buf* is the union member that contains the value of *buf*
- IPC_SET: for the specified semaphore set (*semid*), set the effective user/group identification and operation permissions
- IPC_RMID−remove the specified (*semid*) semaphore set along with its associated data structure.

A process must have an effective user identification of OWNER/CREATOR or super user to perform an IPC_SET or IPC_RMID control command. Read/alter permission is required as applicable for the other control commands.

Use the *arg* argument to pass the appropriate union member to the system call for the control command to be performed:

- *arg.val*
- *arg.buf*
- *arg.array*

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read **"Getting Semaphores with semget()"** ; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (Figure 8−10) is a menu−driven program that exercises all possible combinations of using the *semctl()* system call.

From studying this program, you can observe the method of passing arguments and receiving return values. The user−written program requirements are pointed out.

This program begins (lines 5−9) by including the required header files as specified by the *semctl(*2*)* manual page. Note that in this program *errno* is declared as an external variable, and therefore the *errno.h* header file does not have to be included.

Variable, structure, and union names have been chosen to be as close as possible to those in the synopsis. Their declarations are self−explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. Those declared for this program and their purpose are as follows:

- *semid_ds*: used to receive the specified semaphore set identifier's data structure when an IPC_STAT control command is performed
- *c*: used to receive the input values from the *scanf()* function (line 117) when performing a SETALL control command
- *i*: used as a counter to increment through the union *arg.array* when displaying the semaphore values for a GETALL (lines 97−99) control command, and when initializing the *arg.array* when performing a SETALL (lines 115−119) control command
- *length*: used as a variable to test for the number of semaphores in a set against the *i* counter variable (lines 97, 115)
- *uid*: used to store the IPC_SET value for the effective user identification

- *gid*: used to store the IPC_SET value for the effective group identification
- *mode*: used to store the IPC_SET value for the operation permissions
- *rtrn*: used to store the return integer from the system call which depends upon the control command or a −1 when unsuccessful
- *semid*: used to store and pass the semaphore set ID to the system call
- *semnum*: used to store and pass the semaphore number to the system call
- *cmd*: used to store the code for the desired control command so that subsequent processing can be performed on it
- *choice*: used to determine which member (*uid*, *gid*, *mode*) for the IPC_SET control command that is to be changed
- *arg.val*: used to pass the system call a value to set (SETVAL) or to store (GETVAL) a value returned from the system call for a single semaphore (union member)
- *arg.buf*: a pointer passed to the system call. It locates the data structure in the user memory area where the IPC_STAT control command is to place its return values, or where the IPC_SET command gets the values to set (union member)
- *arg.array*: used to store the set of semaphore values when getting (GETALL) or initializing (SETALL) (union member).

Note that the *semid_ds* data structure in this program (line 14) uses the data structure located in the *sem.h* header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The *arg* union (lines 18−22) serves three purposes in one. The compiler allocates enough storage to hold its largest member. The program can then use the union as any member by referencing union members as if they were regular structure members. Note that the array is declared to have 25 elements (0 through 24).This number corresponds to the maximum number of semaphores allowed per set (SEMMSL), a system−tunable parameter.

The next important program aspect to observe is that although the **buf* pointer member (*arg.buf*) of the union is declared to be a pointer to a data structure of the *semid_ds* type, it must also be initialized to contain the address of the user memory area data structure (line 24). Because of the way this program is written, the pointer does not need to be reinitialized later. If it was used to increment through the array, it would need to be reinitialized just before calling the system call.

Now all of the required declarations have been presented for this program. The next paragraphs explain how it works.

First, the program prompts for a valid semaphore set identifier, which is stored in the *semid* variable (lines 25−27). This is required for all *semctl()* system calls.

Then, the code for the desired control command must be entered (lines 28−42), and the code is stored in the *cmd* variable. The code is tested to determine the control command for subsequent processing.

If the GETVAL control command is selected (code 1), a message prompting for a semaphore number is displayed (lines 49, 50). When it is entered, it is stored in the *semnum* variable (line 51). Then, the system call is performed, and the semaphore value is displayed (lines 52−55). If the system call is successful, a message indicates this along with the semaphore set identifier used (lines 195, 196); if the system call is unsuccessful, an error message is displayed along with the value of the external *errno* variable (lines 191−193).

If the SETVAL control command is selected (code 2), a message prompting for a semaphore number is displayed (lines 56, 57). When it is entered, it is stored in the *semnum* variable (line 58). Next, a message prompts for the value to which the semaphore is to be set, and it is stored as the *arg.val* member of the union (lines 59, 60). Then, the system call is performed (lines 61, 63). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETPID control command is selected (code 3), the system call is made immediately since all required arguments are known (lines 64−67), and the PID of the process performing the last operation is displayed. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETNCNT control command is selected (code 4), a message prompting for a semaphore number is displayed (lines 68−72). When entered, it is stored in the *semnum* variable (line 73). Then, the system call is performed, and the number of processes waiting for the semaphore to become greater than its current value is displayed (lines 74−77). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETZCNT control command is selected (code 5), a message prompting for a semaphore number is displayed (lines 78−81). When it is entered, it is stored in the *semnum* variable (line 82). Then the system call is performed, and the number of processes waiting for the semaphore value to become equal to zero is displayed (lines 83, 86). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the GETALL control command is selected (code 6), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 88−93). The length variable is set to the number of semaphores in the set (line 91). Next, the system call is made and, upon success, the *arg.array* union member contains the values of the semaphore set (line 96). Now, a loop is entered that displays each element of the *arg.array* from zero to one less than the value of length (lines 97−103). The semaphores in the set are displayed on a single line, separated by a space. Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the SETALL control command is selected (code 7), the program first performs an IPC_STAT control command to determine the number of semaphores in the set (lines 106−108). The length variable is set to the number of semaphores in the set (line 109). Next, the program prompts for the values to be set and enters a loop that takes values from the keyboard and initializes the *arg.array* union member to contain the desired values of the semaphore set (lines 113−119). The loop puts the first entry into the array position for semaphore number zero and ends when the semaphore number that is filled in the array equals one less than the value of length. The system call is then made (lines 120−122). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_STAT control command is selected (code 8), the system call is performed (line 127), and the status information returned is printed out (lines 128−139); only the members that can be set are printed out in this program. Note that if the system call is unsuccessful, the status information of the last successful one is printed out. In addition, an error message is displayed, and the *errno* variable is printed out (lines 191, 192).

If the IPC_SET control command is selected (code 9), the program gets the current status information for the semaphore set identifier specified (lines 143−146). This is necessary because this example program provides for changing only one member at a time, and the *semctl()* system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 147−153). This code is stored in the *choice* variable (line 154). Now, depending upon the member picked, the program prompts for the new value (lines 155−178). The value is placed in the appropriate member in the user memory area data structure, and the system call is made (line 181). Depending upon success or failure, the program returns the same messages as for GETVAL above.

If the IPC_RMID control command (code 10) is selected, the system call is performed (lines 183−185). The *semid* is removed from the UNIX operating system along with its associated data structure and semaphore set. Depending upon success or failure, the program returns the same messages as for the other control commands.

The example program for the *semctl()* system call is shown in **Example 5−5**.

Example 5−5 *semctl()* System Call Example

```
1  /*This is a program to illustrate
2   *the semaphore control, semctl(),
3   *system call capabilities.
4   */
5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/sem.h>
10  /*Start of main C language program*/
```

```
main()
{
 extern int errno;
 struct semid_ds semid_ds;
 int c, i, length;
 int uid, gid, mode;
 int retrn, semid, semnum, cmd, choice;
 union semun {
  int val;
  struct semid_ds *buf;
  ushort array[25];
 } arg;
 /*Initialize the data structure pointer.*/
 arg.buf = &semid_ds;
 /*Enter the semaphore ID.*/
 printf("Enter the semid = ");
 scanf("%d", &semid);
 /*Choose the desired command.*/
 printf("\nEnter the number for\n");
 printf("the desired cmd:\n");
 printf("GETVAL = 1\n");
 printf("SETVAL = 2\n");
 printf("GETPID = 3\n");
 printf("GETNCNT = 4\n");
 printf("GETZCNT = 5\n");
 printf("GETALL = 6\n");
 printf("SETALL = 7\n");
 printf("IPC_STAT = 8\n");
 printf("IPC_SET = 9\n");
 printf("IPC_RMID = 10\n");
 printf("Entry = ");
 scanf("%d", &cmd);
 /*Check entries.*/
 printf ("\nsemid =%d, cmd = %d\n\n",
  semid, cmd);
 /*Set the command and do the call.*/
 switch (cmd)
 {
 case 1: /*Get a specified value.*/
  printf("\nEnter the semnum = ");
  scanf("%d", &semnum);
  /*Do the system call.*/
  retrn = semctl(semid, semnum, GETVAL, 0);
  printf("\nThe semval = %d\n", retrn);
  break;
 case 2: /*Set a specified value.*/
  printf("\nEnter the semnum = ");
  scanf("%d", &semnum);
  printf("\nEnter the value = ");
  scanf("%d", &arg.val);
  /*Do the system call.*/
  retrn = semctl(semid, semnum, SETVAL, arg.val);
  break;
 case 3: /*Get the process ID.*/
  retrn = semctl(semid, 0, GETPID, 0);
  printf("\nThe sempid = %d\n", retrn);
  break;
 case 4: /*Get the number of processes
  waiting for the semaphore to
```

```
 become greater than its current
  value.*/
 printf("\nEnter the semnum = ");
 scanf("%d", &semnum);
 /*Do the system call.*/
 retrn = semctl(semid, semnum, GETNCNT, 0);
 printf("\nThe semncnt = %d", retrn);
 break;
case 5: /*Get the number of processes
 waiting for the semaphore
  value to become zero.*/
 printf("\nEnter the semnum = ");
 scanf("%d", &semnum);
 /*Do the system call.*/
 retrn = semctl(semid, semnum, GETZCNT, 0);
 printf("\nThe semzcnt = %d", retrn);
 break;
case 6: /*Get all of the semaphores.*/
 /*Get the number of semaphores in
 the semaphore set.*/
 retrn = semctl(semid, 0, IPC_STAT, arg.buf);
 length = arg.buf->sem_nsems;
 if(retrn == -1)
  goto ERROR;
 /*Get and print all semaphores in the
 specified set.*/
 retrn = semctl(semid, 0, GETALL, arg.array);
 for (i = 0; i < length; i++)
 {
  printf("%d", arg.array[i]);
   /*Seperate each
     semaphore.*/
   printf("%c", ’ ’);
  }
  break;
 case 7: /*Set all semaphores in the set.*/
  /*Get the number of semaphores in
  the set.*/
  retrn = semctl(semid, 0, IPC_STAT, arg.buf);
  length = arg.buf->sem_nsems;
  printf("Length = %den", length);
  if(retrn == -1)
   goto ERROR;
  /*Set the semaphore set values.*/
  printf("\nEnter each value:\n");
  for(i = 0; i < length ; i++)
  {
   scanf("%d", &c);
   arg.array[i] = c;
  }
  /*Do the system call.*/
  retrn = semctl(semid, 0, SETALL, arg.array);
  break;
 case 8: /*Get the status for the semaphore set.*/
  /*Get and print the current status values.*/
  retrn = semctl(semid, 0, IPC_STAT, arg.buf);
  printf ("\nThe USER ID = %d\n",
   arg.buf->sem_perm.uid);
```

```
 printf ("The GROUP ID = %d\n",
  arg.buf->sem_perm.gid);
 printf ("The operation permissions = 0%o\n",
  arg.buf->sem_perm.mode);
 printf ("The number of semaphores in set = %d\n",
  arg.buf->sem_nsems);
 printf ("The last semop time = %d\n",
  arg.buf->sem_otime);
 printf ("The last change time = %d\n",
  arg.buf->sem_ctime);
 break;
case 9: /*Select and change the desired
         member of the data structure.*/
 /*Get the current status values.*/
 retrn = semctl(semid, 0, IPC_STAT, arg.buf);
 if(retrn == -1)
  goto ERROR;
 /*Select the member to change.*/
 printf("\nEnter the number for the\n");
 printf("member to be changed:\n");
 printf("sem_perm.uid = 1\n");
 printf("sem_perm.gid = 2\n");
 printf("sem_perm.mode = 3\n");
 printf("Entry = ");
 scanf("%d", &choice);
 switch(choice){
 case 1: /*Change the user ID.*/
  printf("\nEnter USER ID = ");
  scanf ("%d", &uid);
  arg.buf->sem_perm.uid = uid;
  printf("enUSER ID = %d\n",
   arg.buf->sem_perm.uid);
  break;
 case 2: /*Change the group ID.*/
  printf("\nEnter GROUP ID = ");
  scanf("%d", &gid);
  arg.buf->sem_perm.gid = gid;
  printf("\nGROUP ID = %d\n",
   arg.buf->sem_perm.gid);
  break;
 case 3: /*Change the mode portion of
  the operation
    permissions.*/
  printf("\nEnter MODE = ");
  scanf("%o", &mode);
  arg.buf->sem_perm.mode = mode;
  printf("\nMODE = 0%o\n",
   arg.buf->sem_perm.mode);
  break;
 }
 /*Do the change.*/
 retrn = semctl(semid, 0, IPC_SET, arg.buf);
 break;
case 10: /*Remove the semid along with its
  data structure.*/
 retrn = semctl(semid, 0, IPC_RMID, 0);
}
/*Perform the following if the call is unsuccessful.*/
if(retrn == -1)
```

```
 {
ERROR:
  printf ("\n\nThe semctl system call failed!\n");
  printf ("The error number = %d\n", errno);
  exit(0);
 }
 printf ("\n\nThe semctl system call was successful\n");
 printf ("for semid = %d\n", semid);
 exit (0);
}
```

Operations on Semaphores: semop()

This section details the *semop()* system call and provides an example program that exercises its capabilities.

The synopsis found in the *semop(2)* manual page is as follows:

```
#include  <sys/types.h>
#include  <sys/ipc.h>
#include  <sys/sem.h>

int semop (semid, sops, nsops)
int semid;
struct sembuf *sops;
size_t nsops;
```

semop() returns zero on success, or −1 on failure.

The *semid* argument must be a valid, non−negative, integer value. In other words, it must have already been created by using the *semget()* system call.

The *sops* argument is a pointer to an array of structures in the user memory area that contains the following for each semaphore to be changed:

- the semaphore number
- the operation to be performed
- the control command (flags)

The **sops* declaration means that a pointer can be initialized to the address of the array, or the array name can be used since it is the address of the first element of the array. *Sembuf* is the *tag* name of the data structure used as the template for the structure members in the array; it is located in the */usr/include/sys/sem.h* header file.

The *nsops* argument specifies the length of the array (the number of structures in the array). The maximum size of this array is determined by the SEMOPM system−tunable parameter. Therefore, a maximum of SEMOPM operations can be performed for each *semop()* system call.

The semaphore number determines the particular semaphore within the set on which the operation is to be performed.

The operation to be performed is determined by the following:

- a positive integer value means to increment the semaphore value by its value
- a negative integer value means to decrement the semaphore value by its value
- a value of zero means to test if the semaphore is equal to zero

The following operation commands (flags) can be used:

- IPC_NOWAIT: this operation command can be set for any operations in the array. The system call returns unsuccessfully without changing any semaphore values at all if any operation for which IPC_NOWAIT is set cannot be performed successfully. The system call is unsuccessful when trying to decrement a semaphore more than its current value, or when testing for a semaphore to be equal to zero when it is not.
- SEM_UNDO: this operation command allows any operations in the array to be undone when any operation in the array is unsuccessful and does not have the IPC_NOWAIT flag set. That is, the

blocked operation waits until it can perform its operation; and when it and all succeeding operations are successful, all operations with the SEM_UNDO flag set are undone. Remember, no operations are performed on any semaphores in a set until all operations are successful. Undoing is accomplished by using an array of adjust values for the operations that are to be undone when the blocked operation and all subsequent operations are successful.

Example Program

The example program in this section (**Example 5−6**) is a menu−driven program that exercises all possible combinations of the *semop()* system call.

From studying this program, you can observe the method of passing arguments and receiving return values. The user−written program requirements are pointed out.

This program begins (lines 5−9) by including the required header files as specified by the *semop(2)* manual page. Note that in this program *errno* is declared as an external variable, and therefore, the *errno.h* header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis. Their declarations are self−explanatory. These names make the program more readable, and this is perfectly legal since the declarations are local to the program. The variables declared for this program and their purpose are as follows:

- *sembuf[10]*: used as an array buffer (line 14) to contain a maximum of ten *sembuf* type structures; ten equals SEMOPM, the maximum number of operations on a semaphore set for each *semop()* system call
- _*sops_: used as a pointer (line 14) to *sembuf[10]* for the system call and for accessing the structure members within the array
- *rtrn*: used to store the return values from the system call
- *flags*: used to store the code of the IPC_NOWAIT or SEM_UNDO flags for the *semop()* system call (line 60)
- *i*: used as a counter (line 32) for initializing the structure members in the array, and used to print out each structure in the array (line 79)
- *nsops*: used to specify the number of semaphore operations for the system call; must be less than or equal to SEMOPM
- *semid*: used to store the desired semaphore set identifier for the system call

First, the program prompts for a semaphore set identifier that the system call is to perform operations on (lines 19−22). Semid is stored in the *semid* variable (line 23).

A message is displayed requesting the number of operations to be performed on this set (lines 25−27). The number of operations is stored in the *nsops* variable (line 28).

Next, a loop is entered to initialize the array of structures (lines 30−77). The semaphore number, operation, and operation command (flags) are entered for each structure in the array. The number of structures equals the number of semaphore operations (*nsops*) to be performed for the system call, so *nsops* is tested against the *i* counter for loop control. Note that *sops* is used as a pointer to each element (structure) in the array, and *sops* is incremented just like *i*. *sops* is then used to point to each member in the structure for setting them.

After the array is initialized, all of its elements are printed out for feedback (lines 78−85).

The *sops* pointer is set to the address of the array (lines 86, 87). *Sembuf* could be used directly, if desired, instead of *sops* in the system call.

The system call is made (line 89), and depending upon success or failure, a corresponding message is displayed. The results of the operation(s) can be viewed by using *semctl()* with the GETALL control command.

The example program for the *semop()* system call is shown in **Example 5−6** .

Example 5−6 *semop()* System Call Example

```
1  /*This is a program to illustrate
2   *the semaphore operations, semop(),
3   *system call capabilities.
4   */
```

```
/*Include necessary header files.*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
 *Start of main C language program*/
 main()
 {
  extern int errno;
  struct sembuf sembuf[10], *sops;
  char string[];
  int retrn, flags, sem_num, i, semid;
  unsigned nsops;
  sops = sembuf; /*Pointer to array sembuf.*/
  /*Enter the semaphore ID.*/
  printf("\nEnter the semid of\n");
  printf("the semaphore set to\n");
  printf("be operated on = ");
  scanf("%d", &semid);
  printf("ensemid = %d", semid);
  /*Enter the number of operations.*/
  printf("\nEnter the number of semaphore\n");
  printf("operations for this set = ");
  scanf("%d", &nsops);
  printf("ennosops = %d", nsops);
  /*Initialize the array for the
    number of operations to be performed.*/
  for(i = 0; i < nsops; i++, sops++)
  {
   /*This determines the semaphore in
   the semaphore set.*/
   printf("\nEnter the semaphore\n");
   printf("number (sem_num) = ");
   scanf("%d", &sem_num);
   sops->sem_num = sem_num;
   printf("\nThe sem_num = %d", sops->sem_num);

   /*Enter a (-)number to decrement,
     an unsigned number (no +) to increment,
     or zero to test for zero. These values
     are entered into a string and converted
     to integer values.*/
   printf("\nEnter the operation for\n");
   printf("the semaphore (sem_op) = ");
   scanf("%s", string);
   sops->sem_op = atoi(string);
   printf("ensem_op = %d\n", sops->sem_op);
   /*Specify the desired flags.*/
   printf("\nEnter the corresponding\n");
   printf("number for the desired\n");
   printf("flags:\n");
   printf("No flags = 0\n");
   printf("IPC_NOWAIT = 1\n");
   printf("SEM_UNDO = 2\n");
   printf("IPC_NOWAIT and SEM_UNDO = 3\n");
   printf(" Flags = ");
   scanf("%d", &flags);
   switch(flags)
   {
```

```
  case 0:
   sops->sem_flg = 0;
   break;
  case 1:
   sops->sem_flg = IPC_NOWAIT;
   break;
  case 2:
   sops->sem_flg = SEM_UNDO;
   break;
  case 3:
   sops->sem_flg = IPC_NOWAIT | SEM_UNDO;
   break;
  }
  printf("\nFlags = 0%o\n", sops->sem_flg);
 }
 /*Print out each structure in the array.*/
 for(i = 0; i < nsops; i++)
 {
  printf("\nsem_num = %d\n", sembuf[i].sem_num);
  printf("sem_op = %d\n", sembuf[i].sem_op);
  printf("sem_flg = %o\n", sembuf[i].sem_flg);
  printf("%c", ’ ’);
 }

 sops = sembuf; /*Reset the pointer to
       sembuf[0].*/
 /*Do the semop system call.*/
 retrn = semop(semid, sops, nsops);
 if(retrn == -1) {
  printf("\nSemop failed. ");
  printf("Error = %d\n", errno);
 }
 else {
  printf ("\nSemop was successful\n");
  printf("for semid = %d\n", semid);
  printf("Value returned = %d\n", retrn);
 }
}
```

System V Shared Memory

Shared memory IPC allows two or more executing processes to share memory and consequently the data contained there. This is done by allowing processes to set up access to a common virtual memory address space. This sharing occurs on a segment basis, which depends on memory management hardware.

This sharing of memory provides the fastest means of exchanging data between processes.

A process initially creates a shared memory segment facility using the *shmget()* system call. Upon creation, this process sets the overall operation permissions for the shared memory segment facility, sets its size in bytes, and can specify that the shared memory segment is for reference only (read–only) upon attachment. If the memory segment is not specified to be for reference only, all other processes with appropriate operation permissions can read from or write to the memory segment.

There are two operations that can be performed on a shared memory segment:

- *shmat()* – shared memory attach
- *shmdt()* – shared memory detach

Shared memory attach allows processes to associate themselves with the shared memory segment if they have permission. They can then read or write as allowed.

Shared memory detach allows processes to disassociate themselves from a shared memory segment. Therefore, they lose the ability to read from or write to the shared memory segment.

The original owner/creator of a shared memory segment can relinquish ownership to another process using the *shmctl()* system call. However, the creating process remains the creator until the facility is removed or the system is reinitialized. Other processes with permission can perform other functions on the shared memory segment using the *shmctl()* system call.

System calls, documented in the IRIX manual pages, make these shared memory capabilities available to processes. The calling process passes arguments to a system call, and the system call attempts to perform its function. If the system call is successful, it performs its function and returns the appropriate information. Otherwise, a known error code (–1) is returned to the process, and the external variable *errno* is set accordingly.

Using Shared Memory

The sharing of memory between processes occurs on a virtual segment basis. There is one and only one instance of an individual shared memory segment existing in the UNIX operating system at any point in time.

Before memory sharing can be realized, a uniquely identified shared memory segment and data structure must be created. The unique identifier created is called the shared memory identifier (*shmid*); it is used to identify or reference the associated data structure. The data structure includes the following for each shared memory segment:

- operation permissions
- segment size
- segment descriptor
- process identification performing last operation
- process identification of creator
- current number of processes attached
- in memory number of processes attached
- last attach time
- last detach time
- last change time

The definition for the shared memory segment data structure is located in the */usr/include/sys/shm.h* header file. It is as follows:

```
/*
   * There is a shared mem id data structure for
   * each segment in the system.
   */

struct shmid_ds {
struct ipc_perm shm_perm; /* operation permission struct */
int shm_segsz; /* segment size */
struct region *shm_reg; /* ptr to region structure */
char pad[4]; /* for swap compatibility */
ushort shm_lpid; /* pid of last shmop */
ushort shm_cpid; /* pid of creator */
ushort shm_nattch; /* used only for shminfo */
ushort shm_cnattch; /* used only for shminfo */
time_t shm_atime; /* last shmat time */
time_t shm_dtime; /* last shmdt time */
time_t shm_ctime; /* last change time */ };
```

Note that the *shm_perm* member of this structure uses *ipc_perm* as a template. The *ipc_perm* data structure is the same for all IPC facilities, and is located in the */usr/include/sys/ipc.h* header file.

The *shmget()* system call is used to perform two tasks when only the IPC_CREAT flag is set in the *shmflg* argument that it receives:

- to get a new *shmid* and create an associated shared memory segment data structure for it

- to return an existing *shmid* that already has an associated shared memory segment data structure

The task performed is determined by the value of the *key* argument passed to the *shmget()* system call. For the first task, if the *key* is not already in use for an existing *shmid*, a new *shmid* is returned with an associated shared memory segment data structure created for it, provided no system tunable parameters would be exceeded.

A provision exists for specifying a *key* of value zero, known as the private *key* (IPC_PRIVATE = 0); when specified, a new *shmid* is always returned with an associated shared memory segment data structure created for it unless a system tunable parameter would be exceeded. When the *ipcs* command is performed, the KEY field for the *shmid* is all zeros.

For the second task, if a *shmid* exists for the *key* specified, the value of the existing *shmid* is returned. If you do not want to have an existing *shmid* returned, a control command (IPC_EXCL) can be set in the *shmflg* argument passed to the system call. The details of using this system call are discussed in **"Getting Shared Memory Segments with shmget()."**

When performing the first task, the process that calls *shmget()* becomes the owner/creator, and the associated data structure is initialized accordingly. Remember, ownership can be changed, but the creating process always remains the creator; see **"Controlling Shared Memory: shmctl()."** The creator of the shared memory segment also determines the initial operation permissions for it.

Once a uniquely–identified shared memory segment data structure is created, shared memory segment operations and control can be used.

Shared memory segment operations consist of attaching and detaching shared memory segments. System calls are provided for each of these operations; they are *shmat()* and *shmdt()*. Refer to **"Operations for Shared Memory: shmat() and shmdt()"** for details of these system calls.

Shared memory segment control is done by using the *shmctl()* system call. It permits you to control the shared memory facility in the following ways:

- to determine the associated data structure status for a shared memory segment (*shmid*)
- to change operation permissions for a shared memory segment
- to remove a particular *shmid* from the UNIX operating system along with its associated shared memory segment data structure
- to lock a shared memory segment in memory
- to unlock a shared memory segment

Refer to **"Controlling Shared Memory: shmctl()"** for details of the *shmctl()* system call.

Getting Shared Memory Segments with shmget()

This section gives a detailed description of using the *shmget()* system call along with an example program illustrating its use.

The synopsis found in the *shmget(2)* manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget (key, size, shmflg)
key_t key;
int size, shmflg;
```

All of these include files are located in the */usr/include/sys* directory of the UNIX operating system.

The integer returned from this function upon successful completion is the shared memory identifier (*shmid*) that was discussed earlier.

As declared, the process calling the *shmget()* system call must supply three arguments to be passed to the formal *key*, *size*, and *shmflg* arguments.

A new *shmid* with an associated shared memory data structure is provided if either of the following is true:

- *key* is equal to IPC_PRIVATE
- *key* is passed a unique hexadecimal integer, and *shmflg* ANDed with IPC_CREAT is TRUE

The value passed to the *shmflg* argument must be an integer−type octal value and must specify the following:

- access permissions
- execution modes
- control fields (commands)

Access permissions determine the read/write attributes, and execution modes determine the user/group/other attributes of the *shmflg* argument. They are collectively referred to as "operation permissions." **Table 5−6** shows the numeric values (expressed in octal notation) for the valid operation permissions codes.

Table 5−6 Operation Permissions Codes

Operation Permissions	Octal Values
Read by User	00400
Write by User	00200
Read by Group	00040
Write by Group	00020
Read by Others	00004
Write by Others	00002

A specific octal value is derived by adding the octal values for the operation permissions desired. That is, if read by user and read/write by others are desired, the code value would be 00406 (00400 plus 00006).

Constants located in the *shm.h* header file can be used for the owner. They include:

```
SHM_R 0400
SHM_W 0200
```

Control commands are predefined constants (represented by all uppercase letters). **Table 5−7** contains the names of the constants that apply to the *shmget()* system call along with their values. They are also referred to as flags and are defined in the *ipc.h* header file.

Table 5−7 Control Commands (Flags)

Control	Value
IPC_CREAT	0001000
IPC_EXCL	0002000

The value for *shmflg* is, therefore, a combination of operation permissions and control commands. After determining the value for the operation permissions as previously described, the desired flag(s) can be specified. This is accomplished by using bitwise OR (|) with the operation permissions; the bit positions and values for the control commands in relation to those of the operation permissions make this possible.

The *shmflg* value can be easily set by using the names of the flags in conjunction with the octal operation permissions value:

```
shmid = shmget (key, size, (IPC_CREAT | 0400));
shmid = shmget (key, size, (IPC_CREAT | IPC_EXCL | 0400));
```

As specified by the *shmget(2)* manual page, success or failure of this system call depends upon the argument values for *key*, *size*, and *shmflg* or system tunable parameters. The system call attempts to return a new *shmid* if one of the following conditions is true:

- *key* is equal to IPC_PRIVATE (0)
- *key* does not already have a *shmid* associated with it, and (*shmflg* & IPC_CREAT) is "true" (not zero)

The *key* argument can be set to IPC_PRIVATE in the following ways:

```
shmid = shmget (IPC_PRIVATE, size, shmflg);
```

or

```
shmid = shmget (0, size, shmflg);
```

which causes the system call to be attempted because it satisfies the first condition specified. Exceeding the SHMMNI system−tunable parameter always causes a failure. The SHMMNI system−tunable parameter determines the maximum number of unique shared memory segments (*shmids*) in the UNIX operating system.

The second condition is satisfied if the value for *key* is not already associated with a *shmid* and the bitwise ANDing of *shmflg* and IPC_CREAT is "true" (not zero). This means that the *key* is unique (not in use) within the UNIX operating system for this facility type and that the IPC_CREAT flag has been set (by using *shmflg* | IPC_CREAT). SHMMNI applies here also, just as for condition one.

IPC_EXCL is another control command used in conjunction with IPC_CREAT to exclusively have the system call fail if, and only if, a *shmid* exists for the specified *key* provided. This is necessary to prevent the process from thinking that it has received a new (unique) *shmid* when it has not. In other words, when both IPC_CREAT and IPC_EXCL are specified, a unique *shmid* is returned if the system call is successful. Any value for *shmflg* returns a new *shmid* if the *key* equals zero (IPC_PRIVATE).

The system call fails if the value for the size argument is less than SHMMIN or greater than SHMMAX. These tunable parameters specify the minimum and maximum shared memory segment sizes.

Refer to the *shmget(2)* manual page for specific associated data structure initialization for successful completion. The specific failure conditions with error names are contained there also.

Example Program

The example program in this section (**Example 5−7**) is a menu−driven program that exercises all possible combinations of the *shmget()* system call.

From studying this program, you can observe the method of passing arguments and receiving return values. The program requirements are pointed out.

This program begins (lines 4−7) by including the required header files as specified by the *shmget(2)* manual page. Note that the *errno.h* header file is included as opposed to declaring *errno* as an external variable; either method works.

Variable names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self−explanatory. These names make the program more readable, and this is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- *key*: used to pass the value for the desired key
- *opperm*: used to store the desired operation permissions
- *flags*: used to store the desired control commands (flags)
- *opperm_flags*: used to store the combination from the logical ORing of the *opperm* and *flags* variables; it is then used in the system call to pass the *shmflg* argument
- *shmid*: used for returning the message queue identification number for a successful system call or the error code (−1) for an unsuccessful one
- *size*: used to specify the shared memory segment size.

The program begins by prompting for a hexadecimal *key*, an octal operation permissions code, and finally for the control command combinations (flags), which are selected from a menu (lines 14−31). All possible combinations are allowed even though they might not be viable. Thus you can observe the errors for illegal combinations.

Next, the menu selection for the flags is combined with the operation permissions, and the result is stored in the *opperm_flags* variable (lines 35−50).

A display then prompts for the size of the shared memory segment, which is stored in the *size* variable (lines 51−54).

The system call is made next, and the result is stored in the *shmid* variable (line 56).

Since the *shmid* variable now contains a valid message queue identifier or the error code (−1), it is tested to see if an error occurred (line 58). If *shmid* equals −1, a message indicates that an error resulted and the external *errno* variable is displayed (lines 60, 61).

If no error occurred, the returned shared memory segment identifier is displayed (line 65).

The example program for the *shmget()* system call is shown in **Example 5−7** .

Example 5−7 *shmget()* System Call Example

```
/*This is a program to illustrate
 *the shared memory get, shmget(),
 *system call capabilities.*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <errno.h>
/*Start of main C language program*/
main()
 {
  key_t key; /*declare as long integer*/
  int opperm, flags;
  int shmid, size, opperm_flags;
  /*Enter the desired key*/
  printf("Enter the desired key in hex = ");
  scanf("%x", &key);
  /*Enter the desired octal operation
    permissions.*/
  printf("\nEnter the operation\n");
  printf("permissions in octal = ");
  scanf("%o", &opperm);
  /*Set the desired flags.*/
  printf("\nEnter corresponding number to\n");
  printf("set the desired flags:\n");
  printf("No flags = 0\n");
  printf("IPC_CREAT = 1\n");
  printf("IPC_EXCL = 2\n");
  printf("IPC_CREAT and IPC_EXCL = 3\n");
  printf(" Flags = ");
  /*Get the flag(s) to be set.*/
  scanf("%d", &flags);
  /*Check the values.*/
  printf ("\nkey =0x%x, opperm = 0%o, flags = 0%o\n",
   key, opperm, flags);
  /*Incorporate the control fields (flags) with
    the operation permissions*/
  switch (flags)
  {
  case 0: /*No flags are to be set.*/
   opperm_flags = (opperm | 0);
   break;
  case 1: /*Set the IPC_CREAT flag.*/
   opperm_flags = (opperm | IPC_CREAT);
   break;
  case 2: /*Set the IPC_EXCL flag.*/
   opperm_flags = (opperm | IPC_EXCL);
   break;
  case 3: /*Set the IPC_CREAT and IPC_EXCL flags.*/
   opperm_flags = (opperm | IPC_CREAT | IPC_EXCL);
  }
  /*Get the size of the segment in bytes.*/
  printf ("\nEnter the segment");
  printf ("\nsize in bytes = ");
  scanf ("%d", &size);
  /*Call the shmget system call.*/
  shmid = shmget (key, size, opperm_flags);
  /*Perform the following if the call is unsuccessful.*/
  if(shmid == -1)
```

```
59   {
60    printf ("\nThe shmget system call failed!\n");
61    printf ("The error number = %d\n", errno);
62   }
63   /*Return the shmid upon successful completion.*/
64   else
65    printf ("\nThe shmid = %d\n", shmid);
66   exit(0);
67  }
```

Controlling Shared Memory: shmctl()

This section details using the *shmctl()* system call and provides an example program that exercises all of its capabilities.

The synopsis found in the *shmctl(2)* manual page is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int shmctl (shmid, cmd, buf)
int shmid, cmd;
struct shmid_ds *buf;
```

The *shmctl()* system call requires three arguments to be passed to it, and it returns an integer value.

Upon successful completion, a zero value is returned; when unsuccessful, *shmctl()* returns a −1.

The *shmid* variable must be a valid, non−negative, integer value. In other words, it must have already been created by using the *shmget()* system call.

The *cmd* argument can be replaced by one of following control commands (flags):

- IPC_STAT: return the status information contained in the associated data structure for the specified *shmid* and place it in the data structure pointed to by the **buf* pointer in the user memory area
- IPC_SET: for the specified *shmid*, set the effective user and group identification, and operation permissions
- IPC_RMID: remove the specified *shmid* along with its associated shared memory segment data structure
- SHM_LOCK: lock the specified shared memory segment in memory; must be super user
- SHM_UNLOCK: unlock the shared memory segment from memory; must be super user.

A process must have an effective user identification of OWNER/CREATOR or super user to perform an IPC_SET or IPC_RMID control command. Only the super user can perform a SHM_LOCK or SHM_UNLOCK control command. A process must have read permission to perform the IPC_STAT control command.

The details of this system call are discussed in the example program for it. If you have problems understanding the logic manipulations in this program, read **"Getting Shared Memory Segments with shmget()"** ; it goes into more detail than would be practical to do for every system call.

Example Program

The example program in this section (**Example 5−8**) is a menu−driven program that exercises all possible combinations of the *shmctl()* system call.

From studying this program, you can observe the method of passing arguments and receiving return values. The user−written program requirements are pointed out.

This program begins (lines 5−9) by including the required header files as specified by the *shmctl(2)* manual page. Note in this program that *errno* is declared as an external variable, and therefore, the *errno.h* header file does not have to be included.

Variable and structure names have been chosen to be as close as possible to those in the synopsis for the system call. Their declarations are self−explanatory. These names make the program more readable, and it is perfectly legal since they are local to the program. The variables declared for this program and their purposes are as follows:

- *uid*: used to store the IPC_SET value for the effective user identification
- *gid*: used to store the IPC_SET value for the effective group identification
- *mode*: used to store the IPC_SET value for the operation permissions
- *rtrn*: used to store the return integer value from the system call
- *shmid*: used to store and pass the shared memory segment identifier to the system call
- *command*: used to store the code for the desired control command so that subsequent processing can be performed on it
- *choice*: used to determine which member for the IPC_SET control command that is to be changed
- *shmid_ds*: used to receive the specified shared memory segment identifier's data structure when an IPC_STAT control command is performed
- **buf*: a pointer passed to the system call which locates the data structure in the user memory area where the IPC_STAT control command is to place its return values or where the IPC_SET command gets the values to set.

Note that the *shmid_ds* data structure in this program (line 16) uses the data structure located in the *shm.h* header file of the same name as a template for its declaration. This is a perfect example of the advantage of local variables.

The next important thing to observe is that although the **buf* pointer is declared to be a pointer to a data structure of the *shmid_ds* type, it must also be initialized to contain the address of the user memory area data structure (line 17).

Now that all of the required declarations have been explained for this program, this is how it works.

First, the program prompts for a valid shared memory segment identifier that is stored in the *shmid* variable (lines 18–20). This is required for every *shmctl()* system call.

Then, the code for the desired control command must be entered (lines 21–29), and it is stored in the *command* variable. The code is tested to determine the control command for subsequent processing.

If the IPC_STAT control command is selected (code 1), the system call is performed (lines 39, 40) and the status information returned is printed out (lines 41–71). Note that if the system call is unsuccessful (line 146), the status information of the last successful call is printed out. In addition, an error message is displayed and the *errno* variable is printed out (lines 148, 149). If the system call is successful, a message indicates this along with the shared memory segment identifier used (lines 151–154).

If the IPC_SET control command is selected (code 2), the first thing done is to get the current status information for the message queue identifier specified (lines 90–92). This is necessary because this example program provides for changing only one member at a time, and the system call changes all of them. Also, if an invalid value happened to be stored in the user memory area for one of these members, it would cause repetitive failures for this control command until corrected. The next thing the program does is to prompt for a code corresponding to the member to be changed (lines 93–98). This code is stored in the *choice* variable (line 99). Now, depending upon the member picked, the program prompts for the new value (lines 105–127). The value is placed in the appropriate member in the user memory area data structure, and the system call is made (lines 128–130). Depending upon success or failure, the program returns the same messages as for IPC_STAT above.

If the IPC_RMID control command (code 3) is selected, the system call is performed (lines 132–135), and the *shmid* along with its associated message queue and data structure are removed from the UNIX operating system. Note that the **buf* pointer is not required as an argument to perform this control command and its value can be zero or NULL. Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the SHM_LOCK control command (code 4) is selected, the system call is performed (lines 137,138). Depending upon the success or failure, the program returns the same messages as for the other control commands.

If the SHM_UNLOCK control command (code 5) is selected, the system call is performed (lines 140–142). Depending upon the success or failure, the program returns the same messages as for the other control commands.

The example program for the *shmctl()* system call is shown in **Example 5–8**.

Example 5–8 *shmctl()* System Call Example

```
/*This is a program to illustrate
 *the shared memory control, shmctl(),
 *system call capabilities.
 */
/*Include necessary header files.*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
/*Start of main C language program*/
main()
{
 extern int errno;
 int uid, gid, mode;
 int rtrn, shmid, command, choice;
 struct shmid_ds shmid_ds, *buf;
 buf = &shmid_ds;
 /*Get the shmid, and command.*/
 printf("Enter the shmid = ");
 scanf("%d", &shmid);
 printf("\nEnter the number for\n");
 printf("the desired command:\n");
 printf("IPC_STAT = 1\n");
 printf("IPC_SET = 2\n");
 printf("IPC_RMID = 3\n");
 printf("SHM_LOCK = 4\n");
 printf("SHM_UNLOCK = 5\n");
 printf("Entry = ");
 scanf("%d", &command);
 /*Check the values.*/
 printf ("\nshmid =%d, command = %d\n",
 shmid, command);
 switch (command)
 {
 case 1: /*Use shmctl() to duplicate
  the data structure for
   shmid in the shmid_ds area pointed
   to by buf and then print it out.*/
  rtrn = shmctl(shmid, IPC_STAT,
   buf);
  printf ("\nThe USER ID = %d\n",
   buf->shm_perm.uid);
  printf ("The GROUP ID = %d\n",
   buf->shm_perm.gid);
  printf ("The creator's ID = %d\n",
   buf->shm_perm.cuid);
  printf ("The creator's group ID = %d\n",
   buf->shm_perm.cgid);
  printf ("The operation permissions = 0%o\n",
   buf->shm_perm.mode);
  printf ("The slot usage sequenceen");
  printf ("number = 0%x\n",
   buf->shm_perm.seq);
  printf ("The key= 0%x\n",
   buf->shm_perm.key);
  printf ("The segment size = %d\n",
   buf->shm_segsz);
  printf ("The pid of last shmop = %d\n",
```

```
59     buf->shm_lpid);
60    printf ("The pid of creator = %d\n",
61     buf->shm_cpid);
62    printf ("The current # attached = %d\n",
63     buf->shm_nattch);
64    printf("The in memory # attached = %d\n",
65     buf->shm_cnattach);
66    printf("The last shmat time = %d\n",
67     buf->shm_atime);
68    printf("The last shmdt time = %d\n",
69     buf->shm_dtime);
70    printf("The last change time = %d\n",
71     buf->shm_ctime);
72    break;
   /* Lines 73 - 87 deleted */
88   case 2: /*Select and change the desired
89       member(s) of the data structure.*/
90    /*Get the original data for this shmid
91       data structure first.*/
92    rtrn = shmctl(shmid, IPC_STAT, buf);
93    printf("\nEnter the number for the\n");
94    printf("member to be changed:\n");
95    printf("shm_perm.uid = 1\n");
96    printf("shm_perm.gid = 2\n");
97    printf("shm_perm.mode = 3\n");
98    printf("Entry = ");
99    scanf("%d", &choice);
100    /*Only one choice is allowed per
101    pass as an illegal entry will
102      cause repetitive failures until
103     shmid_ds is updated with
104      IPC_STAT.*/
105    switch(choice){
106    case 1:
107     printf("\nEnter USER ID = ");
108     scanf ("%d", &uid);
109     buf->shm_perm.uid = uid;
110     printf("\nUSER ID = %d\n",
111     buf->shm_perm.uid);
112     break;
113    case 2:
114     printf("\nEnter GROUP ID = ");
115     scanf("%d", &gid);
116     buf->shm_perm.gid = gid;
117     printf("\nGROUP ID = %d\n",
118     buf->shm_perm.gid);
119     break;

120    case 3:
121     printf("\nEnter MODE = ");
122     scanf("%o", &mode);
123     buf->shm_perm.mode = mode;
124     printf("\nMODE = 0%o\n",
125      buf->shm_perm.mode);
126     break;
127    }
128    /*Do the change.*/
129    rtrn = shmctl(shmid, IPC_SET,
```

```
    buf);
   break;
   case 3: /*Remove the shmid along with its
       associated
       data structure.*/
    rtrn = shmctl(shmid, IPC_RMID, NULL);
    break;
   case 4: /*Lock the shared memory segment*/
    rtrn = shmctl(shmid, SHM_LOCK, NULL);
    break;
   case 5: /*Unlock the shared memory
       segment.*/
    rtrn = shmctl(shmid, SHM_UNLOCK, NULL);
    break;
   }
   /*Perform the following if the call fails.*/
   if(rtrn == -1)
   {
    printf ("\nThe shmctl system call failed!\n");
    printf ("The error number = %d\n", errno);
   }
   /*Return the shmid upon successful completion.*/
   else
    printf ("\nShmctl was successful for shmid = %d\n",
     shmid);
   exit (0);
  }
```

Operations for Shared Memory: shmat() and shmdt()

This section details the *shmat()* and *shmdt()* system calls, and presents an example program that exercises all of their capabilities.

The synopsis found in the *shmop(2)* manual page, which includes both *shmat()* and *shmaddr()*, is as follows:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

char *shmat (shmid, shmaddr, shmflg)
int shmid;
char *shmaddr;
int shmflg;

int shmdt (shmaddr)
char *shmaddr;
```

Attaching to a Shared Memory Segment

The *shmat()* system call requires three arguments to be passed to it, and it returns a character pointer value.

The system call can be cast to return an integer value. Upon successful completion, this value is the address in core memory where the process is attached to the shared memory segment, and when unsuccessful, it is −1.

The *shmid* argument must be a valid, non−negative, integer value. In other words, it must have already been created by using the *shmget()* system call.

The *shmaddr* argument can be zero or user−supplied when passed to the *shmat()* system call. If it is zero, the UNIX operating system picks the address of where the shared memory segment will be attached. If it is user supplied, the address must be a valid address that the UNIX operating system would pick. Here are some typical address ranges:

0xc00c0000

0xc00e0000

0xc0100000

0xc0120000

Note that these addresses are in chunks of 20,000 hexadecimal. It would be wise to let the operating system pick addresses so as to improve portability.

The *shmflg* argument is used to pass the SHM_RND and SHM_RDONLY flags to the *shmat()* system call.

Further details are discussed in the example program for *shmop()*. If you have problems understanding the logic manipulations in this program, read **"Getting Shared Memory Segments with shmget()"** ; it goes into more detail than would be practical to do for every system call.

Detaching Shared Memory Segments

The *shmdt()* system call requires one argument to be passed to it, and it returns an integer value.

shmdt() returns zero if it completes successfully; otherwise, it returns −1.

Further details of this system call are discussed in the example program. If you have problems understanding the logic manipulations in this program, read **"Getting Shared Memory Segments with shmget()"** ; it goes into more detail than what would be practical to do for every system call.

Example Program

The example program in this section (Figure 8−17) is a menu−driven program that exercises all possible combinations of the *shmat()* and *shmdt()* system calls.

From studying this program, you can observe the method of passing arguments and receiving return values. The program requirements are pointed out.

This program begins (lines 5−9) by including the required header files as specified by the *shmop(*2*)* manual page. Note that in this program, *errno* is declared as an external variable, and therefore, the *errno.h* header file does not have to be included.

Variable and structure names were chosen to be as close as possible to those in the synopsis. Their declarations are self−explanatory. The names make the program more readable; this is legal since they are local to the program. The variables declared for this program and their purposes include:

- *flags*: used to store the codes of SHM_RND or SHM_RDONLY for the *shmat()* system call
- *addr*: used to store the address of the shared memory segment for the *shmat()* and *shmdt()* system calls
- *i*: used as a loop counter for attaching and detaching
- *attach*: used to store the desired number of attach operations
- *shmid*: used to store and pass the desired shared memory segment identifier
- *shmflg*: used to pass the value of flags to the *shmat()* system call
- *rtrn*: used to store the return values from both system calls
- *detach*: used to store the desired number of detach operations

This example program combines both the *shmat()* and *shmdt()* system calls. The program prompts for the number of attachments, and enters a loop until they are done, for the specified shared memory identifiers. Then, the program prompts for the number of detachments to be performed, and enters a loop until they are done, for the specified shared memory segment addresses.

shmat()

The program prompts for the number of attachments to be performed, and the value is stored in the *attach* variable (lines 17−21).

A loop is entered using the attach variable and the *i* counter (lines 23−70) to perform the specified number of attachments.

In this loop, the program prompts for a shared memory segment identifier (lines 24−27) and it is stored in the *shmid* variable (line 28). Next, the program prompts for the address where the segment is to be attached (lines 30−34), and it is stored in the *addr* variable (line 35). Then, the program prompts for the desired flags to be used for the attachment (lines 37−44), and the code representing the flags is stored in the *flags* variable (line 45). The *flags* variable is tested to determine the code to be stored for the *shmflg* variable used to pass them to the *shmat()* system call (lines 46−57). The system call is made (line 60). If successful, a message stating so is displayed along with the attach address (lines 66−68). If unsuccessful, a message stating so is displayed and the error code is displayed (lines 62, 63). The loop then continues until it finishes.

shmdt

After the attach loop completes, the program prompts for the number of detach operations to be performed (lines 71−75), and the value is stored in the *detach* variable (line 76).

A loop is entered using the detach variable and the *i* counter (lines 78−95) to perform the specified number of detachments.

In this loop, the program prompts for the address of the shared memory segment to be detached (lines 79−83), and it is stored in the *addr* variable (line 84). Then, the *shmdt()* system call is performed (line 87). If successful, a message stating so is displayed along with the address that the segment was detached from (lines 92,93). If unsuccessful, the error number is displayed (line 89). The loop continues until it finishes.

The example program for the *shmop()* system calls is shown in **Example 5−9**.

Example 5−9 *shmop()* System Call Example

```
1  /*This is a program to illustrate
2   *the shared memory operations, shmop(),
3   *system call capabilities.
4   */
5  /*Include necessary header files.*/
6  #include <stdio.h>
7  #include <sys/types.h>
8  #include <sys/ipc.h>
9  #include <sys/shm.h>
10  /*Start of main C language program*/
11  main()
12  {
13   extern int errno;
14   int flags, addr, i, attach;
15   int shmid, shmflg, retrn, detach;
16   /*Loop for attachments by this process.*/
17   printf("Enter the number ofen");
18   printf("attachments for this\n");
19   printf("process (1−4).\n");
20   printf(" Attachments = ");
21   scanf("%d", &attach);
22   printf("Number of attaches = %d\n", attach);
23   for(i = 1; i <= attach; i++) {
24    /*Enter the shared memory ID.*/
25    printf("\nEnter the shmid ofen");
26    printf("the shared memory segment to\n");
27    printf("be operated on = ");
28    scanf("%d", &shmid);
29    printf("\nshmid = %d\n", shmid);
30    /*Enter the value for shmaddr.*/
31    printf("\nEnter the value for\n");
32    printf("the shared memory address\n");
```

```
 printf("in hexadecimal:\n");
 printf(" Shmaddr = ");
 scanf("%x", &addr);
 printf("The desired address = 0x%x\n", addr);
 /*Specify the desired flags.*/
 printf("\nEnter the corresponding\n");
 printf("number for the desireden");
 printf("flags:\n");
 printf("SHM_RND = 1\n");
 printf("SHM_RDONLY = 2\n");
 printf("SHM_RND and SHM_RDONLY = 3\n");
 printf(" Flags = ");
 scanf("%d", &flags);
 switch(flags)
 {
 case 1:
  shmflg = SHM_RND;
  break;
 case 2:
  shmflg = SHM_RDONLY;
  break;
 case 3:
  shmflg = SHM_RND | SHM_RDONLY;
  break;
 }
 printf("\nFlags = 0%o\n", shmflg);
 /*Do the shmat system call.*/
 retrn = (int)shmat(shmid, addr, shmflg);
 if(retrn == -1) {
  printf("\nShmat failed. ");
  printf("Error = %d\n", errno);
 }
 else {
  printf ("\nShmat was successful\n");
  printf("for shmid = %d\n", shmid);
  printf("The address = 0x%x\n", retrn);
 }
}
/*Loop for detachments by this process.*/
printf("Enter the number ofen");
printf("detachments for this\n");
printf("process (1-4).\n");
printf(" Detachments = ");
scanf("%d", &detach);
printf("Number of attaches = %d\n", detach);
for(i = 1; i <= detach; i++) {
 /*Enter the value for shmaddr.*/
 printf("\nEnter the value for\n");
 printf("the shared memory address\n");
 printf("in hexadecimal:\n");
 printf(" Shmaddr = ");
 scanf("%x", &addr);
 printf("The desired address = 0x%x\n", addr);
 /*Do the shmdt system call.*/
 retrn = (int)shmdt(addr);
 if(retrn == -1) {
  printf("Error = %d\n", errno);
 }
 else {
```

```
   printf ("\nShmdt was successful\n");
   printf("for address = 0%x\n", addr);
  }
 }
}
```

IRIX IPC

To meet the demands of parallel programming, IRIX provides a set of fast, low-overhead inter-process communication mechanisms. These mechanisms are powerful and easy to use. However, remember that they are IRIX-specific, so code using them is not portable to other systems.

Unlike System V IPC mechanisms that use their own namespace, IRIX IPC mechanisms are associated with the filesystem namespace. To begin using IRIX IPC, a process must specify the name of a file to be used as a *shared arena*. All processes using the same arena have access to the same set of IPC mechanisms. This makes it relatively easy for unrelated processes to communicate using IRIX IPC. The shared arena file is mapped into the process's user space, which means that most of the shared arena IPC functions do not have to make system calls. This is one reason that the overhead on IRIX IPC is lower than the overhead on standard System V IPC.

IRIX IPC comprises four main mechanisms, two of which—semaphores and shared memory—resemble their System V counterparts. The other mechanisms are *spinlocks*, simple busy-wait locks for low-level synchronization, and *barriers*, which provide rendezvous points for multiple processes.

Note: Modules using IRIX IPC routines should include *<stdio.h>* and *<ulocks.h>*, and should be linked with the *libmpc.so* shared object (*-lmpc*).

For more information on any of the routines presented here, see the appropriate manual pages.

Initializing the Shared Arena

To begin using IRIX IPC, call *usinit()*.

Syntax

```
#include <stdio.h>
#include <ulocks.h>
usptr_t *usinit (const char *filename);
```

The *filename* variable specifies a file to use as the shared arena. If the file doesn't exist, *usinit()* creates it. If a shared arena exists by that name, *usinit()* joins the shared arena. If the file exists but isn't a shared arena, *usinit()* overwrites it. In any case, *usinit()* is subject to regular filesystem permissions: it returns an error if the process doesn't have read and write permission on the file (if it already exists) or permission to create the file (if it doesn't exist).

Using Shared-Arena Semaphores

To allocate a new shared-arena semaphore, call *usnewsema()*.

Syntax

```
#include <ulocks.h>
usptr_t *usnewsema (usptr_t *handle, int initial_value);
```

The *initial_value* argument simply specifies the initial value of the semaphore. An initial value of zero is commonly used for a synchronization semaphore: the first process that attempts a P operation on the semaphore blocks until another process performs a V operation on the semaphore. An initial value of 1 may be used to provide a simple mutual exclusion semaphore: the first process attempting a P operation on the semaphore succeeds, and subsequent processes block until the first process releases the semaphore by performing a V operation on it.

Changing the Values of Shared-Arena Semaphores

To perform P and V operations on shared arena semaphores, use the *uspsema()* and *usvsema()* functions. The *uscpsema()* function provides a conditional P operation: it performs a P operation on the semaphore only if it can do so without blocking. The *ustestsema()* function returns the current value of the semaphore (useful primarily for debugging), and the *usinitsema()* function reinitializes the semaphore to a specified value. Note that if you reinitialize a semaphore on which processes are waiting, the processes will not be woken.

Syntax

```
#include <ulocks.h>
int uspsema (usema_t *sema);
int uscpsema (usema_t *sema);
int usvsema (usema_t *sema);
int ustestsema (usema_t *sema);
int usinitsema (usema_t *sema, int initial_value);
```

The following code fragment demonstrates the use of *uspsema()*, *usvsema()*, and *uscpsema():*

```
char *arenafile = "/usr/tmp/testarena";
usptr_t arena;
usema_t *sema;
/* open an arena, allocate a semaphore */
arena = usinit(arenafile);
if (arena == NULL)
{
    /* error */
}
sema = usnewsema(arena);
if (sema == NULL)
{
    /* error */
}
/* acquire the semaphore */
uspsema(sema);
/* release the semaphore */
usvsema(sema);
/* try to get the semaphore again, without blocking */
if (uscpsema(sema) == 1)
{
   /*we succeeded, so we have to release the semaphore again*/
    usvsema(sema);
}
else
    /* failed to get the semaphore */
```

Using Spinlocks

Spinlocks are somewhat like semaphores. Instead of having an integer value, a spinlock has a binary value: it is either locked or unlocked.

The way that spinlocks are implemented depends upon the hardware architecture of the computer using them. On multiprocessor computers, spinlocks are busy-wait locks, so the processor continually tries to acquire the lock until it succeeds. This implementation only makes sense on multiprocessor systems, in which one processor can release the lock while another processor is "spinning" trying to acquire the lock. On single processor machines, spinlocks are implemented using the same algorithm as semaphores—that is, processes waiting to acquire a lock may be put to sleep until the lock is released by another process.

To allocate a new spinlock, use *usnewlock*():

Syntax

```
#include <ulocks.h>
ulock_t usnewlock (usptr_t *arena);
```

The *usnewlock*() function returns a lock, which may then be used by the process. Locking and unlocking can be performed with the functions *ussetlock*(), *uscsetlock*(), *uswsetlock*(), and *usunsetlock*().

Syntax

```
#include <ulocks.h>
int ussetlock (ulock_t lock);
int uscsetlock (ulock_t lock, unsigned spins);
int uswsetlock (ulock_t lock, unsigned spins);
int usunsetlock (ulock_t lock);
```

The *ussetlock()* function locks the specified lock. On multiprocessor systems it spins until it succeeds; on single−processor systems, it may sleep until it can set the lock. It returns 1, unless it encounters an error. The *uscsetlock()* function tries to set the lock *spins* times; if it succeeds, it returns 1, if it fails, it returns 0. On a single−processor system, uscsetlock() ignores the spins argument and only sets the lock if it can do so without waiting. The *uswsetlock()* function resembles *ussetlock()* except that after every *spins* attempts to set the lock, it yields the processor by calling *sginap()*. On single−processor systems, *uswsetlock()* behaves exactly like *ussetlock()*. The *usunsetlock()* function unlocks the lock; it always returns 0, unless it encounters an error. All of these functions return −1 in the event of an error. You can find the reason for the error by calling *oserror()*.

IRIX also provides a function to test whether a given lock is locked or unlocked: *ustestlock()*. *ustestlock()* returns 1 if the lock is set, and 0 if the lock is not set.

Syntax

```
#include <ulocks.h>
int ustestlock (ulock_t lock);
```

A process can call *usunsetlock()* on a lock that is either not locked or locked by another process. In either case, the lock is unlocked. Double tripping-that is, calling a set lock function twice with the same lock-is also permissible. The caller blocks until another process unsets the lock.

Using Barriers

Barriers provide a convenient way of synchronizing parallel processes on multiprocessor systems. To use a barrier, you must first allocate one by calling *new_barrier()*, which returns a pointer to an initialized barrier structure. You must then communicate this pointer to the other processes (for example, by placing it in a data structure in shared memory). To arrange a rendezvous, have each process call *barrier()*, passing it the pointer to the barrier structure and a numerical argument specifying the number of processes to wait for. Each process blocks until the last process calls *barrier()*. Barriers are always busy−wait, so they aren't suitable for use on single−processor systems.

Syntax

```
#include <ulocks.h>
barrier_t *new_barrier (usptr_t *handle);
void barrier (barrier_t *b, unsigned n);
void free_barrier (barrier_t *b);
void init_barrier (barrier_t *b);
```

The *free_barrier()* function releases the resources associated with a barrier. The *init_barrier()* function restores a barrier to its original state, as if it had just been allocated by *new_barrier()*.

Using IRIX Shared Memory

Allocating shared memory from a shared arena is much like the regular process of allocating memory using the *malloc()* and *free()* library routines.

Syntax

```
#include <ulocks.h>
#include <malloc.h>
void *usmalloc (size_t size, usptr_t *handle);
void usfree (void *ptr, usptr_t *handle);
void *usrealloc (void *ptr, size_t size, usptr_t *handle);
void *uscalloc (size_t nelem, size_t elsize, usptr_t *handle);
int usmallopt (int cmd, int value, usptr_t *handle);
struct mallinfo usmallinfo (usptr_t *handle);
```

Memory allocated using *usmalloc()* and related functions can be accessed by all processes that have joined the shared arena specified by *handle*. Other than that, these functions operate in the same way as their single−threaded cousins.

Exchanging the First Datum

Once you've established a shared arena, it's frequently useful to communicate the location of some object in that arena to another process using the arena. For example, one process might create a data structure in shared memory, and pass the address of the data structure to other processes using the arena. The shared arena has a special one−word area for storing such data. This area is accessed using the calls *usputinfo(), usgetinfo(),* and *uscasinfo().* The first two are fairly self−explanatory: *usputinfo()* puts a word of data into the storage area, and *usgetinfo()* returns the current value in the storage area. *uscasinfo()* is explained later.

Syntax

```
#include <ulocks.h>
void usputinfo (usptr_t *handle, void *info);
void *usgetinfo (usptr_t *handle);
```

The following program fragment initializes an arena, allocates space for a data structure, and places the address of the data structure in the storage area. The exact contents of the data structure are left as an exercise for the reader:

```
#include <stdio.h>
#include <ulocks.h>
    /* ... */
    usptr_t *arena;
    char *arenafile = "/usr/tmp/testarena";
    struct controlStruct *control;
    arena = usinit(arenafile);
    if (arena == NULL) {
      /* error */
    }
    control = usmalloc(sizeof(struct controlStruct), arena);
    if (control == NULL) {
      /* error */
    }
    usputinfo(arena, control);
```

The next program fragment accesses the same shared arena created by the previous program fragment, and obtains the address of the previously allocated data structure:

```
#include <stdio.h>
```

```
#include <ulocks.h>
    /* ... */
    usptr_t *arena;
    char *arenafile = "/usr/tmp/testarena";
    struct controlStruct *control;
    arena = usinit(arenafile);
    if (arena == NULL) {
      /* error */
    }
    control = usgetinfo(arena);
    if (control == NULL) {
      /* error */
    }
```

The simple technique outlined above works fine if you know which process is going to create the arena (for example, if program #1 creates the arena before *fork*ing and *exec*ing program #2). However, if several processes are started independently and they all try to access the same arena, the situation becomes more complicated. You may want one of the processes to set up data structures for the others to use. In this instance, IRIX provides an atomic compare−and−swap operator, *uscasinfo()*, which compares the current value in the storage area with a specified value. If these two are equal, *uscasinfo()* changes the value in the storage area to a second specified value.

Syntax

```
#include <ulocks.h>
int uscasinfo (usptr_t *arena, void *oldinfo, void *newinfo);
```

If the value in the storage area is equal to *oldinfo*, *uscasinfo()* changes the value to *newinfo* and returns 1. Otherwise, *uscasinfo()* leaves the value untouched and returns 0. Usually, *uscasinfo()* is called with *oldinfo* equal to 0. The value in the storage area is 0 if no one has placed any data in it yet.

The following code example illustrates race considerations at the termination of one process and the start of a new process when the arena file gets deleted at termination.

```
YAP_S_PLAYER * yap_s_open(
   char * ifname,  /* File name of yap file */
 YAP_D_FLAGS options,  /* Open options */
 int cr_flags,  /* Create operating system flags */
 unsigned cr_len,  /* create length */
 void * cr_addr,  /* Create address for yap buff */
 unsigned cr_mprim,  /* Creation max prim id */
 unsigned cr_slen,  /* Create length of sync buffer */
 void * cr_saddr,  /* Create address for sync buffer */
 unsigned cr_maxproc  /* Maximum number of processes */
)
{

   int  fd;
 YAP_D_HEAD  * p_d_head;
 YAP_S_HEAD  * p_head;
 YAP_S_HEAD  * p_head2;
 YAP_S_PLAYER  * p_player;
 int  v1, v2, v3, v4;
 char  cwd[ YAP_S_MAXFNAME ];
 char  buf[ YAP_S_MAXFNAME ];
 char  nfname[ YAP_S_MAXFNAME ];
 char  * fname;
 int  fnamelen;
 int  dirlen;
 usptr_t  * usptr;
 int  creatdir;
```

```
 int  creatdir;
 struct  stat p_stat[1];
 int  mmapsiz;
 int  attempts = 0;
 int  nusers;

/* Make sure the file name is absolute. */
if ( * ifname != '/' ) {
   /* Make an absolute file name. */
   if ( ! getcwd( cwd, sizeof( cwd ) ) ) {
   /* If an absolute file name isn't given then we need */
   /* to be able to do a getcwd ! */
   yap_seterror( YAP_S_E_GETCWDN, errno );
   return 0;
   }

   fname = nfname;
   sprintf( fname, "%s/%s", cwd, ifname );
   } else {
   fname = ifname;
   }

/* Create the directory name to place all the sync files */
fnamelen = strlen( fname );
if ( fnamelen >= ( YAP_S_MAXFNAME - 24 ) ) {
   /* Filename too long */
   yap_seterror( YAP_S_E_FNLEN, 0 );
   return 0;
}

tryitagain:
/* Don't get carried away ! */
if ( attempts ++ > 10 ) {
   /* Unlucky process. Another process keeps getting out */
   /* of the buffer and deleting it before I get in*/
   yap_seterror( YAP_S_E_FLUNLNK, 0 );
   return 0;
   }

   /* Make fname__s_ name so we can reference the directory */

   sprintf( buf, "%s%s", fname, YAP_S_SYNCAPPEND );

   /* Create the directory name */
   if ( mkdir( buf, cr_flags | 0111 ) < 0 ) {

   if ( errno != EEXIST ) {
   yap_seterror( YAP_S_E_MKDIR, errno );
   return 0;
   }
   creatdir = 0;
   } else {
   creatdir = 1;
   }

/* Directory exists - then let's open the us file. */
sprintf( buf, "%s%s/%s", fname, YAP_S_SYNCAPPEND, YAP_S_SYNCFILE );

   /* increase the size of the us arena to 1 meg */
```

```
usconfig( CONF_INITSIZE, cr_slen );
usconfig( CONF_INITUSERS, cr_maxproc );
usconfig( CONF_CHMOD, cr_flags );

/* If we have defined an attach address then use it. */
if ( cr_saddr ) {
usconfig( CONF_ATTACHADDR, cr_saddr );
}

/* Open the sync file. */
mmapsiz = _us_mapsize;
usptr = usinit( buf );

/* did it open ? */
if ( ! usptr ) {
yap_seterror( YAP_S_E_USINIT, errno );
/* did we create the directory ? - if so delete it */
if ( creatdir ) {
buf[ dirlen ] = 0;
rmdir( buf );
}
return 0;
}

/* Got to the sync memory ! */
/* Now we have to attach ourselves or initialize memory */

/* Is there a head in the arena yet ? */
p_head = ( YAP_S_HEAD * ) usgetinfo( usptr );

if ( ! p_head ) {

p_head = ( YAP_S_HEAD * ) usmalloc( sizeof( * p_head ), usptr );
if ( ! p_head ) {
yap_seterror( YAP_S_E_USMALLOC, errno );

/* did we create the directory ? - if so delete it */
if ( creatdir ) {
buf[ dirlen ] = 0;
rmdir( buf );
}
return 0;
}

YAP_S_ZERO( p_head );

DBG_TYPSET( p_head, YAP_S_HEAD )

/* We need to be able to unmap ourselves */
p_head->yap_s_mapsiz = mmapsiz;
p_head->yap_s_ushandle = usptr;

p_head->yap_s_startlock = usnewlock( usptr );

if ( ! uscsetlock( p_head->yap_s_startlock, 1 ) ) {
/* Must be able to get the lock I allocated but didn't*/
yap_seterror( YAP_S_E_SETLCKFL, errno );
yap_s_unmap( p_head );
```

```
   /* Can’t decide if I should unlink everything or not ? */
   return 0;
   }

doitagain:
/* Compare and swap the p_head structure */
   if ( ! uscasinfo( usptr, 0, p_head ) ) {

   p_head2 = ( YAP_S_HEAD * ) usgetinfo( usptr );

   if ( ! p_head2 ) {
   /* Need to try again! */
   goto doitagain;
   }
   /* Someone else beat me to it! */

   /* Free my data. */
   usfreelock( p_head->yap_s_startlock, usptr );
   usfree( p_head, usptr );

   p_head = ( YAP_S_HEAD * ) usgetinfo( usptr );

   /* This must be a valid number now */
   if ( ! p_head ) {
   /* Internal error !! */
   yap_seterror( YAP_S_E_INTINIT, 0 );
   munmap( usptr, mmapsiz ); /* have no valid p_head */
   return 0;
   }

   DBG_TYPCHK( p_head, YAP_S_HEAD )

   } else {

   error_recovery:

#if YAP_USINITBUGCHECK
   if ( yap_s_usinitbugflag ) {
   fprintf(
   stderr,
   "File %s:%d Process %d setting up arena\n",
   __FILE__, __LINE__,
   getpid()
   );
   fflush( stderr );
   }
#endif

   /* try to get rev number from the RCS revision string*/
   v1 = v2 = v3 = v4 = 0;
   sscanf(
yap_s_revision + 1, "Revision: %d.%d.%d.%d", &v1, &v2, &v3, &v4
   );
   p_head->yap_s_rcsver = YAP_D_VERMACRO( v1, v2, v3, v4 );

   p_head->yap_s_version = YAP_S_VERSION;

   /* Store the name of the file for later */
   thep_head->yap_s_fname = ( char * ) yap_s_usmalloc(
```

```
   fnamelen + 1, p_head
   );
   memcpy( p_head->yap_s_fname, fname, fnamelen + 1 );

   /* Open the data buffer */
   p_d_head = yap_d_open(
fname, options, cr_flags, cr_len, cr_addr, cr_mprim
   );

   /* Did the file get created ? */
   if ( ! p_d_head ) {
   /* we need to get out of here. */
   get_out_fast:
   p_head->yap_s_unlinked = 1;
   unlink( buf );
   buf[ dirlen ] = 0;
   rmdir( buf );
   usunsetlock( p_head->yap_s_startlock );
   yap_s_unmap( p_head );
   return 0;
   }

   /* Make an access lock with writer priority */
p_head->yap_s_tbreflck = usnewaccess( usptr, USX_WRITEPRIO );

   /* Set the head pointer */
   p_head->yap_s_dhead = p_d_head;
   p_d_head->yap_d_alloclock = usnewlock( usptr );
   p_head->yap_s_tarlock = usnewlock( usptr );
   p_head->yap_s_dtabqlck = usnewlock( usptr );
   p_head->yap_s_tblcdlck = usnewlock( usptr );

   p_head->yap_s_dmsgqlck = usnewlock( usptr );
   p_head->yap_s_dmsgq = 0;
   p_head->yap_s_msgalc = 0;
   p_head->yap_s_dmsgs = 0;

#if YAP_S_COARSELOCK
   p_head->yap_s_tbdatalck = usnewaccess( usptr, USX_WRITEPRIO );
   p_head->yap_s_masterlock = usnewlock( usptr );
#endif
   /* Create the array of tables. */
   /* Create some expansion room too */
   p_head->yap_s_mtar = p_d_head->yap_d_mtablenum * 2 + 64;

   p_head->yap_s_tar = ( YAP_S_TABLE ** ) yap_s_usmalloc(
   sizeof( YAP_S_TABLE * ) * p_head->yap_s_mtar,
   p_head
   );

   YAP_S_AZERO( p_head->yap_s_tar, p_head->yap_s_mtar );

   /* Initialize the player’s data area */
   nusers = usconfig( CONF_GETUSERS, usptr );

   /* Set up the player’s array */
   p_head->yap_s_mpar = nusers;
   p_head->yap_s_par = ( YAP_S_PLAYER * ) yap_s_usmalloc(
```

```
   nusers * sizeof( YAP_S_PLAYER ),
   p_head
   );

   /* Set the array to zero. */
   YAP_S_AZERO( p_head->yap_s_par, nusers );

   /* Initialize myself as the first user since I have the */
   /* start lock. */
   p_player = yap_s_register( p_head );

   /* did I bomb out */
   if ( ! p_player ) {
   p_d_head->yap_d_alloclock = 0;
   yap_d_close( p_d_head );
   goto get_out_fast;
   }

   /* Everything is set up - we can get out. */
   usunsetlock( p_head->yap_s_startlock );

   return p_player;
   }
}

ussetlock( p_head->yap_s_startlock );
   /* In case init process fails or file gets unlinked */
   /* before I get registered check the yap_s_unlinked flag. */
   if ( p_head->yap_s_unlinked ) {
   /* The file got unlinked before I could register. */
   /* Let's try again - first unmap myself. */
   usunsetlock( p_head->yap_s_startlock );
   yap_s_unmap( p_head );
   goto tryitagain;
   }

   p_d_head = yap_d_open(
   fname, options, cr_flags, cr_len, cr_addr, cr_mprim
   );
   p_player = yap_s_register( p_head );

   usunsetlock( p_head->yap_s_startlock );

   return p_player;

} /* end yap_s_open */
```

To run this example program, begin four instances of the program from the command line. For example:

```
uscastest &
uscastest &
uscastest &
uscastest &
```

Chapter 6

File and Record Locking

An Overview of File and Record Locking

Mandatory and advisory file and record locking are available on many current releases of the UNIX system. The intent of these capabilities is to provide a synchronization mechanism for programs accessing the same stores of data simultaneously. Such processing is characteristic of many multi−user applications, and the need for a standard method of dealing with the problem has been recognized by standards advocates like */usr/group*, an organization of UNIX system users from businesses and campuses across the country.

Advisory file and record locking can be used to coordinate self−synchronizing processes. In mandatory locking, on the other hand, the standard I/O subroutines and I/O system calls enforce the locking protocol. In this way, at the cost of a little efficiency, mandatory locking double−checks the programs to avoid accessing the data out of sequence.

This chapter describes how file and record locking capabilities can be used. Examples are given for the correct use of record locking. Misconceptions about the amount of protection that record locking affords are dispelled. Record locking should be viewed as a synchronization mechanism, not a security mechanism.

The manual pages for the *fcntl(2)* system call, the *lockf(3)* library function, and *fcntl*(5) data structures and commands are referred to throughout this section. You should read them before continuing.

Terminology

Before discussing how record locking should be used, let us first define a few terms.

Record A contiguous set of bytes in a file. The UNIX operating system does not impose any record structure on files. Such structure may be imposed by the programs that use the files.

Cooperating Processes

Processes that work together in some well−defined fashion to accomplish the tasks at hand. Processes that share files must request permission to access the files before using them. File access permissions must be carefully set to restrict non−cooperating processes from accessing those files. The term "process" is used interchangeably with "cooperating process" to refer to a task obeying such protocols.

Read (Share) Locks

These are used to gain limited access to sections of files. When a read lock is in place on a record, other processes may also read lock that record, in whole or in part. No other process, however, may have or obtain a write lock on an overlapping section of the file. If a process holds a read lock it may assume that no other process will be writing or updating that record at the same time. This access method also permits many processes to read the given record. This might be necessary when searching a file, without the contention involved if a write or exclusive lock were to be used.

Write (Exclusive) Lock

These are used to gain complete control over sections of files. When a write lock is in place on a record, no other process may read− or write−lock that record, in whole or in part. If a process holds a write lock it may assume that no other process will be reading or writing that record at the same time.

Advisory Locking

A form of record locking that does not interact with the I/O subsystem (which includes *creat(2)*, *open(2)*, *read(2)*, and *write(2)*). The control over records is accomplished by requiring an appropriate record lock request before I/O operations. If appropriate requests are always made by all processes accessing the file, then the accessibility of the file is controlled by the interaction of these requests. Advisory locking depends on the individual processes to enforce the record locking protocol; it does not require an accessibility check at the time of each I/O request.

Mandatory Locking

A form of record locking that does interact with the I/O subsystem. Access to locked records is enforced by the *creat(2)*, *open(2)*, *read(2)*, and *write(2)* system calls. If a record is locked, then access to that record by any other process is restricted performed explicitly by requesting an appropriate record lock before I/O operations, but an additional check is made by the system before each I/O operation to ensure the record locking protocol is being honored. Mandatory locking offers an extra synchronization check, but at the cost of some additional system overhead.

File Protection

There are access permissions for each UNIX file to control who may read, write, or execute the file. These access permissions may only be set by the owner of the file or by the superuser. The permissions of the directory in which the file resides can also affect the access permissions for a file. Note that if the permissions for a directory allow anyone to write in the directory, then files within that directory may be removed even by a user who does not have read, write, or execute permission for those files. Any information that is worth protecting, is worth protecting properly. If your application warrants the use of record locking, make sure that the permissions on your files and directories are set properly. A record lock, even a mandatory record lock, only protects the portions of the files that are locked. Other parts of the files can be corrupted if proper precautions are not taken.

Only a known set of programs and/or administrators should be able to read or write a database. This can be done easily by setting the set−group−ID bit (see *chmod*(1)) of the database accessing programs. The files can then be accessed by a known set of programs that obey the record−locking protocol. An example of such file protection, although record locking is not used, is the *mail*(1) command. In that command only the owning user and the *mail* command can read and write the unread mail files.

Opening a File for Record Locking

The first requirement for locking a file or segment of a file is having a valid open file descriptor. If read locks are to be used, then the file must be opened with at least read access; likewise for write locks and write access. In this example we open a file for both read and write access:

```
#include <stdio.h>
#include <errno.h>
#include <fcntl.h>
int fd;    /* file descriptor */
char *filename;

main(argc, argv)
int argc;
char *argv[];
{
      extern void exit(), perror();

      /* get database file name from command line and open the
       * file for read and write access.
       */
      if (argc < 2) {
          (void) fprintf(stderr, "usage: %s filename\n", argv[0]);
          exit(2);
      }
      filename = argv[1];
      fd = open(filename, O_RDWR);
      if (fd < 0) {
          perror(filename);
          exit(2);
      }
}
```

The file is now open for us to perform both locking and I/O functions. We then proceed with the task of setting a lock.

Setting a File Lock

Several ways exist to set a lock on a file. These methods depend upon how the lock interacts with the rest of the program. Questions of portability and performance exist. Two methods are given: using the *fcntl(2)* system call and using the */usr/group* standards−compatible *lockf(3)* library function call.

Locking an entire file is just a special case of record locking. For both these methods the concept and the effect of the lock are the same. The file is locked starting at a byte offset of zero and going until the end of the maximum file size. This point is beyond any real end−of−file so that no other lock can be placed on the file. To set such a lock, set the size of the lock to zero. Here is a sample code fragment using the *fcntl()* system call:

```
#include <fcntl.h>
#define MAX_TRY 10
int try;
struct flock lck;

try = 0;
/* set up the record locking structure, the address of which
   * is passed to the fcntl() system call. */
lck.l_type = F_WRLCK; /* setting a write lock */
lck.l_whence = 0; /* offset l_start from beginning of file */
lck.l_start = 0L;
lck.l_len = 0L; /* until the end of the file address space */
/* Attempt locking MAX_TRY times before giving up. */
while (fcntl(fd, F_SETLK, &lck) < 0) {
      if (errno == EAGAIN || errno == EACCES) {
          /* there might be other error cases in which
           * you might try again.
           */
          if (++try < MAX_TRY) {
              (void) sleep(2);
              continue;
          }
          (void) fprintf(stderr,"File busy try again later!\n");
          return;
      }
      perror("fcntl");
      exit(2);
}
...
```

This piece of code tries to lock a file. The lock is attempted several times until one of the following things happens:

- the file is locked
- an error occurs
- the program exceeds MAX_TRY and gives up

To perform the same task using the *lockf()* function, use code like this:

```
#include <unistd.h>
#define MAX_TRY 10
int try;
try = 0;

/* make sure the file ptr is at the beginning of file. */
lseek(fd, 0L, 0);

/* Attempt locking MAX_TRY times before giving up. */
while (lockf(fd, F_TLOCK, 0L) < 0) {
      if (errno == EAGAIN || errno == EACCES) {
```

```
          /* there might be other error cases in which
           * you might try again.
           */
          if (++try < MAX_TRY) {
              sleep(2);
              continue;
          }
          (void) fprintf(stderr,"File busy try again later!\n");
          return;
      }
      perror("lockf");
      exit(2);

}
...
```

It should be noted that the *lockf()* example appears to be simpler, but the *fcntl()* example exhibits additional flexibility. Using the *fcntl()* method, it is possible to set the type and start of the lock request simply by setting a few structure variables. *lockf()* merely sets write (exclusive) locks; an additional system call (*lseek()*) is required to specify the start of the lock.

Setting and Removing Record Locks

Locking a record is done the same way as locking a file except for the differing starting point and length of the lock. We now try to solve an example problem of dealing with two records (which may be either in the same file or in different files) that must be updated simultaneously so that other processes get a consistent view of the information they contain. (This type of problem occurs, for example, when updating the inter−record pointers in a doubly linked list.) To do this, answer the following questions:

- What do you want to lock?
- For multiple locks, in what order do you want to lock and unlock the records?
- What do you do if you succeed in getting all the required locks?
- What do you do if you fail to get all the locks?

In managing record locks, you must plan a failure strategy for the case in which you cannot obtain all the required locks. It is because of contention for these records that we have decided to use record locking in the first place. Different programs might:

- wait a certain amount of time, and try again
- end the procedure and warn the user
- let the process sleep until signaled that the lock has been freed
- some combination of the above

Let us now look at our example of inserting an entry into a doubly linked list. For the example, we assume that the record after which the new record is to be inserted has a read lock on it already. The lock on this record must be changed or promoted to a write lock so that the record may be edited.

Promoting a lock (generally from read lock to write lock) is permitted if no other process is holding a read lock in the same section of the file. If processes exist with pending write locks that are waiting for the same section of the file, the lock promotion succeeds and the other (sleeping) locks wait. Promoting (or demoting) a write lock to a read lock carries no restrictions. In either case, the lock is reset with the new lock type. Because the /usr/group *lockf()* function does not have read locks, lock promotion is not applicable to that call. Here is an example of record locking with lock promotion:

```
struct record {
      .
      .             /* data portion of record */
      .
      long prev;    /* index to previous record in the list */
      long next;    /* index to next record in the list */
};

/* Lock promotion using fcntl(2)
```

```
   * When this routine is entered it is assumed that there are
   * read locks on "here" and "next".
   * If write locks on "here" and "next" are obtained:
   * Set a write lock on "this".
   * Return index to "this" record.
   * If any write lock is not obtained:
   * Restore read locks on "here" and "next".
   * Remove all other locks.
   * Return a -1.
   */
long set3lock (this, here, next)
long this, here, next;
{
      struct flock lck;

      lck.l_type = F_WRLCK;    /* setting a write lock */
      lck.l_whence = 0;        /* offset l_start from beginning of file */
      lck.l_start = here;
      lck.l_len = sizeof(struct record);

      /* promote lock on "here" to write lock */
      if (fcntl(fd, F_SETLKW, &lck) < 0) {
          return (-1);
      }
      /* lock "this" with write lock */
      lck.l_start = this;
      if (fcntl(fd, F_SETLKW, &lck) < 0) {
          /* Lock on "this" failed;
           * demote lock on "here" to read lock.
           */
          lck.l_type = F_RDLCK;
          lck.l_start = here;
          (void) fcntl(fd, F_SETLKW, &lck);
          return (-1);
      }
      /* promote lock on "next" to write lock */
      lck.l_start = next;
      if (fcntl(fd, F_SETLKW, &lck) < 0) {
          /* Lock on "next" failed; demote lock on "here" to read lock,... */
          lck.l_type = F_RDLCK;
          lck.l_start = here;
          (void) fcntl(fd, F_SETLK, &lck);
          /* ...and remove lock on "this".  */
          lck.l_type = F_UNLCK;
          lck.l_start = this;
          (void) fcntl(fd, F_SETLK, &lck);
          return (-1) /* cannot set lock, try again or quit */
      }

      return (this);
}
```

The locks on these three records were all set to wait (sleep) if another process was blocking them from being set. This was done with the F_SETLKW command. If the F_SETLK command was used instead, the *fcntl()* system calls would fail if blocked. The program would then have to be changed to handle the blocked condition in each of the error return sections.

Let us now look at a similar example using the *lockf()* function. Since there are no read locks, all (write) locks are referenced generically as locks.

```
/* Lock promotion using lockf(3).
   * When this routine is entered it is assumed that there are
   * no locks on "here" and "next".
   * If locks are obtained:
   * Set a lock on "this".
   * Return index to "this" record.
   * If any lock is not obtained:
   * Remove all other locks.
   * Return a -1.
   */

#include <unistd.h>

long set3lock (this, here, next)
long this, here, next;
{

      /* lock "here" */
      (void) lseek(fd, here, 0);
      if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
          return (-1);
      }
      /* lock "this" */
      (void) lseek(fd, this, 0);
      if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
          /* Lock on "this" failed.  Clear lock on "here". */
          (void) lseek(fd, here, 0);
          (void) lockf(fd, F_ULOCK, sizeof(struct record));
          return (-1);
      }

      /* lock "next" */
      (void) lseek(fd, next, 0);
      if (lockf(fd, F_LOCK, sizeof(struct record)) < 0) {
          /* Lock on "next" failed.  Clear lock on "here", */
          (void) lseek(fd, here, 0);
          (void) lockf(fd, F_ULOCK, sizeof(struct record));
          /* and remove lock on "this". */
          (void) lseek(fd, this, 0);
          (void) lockf(fd, F_ULOCK, sizeof(struct record));
          return (-1); /* cannot set lock, try again or quit */
      }

      return (this);
}
```

Locks are removed in the same manner as they are set; only the lock type is different (F_UNLCK or F_ULOCK). An unlock cannot be blocked by another process and only affects locks that were placed by the unlocking process. The unlock only affects the section of the file defined in the previous example by *lck*. It is possible to unlock or change the type of lock on a subsection of a previously set lock; this may cause an additional lock (two locks for one system call) to be used by the operating system. This occurs if the subsection is from the middle of the previously set lock.

Getting Lock Information

One can determine which processes, if any, are blocking a lock from being set. This can be used as a simple test or as a means to find locks on a file. To find this information, set up a lock as in the previous examples and use the F_GETLK command in the *fcntl()* call. If the lock passed to *fcntl()* would be blocked, the first blocking lock is returned to the process through the structure passed to *fcntl()*. That is, the lock data passed to *fcntl()* is overwritten by blocking lock information. This information includes two pieces of data that have not been discussed yet, *l_pidf* and *l_sysid*, that are only used by F_GETLK. (For systems that do not support a distributed architecture the value in *l_sysid* should be ignored.) These fields uniquely identify the process holding the lock.

If a lock passed to *fcntl()* using the F_GETLK command would not be blocked by another process' lock, then the *l_type* field is changed to F_UNLCK and the remaining fields in the structure are unaffected. Let us use this capability to print all the segments locked by other processes. Note that if several read locks occur over the same segment, only one of these is found.

```
struct flock lck;

/* Find and print "write lock" blocked segments of file. */
(void) printf("sysid pid type start length\n");
lck.l_whence = 0;
lck.l_start = 0L;
lck.l_len = 0L;
do {
      lck.l_type = F_WRLCK;
      (void) fcntl(fd, F_GETLK, &lck);
      if (lck.l_type != F_UNLCK) {
          (void) printf("%5d %5d %c %8d %8d\n",
                        lck.l_sysid,
                        lck.l_pid,
                        (lck.l_type == F_WRLCK) ? 'W' : 'R',
                        lck.l_start,
                        lck.l_len);
          /* if this lock goes to the end of the address
           * space, no need to look further, so break out.
           */
          if (lck.l_len == 0)
              break;
          /* otherwise, look for new lock after the one
           * just found.
           */
          lck.l_start += lck.l_len;
      }
} while (lck.l_type != F_UNLCK);
```

fcntl() with the F_GETLK command always returns correctly (that is, it will not sleep or fail) if the values passed to it as arguments are valid.

The *lockf()* function with the F_TEST command can also be used to test if there is a process blocking a lock. This function does not, however, return the information about where the lock actually is and which process owns the lock. Here is a routine using *lockf()* to test for a lock on a file:

```
/* find a blocked record. */

/* seek to beginning of file */
(void) lseek(fd, 0, 0L);
/* set the size of the test region to zero
   * to test until the end of the file address space.
   */
if (lockf(fd, F_TEST, 0L) < 0) {
      switch (errno) {
      case EACCES:
      case EAGAIN:
          (void) printf("file is locked by another process\n");
          break;
      case EBADF:
```

```
            /* bad argument passed to lockf */
            perror("lockf");
            break;

        default:
            (void) printf("lockf: unknown error <%d>\n", errno);
            break;
        }
}
```

When a process forks, the child receives a copy of the file descriptors that the parent has opened. The parent and child also share a common file pointer for each file. If the parent were to seek to a point in the file, the child's file pointer would also be at that location. This feature has important implications when using record locking. The current value of the file pointer is used as the reference for the offset of the beginning of the lock, as described by *l_start*, when using an *l_whence* value of 1. If both the parent and child process set locks on the same file, possibly a lock will be set using a file pointer that was reset by the other process. This problem appears in the *lockf()* function call as well and is a result of the */usr/group* requirements for record locking. If forking is used in a record locking program, the child process should close and reopen the file if either locking method is used. This results in the creation of a new and separate file pointer that can be manipulated without this problem occurring. Another solution is to use the *fcntl()* system call with an *l_whence* value of 0 or 2. This makes the locking function atomic, so processes sharing file pointers can lock without difficulty.

Deadlock Handling

There is a certain level of deadlock detection/avoidance built into the record locking facility. This deadlock handling provides the same level of protection granted by the */usr/group* standard *lockf()* call. This deadlock detection is only valid for processes that are locking files or records on a single system. Deadlocks can potentially occur only when the system is about to put a record locking system call to sleep. A search is made for constraint loops of processes that would cause the system call to sleep indefinitely. If such a situation is found, the locking system call fails and sets *errno* to the deadlock error number. If a process wishes to avoid the use of the system's deadlock detection it should set its locks using F_GETLK instead of F_GETLKW.

Selecting Advisory or Mandatory Locking

The use of mandatory locking is not recommended for reasons stated in the next section, "Mandatory Locking." Whether or not locks are enforced by the I/O system calls is determined at the time the calls are made, by the state of the permissions on the file (see *chmod*(2)). For locks to be under mandatory enforcement, the file must be a regular file with the set-group-ID bit on and the group execute permission off. If either condition fails, all record locks are advisory. Mandatory enforcement can be assured by code like this:

```
#include <sys/types.h>
#include <sys/stat.h>

int mode;
struct stat buf;
...
if (stat(filename, &buf) < 0) {
      perror("program");
      exit (2);
}
/* get currently set mode */
mode = buf.st_mode;
/* remove group execute permission from mode */
mode &= ~(S_IEXEC>>3);
/* set 'set group id bit' in mode */
mode |= S_ISGID;
```

```
if (chmod(filename, mode) < 0) {
      perror("program");
      exit(2);
}
...
```

Files that are to be record locked should never have any type of execute permission set on them. This is because the operating system does not obey the record locking protocol when executing a file.

The *chmod* command can be easily used to set a file to have mandatory locking. This can be done with the command

```
IRIS% chmod +l filename
```

The *ls* command shows this setting when you ask for the long listing format:

```
IRIS% ls -l filename
-rw---l--- 1 abc other 1048576 Dec 3 11:44 filename
```

Mandatory Locking

Some points to remember about mandatory locking:

- Mandatory locking only protects those portions of a file that are locked. Other portions of the file that are not locked may be accessed according to normal UNIX system file permissions.
- If multiple reads or writes are necessary for an atomic transaction, the process should explicitly lock all necessary pieces before any I/O begins. Thus, advisory locking is sufficient for all programs that perform in this way.
- As stated earlier, arbitrary programs should not have unrestricted access permission to files that are important enough to record lock.
- Advisory locking is more efficient because a record lock check does not have to be performed for every I/O request.

Record Locking Across Multiple Systems

In a UNIX environment, the system on which the locking process resides may be remote from the system on which the file and record locks reside. In this way multiple processes on different systems may put locks upon a single file that resides on one of them, or on yet another system. The record locks for a file reside on the system that maintains the file. It is also important to note that deadlock detection and avoidance is only determined by the record locks being held by and for a single system. Therefore, it is necessary that a process hold record locks on only a single system at any given time for the deadlock mechanism to be effective. If a process needs to maintain locks over several systems, it is suggested that the process avoid the sleep–when–blocked features of *fcntl()* or *lockf()* and that the process maintain its own deadlock detection. If the process uses the sleep–when–blocked feature, then a timeout mechanism should be provided by the process so that it does not hang waiting for a lock to be cleared.

Conclusion

Record locking has been added to UNIX System V to ease the development of small to mid–range database systems. The implementation is compatible with the published standards of */usr/group* to this date. Minor differences in operation may exist; they were made for correctness, or to include things not covered by the published standards.

Chapter 7

Using Real–Time Programming Features

This chapter introduces a number of techniques for achieving real–time performance under IRIX. Note that only multiprocessor machines are really suitable for tasks requiring real–time response, so most of this chapter is geared toward multiprocessor systems, although some of these techniques are useful on single−processor systems as well.

This chapter contains the following sections:

- **"Introduction to Real−Time Programming"** introduces the concepts and techniques of real−time programming under IRIX.
- **"Real−Time Features"** describes the various mechanisms available to help you achieve real−time performance.
- **"Optimal Real−Time Environment"** describes how to use the real−time features together to create the best real−time environment.
- **"Real−Time With Ada"** discusses some special considerations for real−time programming using the Ada language.
- **"Real−Time Latency"** describes the sources of interrupt latency and how to eliminate them.

Introduction to Real−Time Programming

The IRIX operating system supports a powerful set of real−time programming features. You can use these features in combination to accurately time events, use signals as true interrupt routines, control allocation of real memory to the process, and provide for priority scheduling.

In addition, you can use the fully configurable kernel to install custom drivers when wanted. This provides for a range of response time and latency from very fast handling at device interrupt time to high priority dispatch of user processes to handle the event.

Advanced programming features (such as shared process groups and mapped files) provide an environment in which you can construct tightly coupled multiprocess applications, assuring better response for real−time work.

A real−time system provides immediate response to specific external events. Thus, a programmer can schedule particular processes to run within a specified time limit after the occurrence of an event.

Optimal real−time response requires at least two processors: one to handle interrupts and other jobs, and one to service high−priority real−time jobs. A multiprocessor system can provide deterministic real−time response if the unpredictable loads (such as interrupts) are handled on processors other than the processor running the real−time application.

The system scheduler has to do a fair amount of work to switch contexts from one process to another. The scheduling algorithm spends its time among three phases: saving the context of the current process, searching for the highest priority process it can run, and restoring the context of the selected process. Real−time response can be greatly improved if context switches are eliminated. This can be accomplished by creating a shared process group (using the *sproc*(2) system call), in which multiple processes share resources such as address space. One of these processes is dedicated to handle real−time events. You can designate a particular CPU to run only the real−time process; thus, the context of the process is always available in the real−time CPU maps and registers.

Real−Time Features

This set of features refers to the programmer−visible features of the system. An example program (provided later in this chapter) shows how to incorporate these features into your program for optimal real−time response.

The following sections describe features that may be valuable for real−time programming:

- **"Interval Timers"**
- **"Event Timing"**
- **"Reliable and Queued Signals"**
- **"Virtual Memory Control"**
- **"Implicit I/O"**
- **"Asynchronous I/O"**
- **"Scheduling Control"**
- **"Shared Process Groups"**
- **"User Synchronization Primitives"**
- **"Process/Processor Locking"**

- **"Processor Isolation"**
- **"Process Blocking"**
- **"Resource Limits"**
- **"Real−Time Support in the Kernel"**

Interval Timers

In BSD4.2, Berkeley introduced a facility called interval timers, often shortened to just itimers. This facility provides microsecond−resolution for both timers and the time−of−day clock. An interval timer allows the user to specify both an offset from the current time (the delay) and the recurrence time (the interval). The timer does not fire until the delay has passed, and then continues to fire at the end of each interval.

Three timers are provided, each of which delivers a separate signal to the process. The first is the real−time timer, which delivers the standard SIGALRM. The second is a process virtual time timer, which runs only when the process is running in user mode, delivering the signal SIGVTALRM. The third timer is the system virtual time timer, which runs when the process is in either user mode or the kernel is operating on behalf of the user. This timer delivers the signal SIGPROF.

Using the combination of "itimers" and reliable signals (described later), it is possible to implement accurate handling of tasks at regular intervals. Although the resolution of the clock is in microseconds, the actual resolution provided by the kernel is equivalent to the kernel clock frequency, which is 100 Hz on the 4D series, giving 10 millisecond resolution. IRIX fully supports "itimers" as specified in the BSD4.2 programmer’s manual.

Using the ftimer(1) command, it is possible to enable fast itimers, which have a finer resolution. This allows more accurate timing with a slight loss in performance for one of the CPUs in a system, due to the additional overhead of more frequent timer interrupts. The best resolution depends upon the underlying hardware and the setting of the fasthz variable in the kernel. See ftimer(1) and timers(5) man pages for resolution ranges. systune(1M) can be used to query and adjust the value of fasthz.

In addition to interval timers, some applications need to pause for a specified amount of time. Typical UNIX systems include a "sleep" service, which provides this functionality with the granularity of sleep measured in seconds. IRIX includes the sginap call, which provides sleep resolutions down to 10 milliseconds.

In addition (MOVE THIS SECTION?),

- *select*(2) allows a process to sleep awaiting an event associated with a file descriptor, with high resolution.
- POSIX 4 draft 12 *sigtimedwait*(3) supports waiting for a signal event with high resolution.

Event Timing

The generic UNIX operating system offers two types of timing services. The time(2) system call returns the time in seconds since midnight January 1, 1970, in the local time zone. The second type of service is timing between events. Using the times(2) system call, a programmer may get the elapsed time since an arbitrary point in the past (that is, system startup) to the resolution of the kernel clock. By getting the time before and after an operation and then subtracting, the application can calculate elapsed time.

IRIX extends these facilities through support of the BSD4.2 gettimeofday(2) system call, which provides microsecond resolution. This system call returns a structure composed of two elements: the seconds since January 1, 1970 and the incremental microseconds since the last second was counted. The actual resolution of gettimeofday(2) depends on the underlying hardware (see timers(5)), but is never worse than 10 milliseconds.

In addition to event timing via the kernel clock, some computers (Challenge systems and POWER Series systems with the IO3) have a free−running counter that can be mapped into the address space of a user process and used for very fine resolution timing with low overhead. The resolution is 62.5 nanoseconds on the IO3 and 21 nanoseconds on the Challenge system. syssgi(2) can be used to query the existence of this counter, determine its resolution, and obtain an address to be used with the mmap(2) system call in mapping the counter into the user address space.

Reliable and Queued Signals

Reliable signals are necessary because of the poor semantics of the original UNIX signal definition. In the original definition, signals are reset when caught, meaning that the kernel clears any signal handling information set up by the user just before entering the user's signal handler. In most cases, this sets the behavior to the default, which is usually to kill the process. The signal handler is usually responsible for resetting the signal handling information immediately upon entry; however, this leaves a small window in which a signal may be delivered again to a process, causing the (unwanted) default behavior. See the signal(2) manual page in the UNIX Programmer's Manual for a complete description of this behavior.

Such unreliability when dealing with signals is unsuitable for real–time programming. Thus, AT&T added reliable signal handling capabilities to SVR3; these capabilities are described in the sigset(2) man page. Reliable signals are not reset when caught, meaning that user signal handling information is retained until changed by the user. In addition, a method is provided for blocking and unblocking signals in a manner similar to the way hardware interrupts may be blocked or unblocked. This makes it possible to create "critical sections" within a user program that are safe from interference by signal handlers. The signal handler itself is a "critical section"; the kernel automatically blocks that particular signal until the handler returns. IRIX fully supports this reliable signal implementation.

IRIX also supports the full semantics of BSD4.3 signals (signal(3B)), including the signal stack system call sigstack(2) and POSIX signals (sigaction(2)). Both of these signal mechanisms are reliable, like the AT&T mechanism.

All signal mechanisms have support for signals 0–64, except the BSD signals, because a fixed 32–bit signal mask is part of their interface. POSIX 1003.4a Draft 12 reserves signals between SIGRTMIN and SIGRTMAX (49–64) as real–time signals. Within that range, lower signal numbers have higher priority and are delivered first when more than one signal is pending. Signals 0–32 have equal priority for delivery, and are higher than any signals between SIGRTMIN and SIGRTMAX. Signals 33 through 49 are not guaranteed to be available to the user in future releases.

IRIX supports the implementation of queued signals described in POSIX 1003.4a Draft 12. Normally, when a signal is pending, further instances of that same signal are not delivered. With queued signals, multiple instances of the same signal are queued and delivered in FIFO order. Signal queuing behavior can he obtained by setting the SA_SIGINFO flag (see *sigaction*(2), *sigqueue*(3), *signal*(5)). Up to SIGQUEUE_MAX signals can be queued to a process (the default is 64 but can be adjusted via systune(1M).) A process can wait for a signal in two ways: with sigwaitrt(3), if the signal is not immediately available, the process blocks; with sigtimedwait(3), the process blocks for an amount of time specified in an argument to the function call.

The timely delivery of signals is an interesting issue. Typical UNIX systems only look to deliver a signal to a process whenever the process enter or exits the system, or on a kernel timer interrupt (which is usually either 10 mS or 16 mS). There are two problems with this in a multiprocessor real–time environment. First, if the resolution of the interval timer is less than the system clock, and the process does no system calls, the process may not be given the signal until the next system clock tick. The second problem arises when a process can migrate between multiple processors. If the timer fires on a different processor than the one on which the process is running, the process may not see its pending signal until the next scheduling cycle. IRIX solves both these problems—the delivery of SIGALARM signal checks where the target processor is running. If it is on the processor where the timer fired, a scheduling cycle is immediately performed. If the target process is running on another processor, an inter–processor interrupt is dispatched to that process.

Virtual Memory Control

In a paging system, it is always possible that the next reference a program makes, whether it's to text or data, may cause a page fault. Fixing up a page fault can take a long time, destroying any semblance of real–time behavior. To address this problem, IRIX provides the programmer with the ability to lock ranges of pages into real memory, thus avoiding page–fault penalties.

This capability is supplied in a crude way in SVR3; the user can specify that the text, data, or stack segments be locked into core. It may not always be necessary to lock the entire segment; in fact, it may not be possible on a system with limited memory. IRIX provides mpin(2) and munpin(2), which allow the programmer to lock a specified range of addresses into core. When a range of addresses is pinned by the programmer, the kernel allocates memory to all pages in the range and locks them into core before returning to the user program. Conversely, unpinning pages undoes any previous locking. This makes it possible for an application to manage its memory effectively, and ensure that critical text and data (such as signal handlers) are always available.

If an application really needs to lock its entire virtual space, plock(2) can be used. IRIX has enhanced the locking capabilities by extending the System 5.3 plock(2) call to automatically lock shared library sections. In addition, when an entire data section is locked, any growth of that space (via sbrk(2)) causes the pages to automatically be faulted in and locked down. Shared memory segments (see shmop(2)), shared arenas (see usinit(3P)), and mmap(2) files still need to be individually locked using mpin(2).

Implicit I/O

An old concept that is beginning to show up in various UNIX implementations is that of mapped files or implicit I/O. A user instructs the kernel to "map" a portion of a file into his virtual address space at some point. After mapping, the user can access the file simply by reading and writing the mapped memory in a manner identical to all other memory access. Multiple processes can map the same file, thus providing another way to share memory.

Implicit I/O has advantages in terms of access speed and ease of programming. The programmer does not need to make system calls to read or write data, thus speeding up the program. The kernel uses its paging mechanism to manage the mapped data, and thus can deliver data to the user at a higher rate than system calls allow.

IRIX supports mapped files according to the (unimplemented) specification originally intended for BSD4.2, called the mmap(2) interface. This interface allows read or read/write mapping, shared file data, or copy−on−write data (called private mapping). Calls are provided for synchronizing any in−core portions of the file with the actual disk image in both a synchronous and asynchronous manner.

It is also possible to map files into memory that reside on remote systems and are available via NFS. The synchronization calls provided also ensure that changes to a remote file are flushed back to the NFS server.

While real−time performance cannot be guaranteed for such files, large databases may be profitably shared or data collection algorithms supported. Finally, access to certain devices as mapped files is supported if that access is also supported by the driver. Thus, the driver writer supplies a "map" routine, which identifies a range of addresses in physical I/O space to be mapped, and performs any permission checking necessary. The kernel then maps that address range directly to the user’s address space. If access to the VME address space is desired, it is not necessary for a driver to be involved. By mapping the device /dev/mmem into its address space, a user process may directly access VME devices. The range of addresses that may be specified by the user is controlled through the lboot facility, and thus may be adjusted for security or special access conditions.

Asynchronous I/O

With IRIX 5.0, support for asynchronous I/O was added in accordance with the specification in POSIX 1003.4a Draft 12. A user can queue read and write requests to a device (see aio_read(3) and aio_write(3)) and optionally receive a queued signal when the request completes. The read or write function call returns when the request is queued, instead of blocking until the operation is finished. The user can optionally use process priority to establish the order in which queued requests are completed.

The user can query the state of an asynchronous I/O operation using aio_error(3) and aio_return(3), getting, respectively, error codes and return values similar to those obtained from read(2) and write(2) calls. With lio_listio(3), a program can simultaneously queue a number of requests.

Scheduling Control

UNIX implements priority aging for processes. Thus, a process that is CPU bound has its priority lowered gradually as it runs. This ensures that lower priority processes can eventually run and not be starved out of the CPU. This behavior is good in the original environments for which UNIX was designed: interactive users pounding on ASCII terminals. In real−time programming, the programmer often wishes to ensure that a process will run immediately when an event occurs (for example, delivery of a timer signal). Specifying a high priority with the nice(2) system call is only a partial solution; even "niced" jobs age, which can mean a significant delay before a process gets to run.

To alleviate this problem, the notion of non−degrading or fixed priorities has been added to the IRIX kernel. With this facility, the programmer can specify via schedctl(2) a priority for a process that does not decrease over time, thus insuring that it maintains its priority order in the system. This makes it possible to guarantee that certain processes run in a timely manner, and that they can control the CPU as long as necessary to accomplish their tasks. Non−degrading priorities are available in three bands: above normal UNIX priorities, within the same priority range as normal UNIX processes, and below normal UNIX priority ranges. This gives the ability to support time−critical applications, server applications and batch processing, for example. In order for a program to raise its priority above those of normal UNIX processes, its effective user ID must be that of the superuser.

Non−degrading priorities above the normal UNIX band can be dangerous if misused. For instance, consider a process with the highest non−degrading priority in the system. If this process enters an infinite loop, then all other processes are blocked from running, and it becomes necessary to reset the machine to regain control. On a multiprocessor system, however, it is possible to dedicate a CPU to a particular task using non−degrading priorities without deadlocking the system.

In addition to offering non−degrading priorities, IRIX has the capability of deadline scheduling, whereby a process is given a guaranteed opportunity to execute for a specific amount of time within a specific, recurring period. This capability is especially useful for frame−rate scheduling for simulation or continuous media (DAT, CD−ROM). Via schedctl(2), the user can set the length of the recurring period and the time allocated to the process. Even though it is not guaranteed exactly when the process will execute within the period, it is guaranteed that the process will receive its allotment at some time during the period. Non−degrading priorities in the high band (above normal UNIX processes) take precedence over deadline scheduling.

In IRIX, the user can implement processor sets using the pset(1M) command. Each of these processor sets can be assigned to one of seven different queues, which implement various process scheduling semantics. One of the queues is the real−time queue of processes that have priorities in the high real−time band. Therefore, it is possible to have a set of four processors that are only responsible for handling user real−time processes. Other processors will not run real−time processes, nor will processors in the real−time set ever run other process types.

Shared Process Groups

At the lowest level, IRIX shared process groups (see sproc(2)) allow multiple processes to share resources. This means that a set of processes may share the same virtual address space, and thus have free access to shared data without the overhead of copying or expensive protocols. These processes are referred to as a shared process group, or simply a share group. The processes in a share group are scheduled independently and may run at either user or kernel level in parallel. Because each process in a share group is an independently schedulable process with its own context, share group processes are not the same as "lightweight" processes or "threads" implemented in some operating systems. Share groups are supported in all configurations, whether they are single−processor or multiprocessor.

IRIX also adds a novel innovation to this basic scheme: other resources may be selectively shared as well. For instance, file descriptors may be shared, meaning that if one process opens the descriptor, all other processes in the share group may access it as well. Other resources may be shared, such as the current directory or ulimit(2) values.

Using this facility, it is possible to construct very tightly coupled multiprocessor applications with fast response times. For instance, if two processes share file descriptors, one could open a file and suffer the overhead of disk access and resource allocation while the other maintained real−time response for display update. Once the descriptor is opened, the kernel propagates it automatically across all processes in the share group, which is much cheaper than multiple opens. The real−time process would then have immediate access to data through the previously opened descriptor. Mapped files could be used in this instance as an even lower−overhead method for passing data.

User Synchronization Primitives

IRIX provides several extremely powerful shared memory models (usinit(3P), usmalloc(3P)) for programmers to use. To effectively use shared memory in real−time applications, the synchronization primitives used for access to that memory must be very fast. Invoking kernel primitives, such as System V messages or semaphores, is only acceptable in applications where latency isn't an issue.

IRIX provides a set of user−level primitives that allow very low latency synchronization between processes. There are two levels of synchronization primitive offered, spinlocking (usnewlock(3P)) and semaphores (usnewsema(3P)). If the program is running on a POWER or Challenge Series system, then spinlocking automatically takes advantage of the built−in test−and−set hardware. On other 4D systems, a fast software spinlock mechanism is used instead. Semaphores, which can block or unblock processes, use spinlocks for the fundamental control mechanism and only interact with the kernel when putting a process to sleep or unblocking a waiting process. Semaphores may be used in conjunction with event−based applications by using a special pollable semaphore (via poll(2) and usnewpollsema(3P)). This permits applications to be more event driven, where an events could be hardware input, user input, or synchronization messages from other cooperating processes. These synchronization facilities also provide extensive tracing and metering support, making it much easier to debug and tune an application using them. For more information on IRIX IPC, see **"IRIX IPC" in Chapter 5** .

Process/Processor Locking

Using the sysmp(2) system call or the runon(1) or mpadmin(1) command, the programmer or user may control the distribution of processes among the processors in a multiprocessor system. For instance, it is possible to lock a particular process onto a processor, such that it will always run there, even when performing work within the kernel on behalf of that process. Conversely, it is possible to restrict a processor to run only those processes that are explicitly assigned to it. This makes it possible to dedicate one or more processors to particular processes. However, if a process accesses a driver that is locked to a different processor, the process will execute on the other processor while in the driver; conversely, if a processor is restricted, but drivers or system code must run on that processor, the restricted processor will run those tasks as necessary. Consequently, IRIX allows full control over which processor in the system deals with any particular interrupt.

As an example, consider a time−critical environment in which a processor can be dedicated to real−time events. Using the mpadmin(1) command, all but minimal clock handling is moved to a different processor. All noncritical processes are locked off the real−time processor. The time−critical interrupt is vectored to the real−time processor, while all other interrupts go elsewhere. The programmer uses several threads, one of which is locked onto the real−time processor and the others are locked off. The other threads are responsible for nondeterministic system activity, such as reading or writing files or dealing with the network, while the real−time thread simply awaits time−critical events and handles them.

In this example, the context switch latency is effectively reduced to nearly zero, since the processor is never executing kernel code or other user code. The normal UNIX environment is maintained for programming, simplifying the implementation. This normal UNIX environment includes periodic time slicing (though the time slice can be altered). This implies that even if there is only one process locked onto a processor, and the processor runs only that process, there will be periodic clock interrupts. These interrupts will update the accounting statistics for the running process, decrement its time slice, and perform some other time−sharing accounting. IRIX permits this clock interrupt to be turned off (again, using the mpadmin(1) command or the sysmp(2) system call), which eliminates these interrupts and the resulting latency anomalies. This implies that the process running on such a processor is responsible for all scheduling policy on that processor, and that certain UNIX accounting information will be lost. The clock interrupt can only be turned off on an isolated processor.

Processor Isolation

Nominally, when IRIX is running in a multiprocessor configuration, certain system services require synchronization of all processors in the complex. This is done for two main reasons: to synchronize the instruction caches and to synchronize virtual−to−physical translations (called the TLB). In IRIX 3.3 and earlier releases, this synchronization was accomplished by sending an interrupt to all processors to instruct them of what action was required. Although neither of these actions takes a long time, the fact that a process running on one processor can cause system services to be run on another processor adds to the worst−case dispatch latency on all processors. To solve this issue, IRIX 4.0 and later releases provide the ability to "isolate" a processor from the complex. This can be used in a couple of different ways.

First, when a processor is isolated, the normal IRIX clock processing may be turned off. This implies that

First, when a processor is isolated, the normal IRIX clock processing may be turned off. This implies that normal IRIX time slicing will no longer preempt the running process. It also means that a device driver may stay and poll its device for as long as it wishes, and there will be no effect on the other processors.

Second, processes that run on an isolated processor have control over when, if ever, the synchronizing events take place. If the process never requests any system services, then there is never a need to synchronize. If a processor is isolated, no devices are configured onto that processor, the clock service is disabled, and the application process is entirely locked into main memory, then a user can accurately determine how long a particular compute−bound computation will take.

Process Blocking

IRIX provides several new system calls for managing the run state of a process. A process may block another (after appropriate security checks) or itself, and may unblock any process. This blocking operation is implemented via a counter in the kernel, and thus is free of race conditions. The blocking operations have been tuned for high performance.

Such primitives find their use in many areas, such as user−level semaphore support. Real−time modeling and simulation programs can use such calls to implement preemptive schedulers at the user level. For instance, an operating system environment can be simulated using a shared process group (sharing the virtual memory image) and blocking calls for process control. See *blockproc*(2) for more information.

Resource Limits

IRIX implements the BSD4.3 getrlimit(2) and setrlimit(2) system calls so you can control the resource usage of a process, such as CPU time, resident set size, and disk usage. The most useful of these is resident set size. By controlling the resident set size of various processes, the real−time application developer can ensure that background or low−priority processing does not impact memory usage for infrequently called but high−priority tasks.

Default and hard resource limits are set in the kernel (see stune(1M)). A non−superuser process can successfully adjust its limits up to the hard limit.

Real−Time Support in the Kernel

This set of features is not visible to the programmer but is critical in any real−time application.

The system call and trap handling paths in the kernel have been optimized to minimize overhead. Part of this reduction involves hand coding much of the initial state handling and exception processing in assembly language. Such coding is extremely efficient due to the RISC nature of the MIPS microprocessors, often allowing handling at the full rated clock speed of the processor. In the high−level parts of these handlers, exception conditions, which are normally tested for in a sequential manner in most UNIX implementations, are combined into a single bit−vector that can be tested in a single instruction. This is an advantage, as exception conditions (such as signals or process preemptions) occur rarely for most processes.

The kernel also implements delayed binding for access to the floating−point coprocessor and graphics head. This allows active floating−point registers to stay in place during other processing, unless the floating−point unit is needed for another process (the kernel does not use floating point). In addition, the cost of saving and restoring the state of these units is eliminated for interrupt and trap handling.

In standard UNIX implementations, the scheduler scans the entire run queue when searching for the next job to run. Processes are always added to this queue at the back, so there is no priority ordering to the queue. Normal UNIX scheduling is O(n), where n is the number of processes on the run queue. The IRIX kernel sorts processes on the run queue when adding them (typical UNIX round−robin scheduling is maintained). This insertion sort takes O(n/2) time and is only performed when placing processes on the queue. When dispatching processes from the queue, the first runnable process is taken from the queue, thus eliminating back−end searching.

With IRIX 5.0, the scheduler was redesigned to have seven different queues that utilize different scheduling semantics: the kernel, user real−time, deadline, gang, time−share, gang batch, and batch queues. The kernel and user real−time queues utilize the scheduling described in the preceding paragraph. The deadline queue contains processes that have requested deadline scheduling (see paragraph. The deadline queue contains processes that have requested deadline scheduling (see schedctl(2)). The real−time and deadline queues would be the ones used most often by real−time processes.

Share group processes that have requested gang scheduling (see schedctl(2)) are scheduled on the gang queue if their priority is better than 128; otherwise, they go to the batch gang queue. Processes on the time share queue of a multiprocessor are selected based upon their priority and their cache affinity, a measure of how long the process has spent filling the processor's cache with valid data. Processes with a high cache affinity are preferentially selected ahead of higher priority processes. The hope here is that overall system throughput will increase due to more efficient overall use of the cache. The batch queue uses time−sharing queue semantics for processes with priorities worse than 127. (A list of the queues, their scheduling types, and priority ranges can be obtained via the pset(1M) command.)

By default, all processors can run all processes on all queues, but with the pset(1M) command, the superuser can assign groups of processors (processor sets) to service the various queues. pset(1M) can also be used to assign various disciplines to a processor set, whereby the set's access to various resources (especially graphics) can be controlled. Finally, pset(1) and schedctl(2) can be used to assign processes to various processor sets.

Probably the largest enhancement to IRIX's real−time capabilities has been to make the kernel preemptable. This means that the time in which the kernel ignores a high−priority runnable process because of critical sections is now bounded.

IRIX also has the capability of directing interrupts from the VMEbus to selected processors, to prevent a high−priority real−time process from being interrupted by unwanted signals. This is customizable via the IPL directives in the kernel. By default, all VME interrupts are handled by processor 0, but the user can assign specified VME interrupt levels to specified processors, thereby dedicating a processor to handle one high−priority VME interrupt.

IRIX handles networking in either of two ways: interrupts from the ethernet hardware can be redirected via the IPL mechanism, or the rtnetd(1M) daemon can be run to provide preemptable networking behavior.

All critical sections in which the kernel or Silicon Graphics supplied drivers hold off interrupts have been examined, and their length minimized.

Optimal Real−Time Environment

This section explains how to use the IRIX real−time features to set up optimal real−time response.

Establishing Scheduling Control

Use the *schedctl(2)* command to assign a high, non−degrading priority to a process to prevent interference by other, less important processes.

Establishing Multiprocessor Control

Multiprocessing eliminates unwanted interrupts and lower−priority processes from competing with the real−time process for CPU cycles.

IRIX supports the *sysmp*(2) commands MP_RESTRICT and MP_MUSTRUN (see *sysmp*(2) for details on these commands). These system calls let you dedicate a processor on which to run the real−time program. These calls also let you restrict a particular processor to run only certain designated processes. You must be the superuser to use these commands. The MP_MUSTRUN command assigns the calling process to run only on the processor specified, except as required for communications with hardware devices. The MP_RESTRICT command restricts the processor specified from running any process except those assigned to it via the MP_MUSTRUN command or the *runon*(1) command, or because of hardware necessity.

Under IRIX, each processor in the system handles its own clock interrupt. The overhead of handling the clock interrupt is quite low for all the processors except for one special processor, the clock processor. It is selected by the MP_CLOCK (to specify a processor to handle the system clock) *sysmp* command. The real−time processor should not be the clock processor. Note that the program sample that follows uses various *sysmp* commands.

For extremely low latency situations, *sysmp(2)* supplies the commands MP_ISOLATE and MP_NONPREEMPTIVE to isolate a processor by eliminating TLB and instruction cache synchronization, and turning off the scheduling clock. Processes isolated in this way cannot receive signals or be share group (sproc(2)) processes. In addition, they should only run user code (no system calls); otherwise, they are subject to the latencies involved in any kernel access.

Locking Interrupts

On a single processor system, the time required for all interrupts to be serviced until no more interrupts are pending and process scheduling can proceed constitutes an unpredictable latency. Fortunately, on the multiprocessor systems, VME interrupt levels can be individually locked on to any processor in the system. For real–time application, move all the unwanted VME interrupt levels away from the real–time processor. To lock a particular VME interrupt level to a processor, edit the */usr/sysgen/system* file and run *lboot*(1M) for the changes to take effect. The format is:

IPL: *level cpu#*

where *level* is the priority level (0 – 7, with 7 being the highest), and *cpu#* is the number of the CPU on which you want the VME interrupts of that level to occur.

For example:

```
IPL: 4 1
```

designates VME interrupt priority level 4 on CPU number 1.

After editing the *system* file, you must run *lboot*(1M) to reconfigure the system for the changes to take effect. See *lboot*(1M) and *system*(4) for details.

On Challenge and Onyx systems, you must also use the NOINTR directive (see *system*(4)) to prevent other interrupts from being sent to the real–time processor.

Reducing Other Latencies

Use the *plock(2)* and/or *mpin(2)* system calls to lock the important parts of your program in memory to prevent additional latencies due to page faults.

A Real–Time Example

The following example shows how to set up a user program for real–time application and acquire system resources for optimal real–time processing. This program includes examples of the use of fine–grained memory locking, process blocking, shared process group, and reliable signals.

In this program, the user process is broken up into two threads executing on different processors. The real–time thread runs on a dedicated real–time processor, while the slave thread runs on a different processor.

The real–time thread execution:

- sets up the itimer to simulate a real–time device that keeps interrupting the real–time thread by periodically sending it a SIGALARM.
- does some CPU–intensive calculation, trying not to sleep to avoid any context switching when it gets the itimer signal.
- on receipt of the itimer signal, wakes up the slave thread for additional I/O related processing.

```
# include <sys/types.h>
# include <sys/time.h>
# include <sys/schedctl.h>
# include <sys/sysmp.h>
# include <sys/pda.h>
# include <signal.h>
# include <setjmp.h>
# include <stdio.h>
# include <sys/prctl.h>
```

```
int realtime_cpu = -1;
int master_cpu = 0; /*default to CPU 0 to
   * handle system clock*/
extern int errno;
struct timeval lasttime;

int  npri;
struct { /* structure we communicate through */
 int ppid; /* parent process ID */
 int cpid; /* child process ID */
 unsigned count;   /* counter, bumped by parent */
 unsigned nintr;   /* counter of itimer signals */
} comarea;
main(argc, argv)
   int  argc;
   char  *argv[];
{
   extern int  optind;
   extern char  *optarg;
   int  c;
   int  err;
   int  i;
   int  nprocs;
   int  catcher();
   register struct pda_stat *pstatus, *p;
   struct itimerval itv;
   int asyncslave();
     /*
  * Parse arguments.
  */
 while ((c = getopt(argc, argv, "r:f:")) != EOF) {
   switch (c) {
  case 'f':
     /*
      * Set a non-degrading or fixed priority
      * at the given value.
      */
       if ((npri = strtol(optarg, (char **) 0, 0)) <= 0) {
      err++;
      break;
  }
  /*
   * Figure out which band it is in.
   * Smaller values mean higher priorities.
   * These codes are here for illustration.
   */
       if (npri >= NDPHIMAX && npri <= NDPHIMIN) {
   /*
    * High priority (higher than all other
    * processes).
     */
      }
       else if (npri >= NDPNORMMAX && npri <= NDPNORMMIN) {
    /*
     * Non-degrading in the normal UNIX priority
     * bands. This should give "constant response."
     */
      }
```

```
        else if (npri >= NDPLOMAX && npri <= NDPLOMIN) {
     /*
      * Lower than all other processes. Suitable
      * for batch work, etc.
      */
       }
         else
      err++;
    break;
  case 'r':
        if ((realtime_cpu = strtol(optarg,
   (char **) 0, 0)) < 0)
     err++;
  break;
  case '?':
     err++;
  break;
  }
}
if (err) {
 fprintf(stderr, "usage: %s [-rcpu] [-fpri]\\n",
   argv[0]);
 exit(1);
}
/*
 * Start the slave thread first.
 */
comarea.ppid = getpid();
comarea.cpid = sproc(asyncslave, PR_SALL, 0);
/*
 * Pin the signal handlers down to improve signal
 * handling performance. We pass in a stack address
 * so we know which part of the stack to lock.
 */
pinmem((char *) &itv);
/*
 * Set the priority if requested. If in the high
 * priority band, then you should always get control.
 * If in the middle band, depends on who else is
 * running. If in the lower band, you only
 * get cycles if nothing else is running. Note
 * that setting a non-degrading priority requires
 * superuser privileges.
 */
if (npri != 0)
 schedctl(NDPRI, 0, npri);
/*
 * To dedicate a processor to handle interrupts
 * from a real-time device, use lboot to reconfig
 * the kernel and reboot the system (see system(4),
 * lboot(1M)). To run your process only on the
 * dedicated real-time processor,
 * then use the following system call.
 */
if (realtime_cpu >= 0)
 if (sysmp(MP_MUSTRUN, realtime_cpu) < 0) {
   perror("Failed MP_MUSTRUN.
          Resource not available!");
 }
```

```
   /*
  * To obtain maximum real-time processing power
  * out of the selected processor, use the following
  * system call to kick all other processes
  * from the real-time processor.
  */
 if (realtime_cpu >= 0 && sysmp(MP_RESTRICT,
   realtime_cpu) < 0) {
 perror("Failed MP_RESTRICT.
   Resource not available!");
}
/*
 * Move the handling of the system clock
 * to another CPU, if it is currently
 * handled by the real-time CPU. This will
 * increase the bandwidth of the real-time CPU.
 */
nprocs = sysmp(MP_NPROCS);
if (nprocs < 0) {
  perror("Failed MP_NPROCS. Fatal system error!");
  quit(-1);
}
pstatus = (struct pda_stat *)
  calloc(nprocs, sizeof(struct pda_stat));
if (sysmp(MP_STAT, pstatus) < 0) {
  perror("Failed MP_STAT. Fatal system error!");
  quit(-1);
}
 /*
  * Figure out which processor is currently
  * handling the system clock. If that processor
  * happens to be the real-time CPU,then move
  * that functionality to another processor
  * (the master processor in this example).
  */
for (i = 0, p = pstatus; i < nprocs; i++) {
if ((p->p_flags & PDAF_CLOCK) &&
   (i == realtime_cpu)) {
         if (sysmp(MP_CLOCK, master_cpu) < 0) {
  perror("Failed MP_CLOCK.
   Fatal system error!");
  quit(-1);
   }
   break;
   }
    p++;
}
/*
 * Set up signal handling. Initialize the alarm
 * signal to be "held", which means that
 * timer pops will be ignored until we
 * "release" the signal.
 */
sigset(SIGALRM, catcher);
sighold(SIGALRM);
/*
 * Set up timer. The interval is the time between
 * each successive timer pop. The value is the
```

```
 * initial value of the timer, which can be anything.
 * We will set the timer to start 10ms from now, and
 * keep interrupting every 10ms thereafter.
 */
itv.it_interval.tv_sec = 0;
itv.it_interval.tv_usec = 10000;
itv.it_value = itv.it_interval;
setitimer(ITIMER_REAL, &itv, (struct itimerval *) 0);
/*
 * Get the starting time, and release
 * the interrupt signal.
 */
gettimeofday(&lasttime, 0);
sigrelse(SIGALRM);
   while (1) {
    /* simulate CPU intensive processing */
  comarea.count++;
   }
    /*NOTREACHED*/
}
/*
 * Slave thread
 * block waiting for parent to wake child up upon
 * receipt of the itimer signal.
 */
asyncslave()
{
    static struct timeval last;
    static int lastintr;
    struct timeval newtime;
    double  rt;
    int cdone();
       /*
      * Put us at the highest possible real-time
      * priority so we respond to events very quickly.
      */
    schedctl(NDPRI, 0, npri);
      last = lasttime;
      /* initialize semaphore count */
    setblockproccnt(comarea.cpid,0)
      /* catch INT signal to let parent know */
    sigset(SIGINT, cdone);
      for (;;) {
 /*
  * parent will wake us up
  */
 blockproc(comarea.cpid);
 /*
   * Get the time that we finished.
  */
 gettimeofday(&newtime, 0);
 /*
  * Turn the net time difference into a
  * floating-point number so we can
  * reasonably deal with it.
  */
 rt = newtime.tv_sec - last.tv_sec;
 rt += (double)(newtime.tv_usec - last.tv_usec) / 1000000;
 /*
```

```
  * Tell the user what the result was.
  */
 printf("Current rate of interrupts/sec = %.2f\\n"
   (comarea.nintr - lastintr)/rt);
 last = newtime;
 lastintr = comarea.nintr;
    }
    /*NOTREACHED*/
}
/*
** parent signal handler
*/
catcher()
{
    comarea.nintr++;
    /* Wake up slave thread to output the results
     * every 100 signals.
     */
    if ((comarea.nintr % 100) == 0)
 unblockproc(comarea.cpid);
}
/*
** code to lock critical memory in core; lock
** stack, signal handler, and the communication area
*/
pinmem(sbot)
   char  *sbot;
{
       /*
      * Pin down various pieces of critical
      * memory. Start with the stack.
      * Allow about 2K for the stack (signal
      * handlers run on the same stack).
      */
    /* stack grows downward */
   mpin(sbot - (2*1024), 2*1024);
      /* pin data areas */
   mpin(&comarea, sizeof(comarea));
      /* pin text areas */
    mpin((char *) catcher, (int) pinmem - (int) catcher);
}
quit(code)
{
    kill(comarea.cpid, SIGINT);
    exit(code);
}
cdone()
{
    kill(comarea.ppid, SIGINT);
    exit(0);
}
```

Real−Time With Ada

Many real−time programs require Ada as their language of development. Without MP Ada (MPXM−6.2), this is accomplished by making *pragma interface* calls to the C function calls (*sysmp(2)*, *schedctl(2)*, and so on). In this way, you can develop a program with a guaranteed response in Ada.

One problem is the use of itimers with standard Ada. Because the Ada execution environment uses the *SIGALRM* signal to handle task scheduling, and itimers to execute *delay* statements, these uses conflict with itimers that also send a *SIGALRM*. Another problem is that all Ada tasks execute within the process space of a single process; therefore, adjusting the priority of one Ada task changes the priority of all the Ada tasks in the Ada program.

The MP Ada product allows Ada tasks to run on separate share group processes created via *sproc(2)*. The paradigm selected for this capability is the POSIX threads detailed in the POSIX 1003.4a Draft 6 document.

This *sproc(*2) creation can be hidden from the user; she can determine the number of default "sprocs" to create via the *VADS_SPROC_COUNT* variable, and the Ada execution environment would assign tasks to any available sprocs for true symmetric multiprocessing.

To allow greater control necessary for real−time programs, the user can individually create sprocs via the *sproc_create()* call (from package *os_sproc*), assigning the sproc a set of attributes. The attributes assignable include the CPU to execute on (*sproc_attr_setcpu()*), whether the sproc should be locked in memory (*sproc_attr_setresident()*), and the non−degrading priority (*sproc_attr_setprio()*). In addition, the *sproc_create()* routine can be passed the name of another routine, and an argument to that routine, that can be used to configure other real−time parameters (such as processor restriction, location of clock processor, and so on). Interfaces to the *sysmp(2)* command are now available as a part of MP Ada and do not need to be written by the user (see package *os_prctl*).

Another important attribute of the sproc is a resource vector, which the Ada execution environment uses to assign specific tasks to the sproc. The *pragma task_attributes* assigns a particular resource vector to a specific task (done in the task specification), and then the Ada execution environment provides the match of sprocs and tasks having the same resource vectors.

By default, Ada execution environment time slicing is turned off; therefore, as long as the user avoids the use of *delay* statements, itimers will work with MP Ada. An interface to *sginap(2)* could be used in lieu of *delay*s.

Since share group processes do not work on isolated processors, Ada programmers cannot use the *sproc_create()* routines when creating a process to run on an isolated processor. Instead, they should use *fork()*/*exec()* to create the process and interfaces to *schedctl(2)* and *plock(2)* to give it a high, non−degrading priority and lock it in memory. Since the interfaces for *sysmp(2)* are already written, they can use these for other real−time needs.

Real−Time Latency

Real−time processes often sit dormant waiting for some real−life event to occur. The real−life event is typically signaled by the occurrence of an interrupt that is generated by a device connected to the computer. When this interrupt occurs, a device driver interrupt handler is dispatched to service the interrupt. The interrupt handler's task often is to wake up a dormant real−time process so that it can respond to the real−life event. The following sections describe all the components of latency that occur during this chain of events under IRIX.

Total interrupt response time is the interval of time between the occurrence of an interrupt and the start of execution of the associated user process. Total interrupt response time is the sum of the interrupt latency, the interrupt service time, and the kernel dispatch latency.

On a typical UNIX system, no maximum can be given for these latencies, since they depend on the amount of system activities generated by local processing and external events. Under IRIX, it is possible to take actions that place bounds on these latencies. The following sections describe the sources of latency and the actions required to minimize their effects. Worst−case results were measured after the recommended approaches were taken.

Interrupt Latency

Interrupt latency is the elapsed time between the occurrence of an interrupt and the start of execution of the appropriate interrupt handler.

Sources of Interrupt Latency

There are seven interrupt levels, each of which executes at a higher level than any user process and most kernel routines. A new interrupt is serviced immediately unless:

- An equal or higher−priority interrupt handler is currently executing.
- Interrupts at that level are disabled during kernel execution.

These two sources of latency are discussed in the following paragraphs.

While an interrupt handler is executing, it prevents another interrupt at an equal or lower level from being serviced. Furthermore, all pending interrupts that are equal to or higher than the priority of a new interrupt must complete execution before the new interrupt is serviced.

Certain critical sections of code in the kernel need exclusive access to shared resources. In order to guarantee exclusive access, the kernel acquires a spinlock (only on the POWER Series) and raises the processor's interrupt level. New interrupts that are below the processor's interrupt level will not be serviced until the critical section of code is complete and the processor's interrupt level is lowered.

The seven interrupt levels and the recommended approach to minimizing their effects on the latency of a real−time interrupt (presumed to be VME) are summarized in **Table 7−1** . A discussion of the approach to minimization follows.

Table 7−1 POWER Series Interrupt Levels

Interrupt Level	Source	How To Minimize	Facility
1	Software— kernel/ driver timeout	Redirect non−RT processes	sysmp(2), runon(1) mpadmin
2	Software— network activity	Redirect network processes	sysmp(2), runon(1) mpadmin
3	VME interrupts & SCSI interrupts & Ethernet interrupts	Redirect non−RT interrupts	lboot(1M), system(4)
4	RT processors' local DUARTS	Disable getty on the real−time processor	/etc/inittab
4	Graphics interrupts	Redirect graphics interrupts	sysmp(2), runon(1), mpadmin(1)
4	Inter−processor interrupts	Isolate RT processor and allocate memory early	sysmp(2), mpadmin(1)
5	Scheduler clock	Isolate RT processor and disable scheduler clock	sysmp(2), mpadmin(1)
6	Floating−point exception	Eliminate programming errors	
6	Profiling clock	Disable profiling	prfstat(1)
7	Bus error	(Indicates hardware problem)	

Reducing Interrupt Latency

To prevent the latency of a particular real−time interrupt from being affected by other interrupts in the system, those other interrupts should be prevented from occurring on the same processor as the interrupt of interest. The techniques for accomplishing this fall into four categories:

1. Disabling unused interrupt sources.
2. Redirecting non−real−time interrupts.
3. Isolating a real−time processor to exclude inter−processor interrupts caused by memory management activities on other processors.

4. Avoiding activities on a real−time processor that subject it to memory−management−related interrupts at unpredictable times.

To prevent the latency that results from executing kernel code (with interrupts disabled), the approach is to stay out of the kernel at times that are critical to the real−time application. This requires:

1. Ensuring that non−real−time processes (whose entries into the kernel are assumed to be unpredictable) are prevented from running on real−time processors.
2. Avoiding system calls in real−time processes during periods when minimum interrupt latency is required.

All of these techniques are discussed in the upcoming sections.

Reducing Interrupt Latency Due to Other Interrupts

Disabling Unused Interrupt Sources

Many devices in the system that can generate interrupts are not needed in most real−time applications. These devices can be disabled to ensure that they do not generate interrupts.

The local DUARTS and the profiler clock certainly fall into this category. The DUARTS are used for RS−232 serial communication, and the profiler clock is used only for tuning the IRIX kernel.

Another source of interrupt latency is the standard UNIX scheduling clock that interrupts at priority level 5. The clock interrupt is used for implementing time slicing and system accounting and to provide the kernel and device drivers with a timeout facility. In the case of a real−time process being run at very high priority to ensure that it does not get preempted, clock processing does not serve any useful purpose. On the multiprocessor POWER Series system, the clock interrupt can be turned off on any isolated processor using sysmp(2).

One important side effect of turning off the scheduling clock interrupt on a processor is that a process can then run until it completes or until a higher priority process becomes ready to run on that processor. Higher−priority processes continue to preempt lower priority processes after the scheduling clock is disabled, but processes of equal priority do not preempt one another.

Another side effect of disabling the scheduling clock is that the kernel timeout facility is no longer available to drivers running on that processor.

Redirecting Non−Real−Time Interrupts

On the multiprocessor POWER Series system, any VME interrupt level can be routed exclusively to any processor using lboot(1M) and system(4). The same is true for local SCSI and Ethernet interrupts.

On a single processor system, disk interrupts can be avoided by making use of the traditional UNIX buffer cache. Non−synchronous UNIX disk writes usually don’t immediately cause a physical write to the disk, but instead the data stays in the kernel’s buffer cache until one of the following things occur: 1) the system needs to recycle a buffer that contains data to be written to the disk or 2) the system needs to push out some delayed write data for increased disk consistency. A user process can dictate when the flushing of the delayed write buffers occurs (using syssgi(2)), thus avoiding disk interrupts during a time−critical section.

Also on a single processor, a real−time application that does not need network services during a time−critical section can momentarily turn off the network interface (using ifconfig(1)) to avoid any network−related interrupts.

Isolating a Processor

The POWER Series’ shared memory architecture enables processors to share physical memory. This requires consistency in the virtual−to−physical mapping of shared memory pages by all processors that participate in the sharing.

Maintaining instruction cache and TLB consistency among all processors in the system requires the kernel to periodically broadcast inter−processor interrupts to request synchronization. The frequency of these interrupts is proportional to the system load.

In describing the approach to minimizing the impact on latency, the instruction cache and TLB cases are considered separately.

Instruction Cache Flushes

Under IRIX 4.0, whenever the kernel allocates physical memory to hold a user text page, it determines if the physical page is from the list of recycled (previously allocated) memory pages. If so, a system−wide instruction cache flush is required to ensure that as the user process migrates among processors, it never accesses stale data from the local instruction cache.

POWER Lock™ Processor Isolation, a feature of IRIX 4.0, provides a process with a measure of protection from inter−processor interrupts. Processor isolation is established using either the sysmp(2) system call or the mpadmin(1) shell command.

Isolating a processor excludes that processor from the system−wide cache flushes, unless it is the processor that is initiating the memory allocation. Therefore, by allocating all text memory resources before entering the time−critical section of a real−time process, that process can be fully exempt from synchronization cache flushes during its time−critical section. This can be accomplished either by linking the real−time program as a preloaded program (versus a loaded−on−demand program, see ld(1)) or by using plock(2) to lock all text pages in core immediately.

TLB Flushes

The UNIX kernel allocates virtual pages to facilitate mapping in of user process physical pages during long DMA transfers, since typically the user process context is switched out during these long waits. This virtual−physical mapping established by the kernel tends to be short−lived, since the mapping is meaningful only during the DMA transfer. Once the transfer is complete, the kernel immediately frees its allocated virtual pages into a recycled list of virtual pages. As this process continues, the list of free and untouched virtual pages is depleted. The kernel must allocate virtual memory from the recycled list.

Before the kernel can allocate pages from the recycled list, it requires a system−wide TLB flush to ensure that the TLBs in different processors do not hold conflicting translations of the same virtual page.

To minimize TLB flushes, IRIX 4.0 takes the extra step of keeping track of each kernel page fault. When the time comes to allocate from the recycled list of virtual pages, only the processors that previously used pages that are now on the recycled list are required to flush their TLB.

There are two approaches to minimizing the latency introduced by TLB flushes. The first is to avoid freeing any pages on a real−time processor, either in user code or in a device driver. Instead, memory resources should be allocated once and recycled within the user program or device driver.

The other approach is to allow exposure to TLB flushes only at noncritical times. Processor isolation prevents servicing of an inter−processor interrupt while the isolated processor is executing user code. Instead, servicing of an inter−processor interrupt is delayed until the processor enters the kernel. The kernel is entered when the process running on that processor makes a system call or when any interrupt occurs on that processor.

At present, no shared−resource processes (sproc(2)) are allowed to run on an isolated processor; all other IRIX services are still available.

Reducing Latency Due to Kernel Code Execution

Redirecting Non−Real−Time Processes

POWER Series systems utilize a fully symmetric multiprocessor architecture and normally allow processes to migrate freely among processors (though there is affinity for the processor on which a process was most recently running). This results in optimal load−balancing and the most efficient use of processors. However, this is not optimal in a real−time environment because some processes delay servicing of an interrupt directed to the processor on which they are running. There are three causes of this interrupt latency.

First, there are software interrupts that are self−posted by various interrupt handler routines. In general, if some portions of an interrupt handler are lengthy and are not important enough to be executing at a high interrupt level, the interrupt handler posts a software interrupt to continue the processing at a lower interrupt level before returning the processor to the user program. The networking interrupt handler and the scheduling clock handler are the two handlers that post software interrupts routinely.

Second, processors that pass data to the graphics pipeline (graphics systems only) are subject to receiving a graphics FIFO overflow interrupt that is posted to the processor that was sending data to the graphics pipeline at the time it became full.

Third, during a system call, certain critical sections of the kernel must be protected from being preempted by an interrupt handler to avoid contention for the same resource. This means that if a real−time event interrupts the processor when the processor is executing one of these critical sections, it must wait.

If the activities they are associated with are not essential to the real−time process, these sources of latency can be completely avoided. By dedicating one or more processors to real−time processing and redirecting all non−real−time processes to execute on the other processor(s), the unpredictable latencies introduced by unrelated activity can be eliminated. IRIX provides facilities (sysmp(2), mpadmin(1), runon(1)) to control the distribution of processes among the processors in a multiprocessor system.

Controlling System Calls on Real−Time Processors

The latency increase introduced by system calls made by real−time processes is similar to the third source of latency described in the previous section. The difference is that the real−time user has control over when system calls are made in her own process. This presents the possibility of making system calls at times when increased interrupt latency is not a problem.

The maximum potential latency introduced under any circumstances by making a system call has been measured to be 650 uS.

Worst−Case Interrupt Latency

On an isolated processor that is executing a process in user mode, the maximum time required to save the isolated processor's status and dispatch the interrupt handler has been measured to be 51 uS (from interrupt pending to the first line of C code in the driver interrupt handler, with no other interrupt sources).

Interrupt Service Time

The real−time interrupt handler typically wakes up a sleeping process, clears the interrupt, and returns. The time required to perform these steps is called the interrupt service time. The interrupt service time is application dependent.

Kernel Dispatch Latency

Once the interrupt handler wakes up a real−time process (makes it ready to run), the kernel must exit the interrupt handler, compare the priorities of the just−awakened process and the currently running process, then dispatch the winner. If the new process has the highest priority, a context switch is required. The interval from the time that the new process is ready to run until the execution of one of the user processes actually begins is called the kernel dispatch latency.

Sources of Kernel Dispatch Latency

The factors that can affect kernel dispatch latency and the recommended approach to minimizing them are summarized in **Table 7−2** .

Table 7−2 Kernel Dispatch Latency Sources

Source	How To Minimize	Facility
Process text and/or data not in memory	Allocate memory early and lock into core	mpin(2), plock(2)
Need to grow stack to handle a signal	Specify pre−allocated alternate stack	sigstack(2)
A higher priority process is executing	Assign RT process higher, non−degrading priority	schedctl(2)
Lower priority process is	Redirect process to another	sysmp(2), runon(1),

executing a system call	processor	mpadmin(1)

Worst–Case Kernel Dispatch Latency

The time required to exit the interrupt handler is the same as the time required to enter it; that is, 51 uS. The additional time required to select the highest–priority process and dispatch it, including a context switch, has been measured to be less than 87 uS.

Worst–Case Total Interrupt Response

After following the guidelines outlined in this report for minimizing latency, the worst–case total interrupt response was measured to be as follows, for POWER, Challenge, and Onyx Series computers:

Interrupt latency: 51 uS
Interrupt service: (application specific)
Kernel dispatch: 138 uS
total: 189 uS + interrupt service time

POWER Series measurements were made on a 4D/320 (2 processor, 33 MHz) system. Challenge measurements were made on a four–processor Challenge system. One processor was isolated with no other non–real–time interrupt sources pending. A user process was executing at user level with the real–time process sleeping at high priority. The scheduling clock was turned off on the isolated processor. On the remaining processor, various disk–, network–, and graphics–related processes were executing. The interrupt service routine awakened the real–time process.

Chapter 8

Working with Fonts

This chapter describes how to work with fonts on Silicon Graphics computers. It begins with an introduction to fonts and digital typography. Then it explains which fonts are available and how to install additional fonts. It also covers how to download outline fonts in the Type 1 format to a PostScript printer.

This chapter contains the following sections:

- **"Font Basics"**
- **"Using Fonts with the X Window System"**
- **"Installing and Adding Font and Font Metric Files"**
- **"Downloading a Type 1 Font to a PostScript Printer"**

Font Basics

Fonts are collections of characters. A font contains the information about the shape, size, and position of each character in a character set. That information is needed by those programs which process characters, such as editing, word–processing, desktop–publishing, multimedia, titling, and pre–press application programs. Almost all software components in a computer system use fonts to display messages, prompts, titles, etc.

Computers from Silicon Graphics are called digital computers, because discrete voltage levels are used to represent the binary digits 0 and 1. Binary digits are used to represent all other information stored in a digital computer, including fonts. Digital typography deals with the style, arrangement, and appearance of typeset matter in digital systems. If you want to use font and font metric files to correctly typeset text on a digital computer, you need to know some basic things about digital typography. This section contains a brief introduction to fonts and digital typography. You may want to read a book on typography for more in–depth information.

Introduction to Digital Typography

Typography is the art and technique of working with type. In traditional typography, the term type refers to a piece of wood or metal with a raised image of a character or characters on its upper face. Such pieces of wood or metal are assembled into lines and pages which are printed by a letterpress process.

What typographers do with type is called typesetting or composition. Type can also refer to the images produced by using such pieces of wood or metal.

Traditional typesetting is seldom used today. In modern typography, the term type usually refers to the images produced on typesetting or composition systems, which do not use wooden or metal type, such as photo and digital composition systems. The typography on a digital system, such as a digital computer, is called digital typography.

Digital typography is based on a hierarchy of objects called characters, fonts, and font families or typefaces. Numeric values or measurements related to those objects can be divided into character metrics, font metrics, and font family or typeface metrics. Sometimes all information about a font family or typeface is stored in a set of font files, but sometimes metric information for a set of font files is stored in a separate file called the font metric file.

What Is a Character?

A character is a graphical or mathematical representation of a glyph. Letters, digits, punctuation marks, mathematical symbols, and cursors are examples of glyphs.In a *bitmap* font, the shape of a character is usually represented by a rectangular bitmap. In an *outline* font, the shape of a character is usually represented by a mathematical description of its outline.

What Is a Font?

A font is a set of characters. There is a distinction between base fonts and composite fonts. A*base font* is a set of characters of the same size and style. A *composite font* is composed of base fonts with various attributes. Characters in a base font usually match each other in size, style, weight, and slant because their shape, size, position, and spacing have been carefully designed by a skilled font designer.

What Is a Font Family or Typeface?

A professional font designer usually creates an entire *font family* or *typeface*, rather than a single font. A base font family or typeface is a set of base fonts with the same style or design. A composite font family or typeface is composed of base font families or typefaces. A base font family or typeface can consist of bitmap fonts in certain sizes, a scalable font that can be used to produce bitmap fonts in different sizes, or both.

How Resolution Affects Font Size

The images on some output devices, such as laser printers and some types of video monitors, are created by drawing small dots or pixels (picture elements). The number of dots or pixels that can be drawn per unit of length in a horizontal direction is called the *horizontal resolution*, while the number of dots or pixels that can be drawn per unit of length in a vertical direction is called the *vertical resolution*. The most commonly used unit of measure for resolution is the number of dots per inch (dpi). The size of each dot or pixel decreases as the resolution of the output device increases and vice versa; therefore, dots and pixels are device−dependent units of measure.

To display the resolution of your video monitor, enter this command:

```
xdpyinfo | grep resol
```

You should get a response similar to this:

```
resolution: 96x96 dots per inch
```

If you draw all of the characters in a font in the same place (without advancing), you will get a composite image of those characters. If you then draw smallest rectangle that encloses that image, you will get the *bounding box* for that font. The size of a font is usually measured in the vertical direction. That size is usually not smaller than the height of a font bounding box, but it can be greater than that height. It may include additional vertical spacing that is considered part of the font design.

Typographers use small units of measure called *points* to specify font size. A point is approximately equal to 1/72 of an inch. The exact value is 1/72.27 (0.013837) of an inch or 0.351 mm.

A point is a device–independent unit of measure. Its size does not depend on the resolution of an output device. A 12–point font should have approximately the same size on different output devices, regardless of the resolution of those devices.

If the resolution of an output device is equal to 72 dots per inch (dpi), the size of a dot or pixel is approximately equal to the size of a point. If the resolution of an output device is greater than 72 dpi, the size of a dot or pixel is smaller than the size of a point, and vice versa. You can use the following formula to calculate a pixel size from a point size:

```
pixel–size = point–size x device–resolution / 72.27
```

A bitmap font is usually designed for a particular resolution. That font has the point size specified by its designer only when it is used on an output device whose resolution matches the resolution for which that font was designed. This is because a font designer specifies a fixed bitmap for each character. If a pixel is smaller than a point, characters will be smaller than intended, and vice versa.

Font Names

When a font is designed, it is assigned a name such as Courier Oblique. The Courier Oblique font belongs to a font family called Courier. The Courier font family includes the following fonts:

- Courier
- Courier Bold
- Courier Bold Oblique
- Courier Oblique

The size of a font is usually not specified for scalable fonts, because they can be scaled to any specified size. Bitmap fonts are usually designed in specific sizes. They are referred to by names such as 12–point Courier or 10–pixel Courier Bold.

When the PostScript page description software language was developed by Adobe Systems, the spaces embedded in font names were replaced with dashes. PostScript font names look like this:

```
Courier
Courier–Bold
Courier–BoldOblique
```

The X Consortium specified 14–part font names for the X Window System.

Figure 8–1 shows an example 14–part name for a bitmap font, with each part labeled. In this Point sizes in X font names are specified in *decipoints* (tenths of a point.)

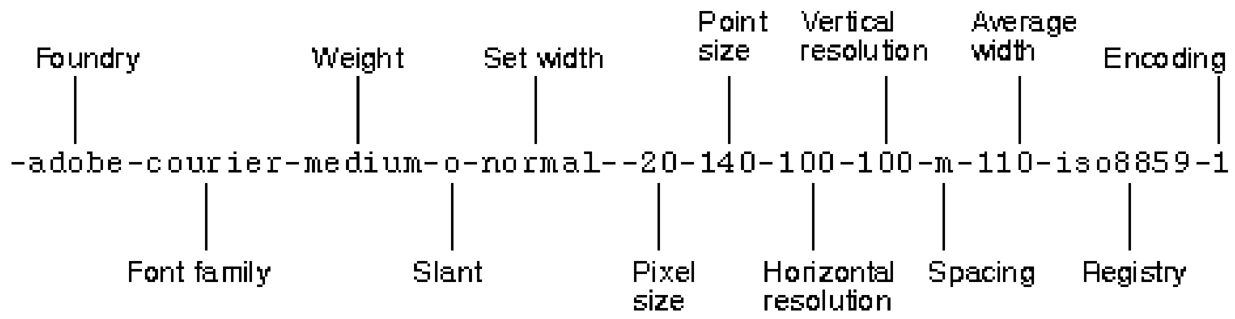

Figure 8–1 X Window System Font Name

For a detailed description of X font names, see the recommended X Window System documentation.

Writing Programs That Need to Use Fonts

You can write different types of programs for Silicon Graphics computers, for example, X, Display PostScript (DPS), GL, and mixed–model programs. Some of your programs need fonts. How a program accesses font files depends on the program type. X programs access fonts by calling X font functions, such as *XListFonts()* and *XLoadFont()*. DPS programs access fonts by calling X and DPS functions, or by using PostScript. GL and mixed–model programs usually access fonts by calling font management (fm) functions from the IRIS Font Manager library, such as *fmenumerate()* and *fmfindfont()*.

Most fonts are installed when you install the X Window System (X11 Execution Environment). Some fonts are installed with other software components, such as DPS, Impressario™ and IRIS Showcase™. Japanese fonts are installed when you install the Japanese Language System. However, most fonts are shared by the X Window System, DPS (which is an extension of the X Window System), IRIS Font Manager, and other software components.

To maintain compatibility and portability, it is best not to access font files directly from an application program because font formats, font names, font contents, and the location of font directories may change. Your program should use the Application Programming Interfaces (APIs) specified for the X Window System, DPS, and IRIS Font Manager, or call even higher level functions for the 2D and 3D text available from toolkits such as Inventor™ and Performer.

Using Fonts with the X Window System

This section describes how to use fonts with the X Window System.

The X Window System has several font utilities. This section touches on a few of the most useful utilities. For a complete description of those utilities, refer to your X Window System documentation.

Getting a List of Font Names and Font Aliases

To find out which font names and font aliases are known to the X Window System, enter this command:

```
xlsfonts > /tmp/fontlist
```

The resulting file, *fontlist*, contains entries such as:

```
-adobe-courier-bold-o-normal--0-0-0-0-m-0-iso8859-1
-adobe-courier-bold-o-normal--14-100-100-100-m-90-iso8859-1
-sgi-screen-medium-r-normal--14-140-72-72-m-70-iso8859-1
screen14
```

The first entry is an example of a 14–part X name for an outline (scalable) font. Numeric parts of font names are set to zero for outline fonts, because those fonts can be scaled to various sizes. The second and third entries are examples of 14–part X font names for bitmap fonts, while the last entry is an alias for the third entry. An X or DPS program can get a list of available fonts by calling *XListFonts()* or the function *XListFontsWithInfo()*.

Viewing Fonts

To see what a particular font looks like, use the command *xfd*, and specify a font name or font alias from the file *fontlist* by using the option *–fn*. For example, to display the 14–point Adobe Courier Bold font, enter:

```
xfd -fn -adobe-courier-bold-r-normal--14-140-75-75-m-90-iso8859-1
```

To request a Utopia Regular font scaled to the size of 28 points, enter:

```
   xfd -fn -adobe-utopia-medium-r-normal--0-280-0-0-p-0-iso8859-1
```

You can use wild cards (*) to indicate that any value is acceptable for parts of an X font name. For example, enter:

```
   xfd -fn "-*-courier-bold-r-normal--14-140-75-75-m-90-iso8859-1"
```

to indicate that xfd can use a Courier Bold font from any foundry. Enclose the X font name on a command line in single or double quotes when that name contains wild cards or embedded space characters.

The *xfd* command displays all characters in the specified font, as shown in **Figure 8–2** .

Figure 8–2 Sample Display from *xfd*

To open a shell window that uses a certain font, enter:

```
xwsh –fn font-name
```

Installing and Adding Font and Font Metric Files

This section explains how to install and add font files and font metric files for system–wide use.

Installing Font and Font Metric Files

By default, bitmap font files are installed in these directories: */usr/lib/X11/fonts/100dpi*, */usr/lib/X11/fonts/75dpi*, and */usr/lib/X11/fonts/misc*. The *100dpi* directory contains bitmap fonts designed for the screen resolution of 100 dpi. The *75dpi* directory contains bitmap fonts designed for the screen resolution of 75 dpi. The *misc* directory contains miscellaneous other fonts.

By default, outline font files in the Type 1 format are installed in the directory */usr/lib/DPS/outline/base*.

By default, font files in the directories listed above are used by the X Window System, DPS, IRIS Font Manager, and other software components.

The X Window System accesses Type 1 font files with symbolic links in the directory */usr/lib/X11/fonts/Type1*.

By default, outline font files in the Speedo™ format are installed in the directory */usr/lib/X11/fonts/Speedo* . Font files in this directory are currently used only by the X Window System.

Getting the Current X Font Path

To display the current X font path, enter this command:

```
xset q
```

In addition to other information, the *xset* utility displays font path information that may look like this:

```
Font Path:
/usr/lib/X11/fonts/100dpi/,/usr/lib/X11/fonts/75dpi/,
/usr/lib/X11/fonts/misc/,/usr/lib/X11/fonts/Type1/, /usr/lib/X11/fonts/Speedo
```

The X Window System checks the resolution of your video monitor. If that resolution is closer to 75 dpi than 100 dpi, it puts the directory 75dpi ahead of the directory 100dpi in the X font path.

By default, Adobe Font Metric (AFM) files are installed in the directory */usr/lib/DPS/AFM*.

There is one font metric file per typeface. Font metric files are primarily used by text–processing and desktop–publishing programs to, for example, generate PostScript code for a specified document.

Changing the X Font Path

X font pathYou can change the default X font path by using the option *fp* on an *xset* command line, followed by an equal sign (=), and a new font path (*newpath*):

```
xset fp=newpath
```

Adding Font and Font Metric Files

When you purchase fonts or obtain a font that is in the public domain, you may need to add that font to your system or printer in order to use it. Adobe Systems donated bitmap, outline, and font metric files for the Utopia font family to the X Consortium. This section shows how the font and font metric files for Utopia Regular were added to the IRIX operating system. Other font and font metric files can be added in a similar way.

You must log in as root to make any changes to X font directories. Before you make any changes to any

IRIX directory, you should make a copy of its contents so that you can restore that directory if anything goes wrong. For example, your font files may not be in the right format, and they may interfere with the access of Silicon Graphics font files. You should keep a log of the changes you make, and mention those changes when you report a problem with font files to Silicon Graphics; otherwise, it may be very difficult or impossible for other people to reproduce any problems that you might report.

Adding a Bitmap Font

As an example, we show how Utopia Regular bitmap fonts were added to IRIX. Other fonts can be added in a similar way.

To add the Utopia bitmap fonts to the X Window System and to Display PostScript:

1. Log in as root.
2. Look at the names of existing bitmap font files and try to match these names when you specify new font names. For example, Adobe provided two sets of Utopia Regular bitmap font files that were designed for the resolutions of 100 and 75 dpi. These files were in the extended Bitmap BDF 2.1 format.

   The names of the bitmap files were:

   ```
   UTRG_10.bdf through UTRG_24.bdf
   ```

   When these fonts were added to IRIX, the names were changed to:

   ```
   utopiaR10.bdf through utopiaR24.bdf
   ```

   to match the names of other X bitmap font files.
3. Convert the BDF font files to Portable Compiled Format (PCF) font files.

   BDF font files are text (ASCII) files. You can think of them as source font files. You could put BDF font files into an X font directory, but people usually use binary font formats such as the PCF (*pcf*) or compressed PCF format (*.pcf.Z*), so you should covert new fonts to one of these formats.

   To convert a BDF font file to a PCF font file, enter a command such as:

   ```
   bdftopcf –o file-name.pcf file-name.bdf
   ```

   Note: If you have used the *bdftopcf* command before, you may not have specified the name of the output file, but the command now requires that you enter the name of the output file.

   You can compress a PCF file by entering a command such as:

   ```
   compress file-name.pcf
   ```

   That should give you a file called:

   ```
   file-name.pcf.Z
   ```

   Most bitmap font files shipped by Silicon Graphics are in the compressed PCF format to save on disk space.
4. Put the bitmap font files for 100 dpi in the directory: */usr/lib/X11/fonts/100dpi*.

   You can tell the resolution for which a font was designed by the name of the directory in which the font designer stored the font files, or by the information in the header of a bitmap font file. In a BDF 2.1 font file, the horizontal (*x*) and vertical (*y*) resolution are specified in the X font name. They are also specified after the point size as the second and third numeric values in a SIZE entry. For example, the entry:

   ```
   SIZE 8 100 100
   ```

   indicates an 8–point font that was designed for the horizontal and vertical resolution of 100 dpi.

   For example, when 100–dpi Utopia Regular bitmap font files were added to IRIX, they were moved to the directory */usr/lib/X11/fonts/100dpi*.
5. Similarly, put the bitmap font files for 75 dpi in the directory */usr/lib/X11/fonts/75dpi*, and put other

bitmap fonts into the directory */usr/lib/X11/fonts/misc.*

For example, when 75–dpi Utopia Regular bitmap font files were added to IRIX, they were placed in the directory */usr/lib/X11/fonts/75dpi*.

6. Whenever you make changes in an X font directory, enter the command:

   ```
   mkfontdir
   ```

   to create a new *fonts.dir* (fonts directory) file in that directory. For example, when Utopia Regular fonts were added to the directories 100dpi and 75dpi, the command mkfontdir was executed in both of those directories.
7. Tell the X Window System that it should check again which fonts are available, by entering:

   ```
   xset fp rehash
   ```

8. To check whether the fonts you added are known to the X Window System, enter:

   ```
   xlsfonts > /tmp/fontlist
   ```

   The names of the fonts you added should appear on the list of font names and aliases produced by *xlsfonts*.

Bitmap fonts should now be added to the X Window System. Since DPS needs both outline and bitmap fonts for each supported typeface, it first checks which outline fonts are stored in the directory */usr/lib/DPS/outline/base*. Then it looks for the corresponding bitmap fonts in other X font directories. It ignores all other bitmap fonts. Therefore, DPS ignores the bitmap fonts you added, until you add the corresponding outline fonts.

To add these bitmap fonts to the IRIS Font Manager, you must insert an entry for each new typeface in the file */usr/lib/X11/fonts/ps2xlfd_map*. This file is used to map the PostScript and other short font names to X font names. A typical entry in that file looks like this:

```
Utopia-Regular -adobe-utopia-medium-r-normal--0-0-0-0-p-0-iso8859-1
```

You should not put any tabs in that file when you need to separate strings of characters; rather, use space characters. The pixel size, point size, horizontal resolution, vertical resolution, and average character advance width should all be set to 0, because this entry is like a template for all Utopia Regular bitmap font names.

Adding an Outline Font

To add the Utopia outline fonts to the X Window System and to Display PostScript:

1. Log in as root.
2. Look at the names of existing outline font files in the directory */usr/lib/DPS/outline/base*. Display PostScript requires that the name of each outline font file match the PostScript font name specified in the `/FontName` entry in the header of that outline font file. Enter:

   ```
   grep /FontName Courier-Bold
   ```

   You should get the response:

   ```
   /FontName /Utopia-Regular def
   ```

   You should put only Adobe text (ASCII) Type 1 font files or compatibles into that directory, not binary Type 1 font files or Type 3 font files. Display PostScript can handle Type 3 font files, but the X Window System and IRIS Font Manager cannot. You should put into that directory only those outline font files that look like the font files that are already in that directory.

   If you have a binary Type 1 font file (a *.pfb* file), you need to convert it into a text (ASCII) Type 1 font file before you can use it on a Silicon Graphics system. You can get the program for converting *.pfb* files to *.pfa* files from Silicon Graphics.

   For example, Adobe provided the Utopia Regular outline font file *UTRG____.pfa*, which is an outline font file in the Type 1 format, a special PostScript program that specifies a scalable outline font.

To find the PostScript font name for this font, enter:

```
grep /FontName UTRG____.pfa
```

You should get the response:

```
/FontName Utopia-Regular def
```

which tells you that the PostScript font name for this font is Utopia−Regular.

When this font was added to IRIX, the name of the file *UTRG____.pfa* was changed to *Utopia−Regular*.

3. Add the font to the */usr/lib/DPS/outline/base* directory.

   For example, when this font was added to IRIX the file *Utopia−Regular* was placed in the directory */usr/lib/DPS/outline/base*.

4. You should now be able to access the font file you added via Display PostScript or IRIS Font Manager. If you select Demos−>PostScript Demos−>Fontview from the Toolchest, you should see the new font in the list of available fonts. Display PostScript first checks for outline font files in the directory */usr/lib/DPS/outline/base*, then it looks for the corresponding bitmap font files. It ignores other bitmap font files.

5. Display PostScript (DPS) is an extension of the X Window System, so it knows how to access outline font files because the directory */usr/lib/DPS/outline/base* is a DPS directory. If you want the rest of the X Window System to access new outline fonts, you need to enter the following command:

   ```
   type1xfonts
   ```

   This re−creates symbolic links in the directory */usr/lib/X11/fonts/Type1* that point to outline font files in the directory */usr/lib/DPS/outline/base*.

Adding a Font Metric File

AFM files are primarily used by application programs—for example, to generate PostScript code for a specified document. Follow these steps to add a font metric file:

1. Log in as root.
2. Put Adobe Font Metric files in the directory */usr/lib/DPS/AFM*.

   For example, Adobe provided the Utopia Regular font metric file *UTRG____.AFM*.

   The name of an AFM file should also match the PostScript font name. When this font was added to IRIX, the name of the file *UTRG____.AFM* was changed to *Utopia−Regular*, and it was put in the directory */usr/lib/DPS/AFM*.

Downloading a Type 1 Font to a PostScript Printer

Some outline fonts are usually built into a PostScript printer. You can find out which fonts are known to the PostScript interpreter in your printer by sending the following file to that printer:

```
%!
% Produce a list of available fonts
/f 100 string def
/Times-Roman findfont 12 scalefont setfont
/y 700 def
72 y moveto
FontDirectory {
pop f cvs show 72 /y y 13 sub def y moveto
} forall
showpage
```

Utopia fonts are not usually built into PS printers. If you try to print a document that requires a Utopia font on a PS printer that does not have that font, a warning message about the replacement of a missing font with a Courier font is sent to the file */usr/spool/lp/log* on the machine to which that PS printer is attached.

You can download a Type 1 font to a PS printer in either of the following two ways:

1. You can insert a Type 1 font file at the beginning of the PostScript file that needs that font. You should have a statement that starts with:

   ```
   %!
   ```

   at the beginning of your PS file. If you have two such lines, delete the second one.

   When you download a font this way, the font is available only while your print job is being processed.
2. You can make a copy of a Type 1 font file, and then insert the statement:

   ```
   serverdict begin 0 exitserver
   ```

   after the first group of comment statements (lines that start with %) if no password has been specified for your printer; otherwise, you should replace 0 in the above statement with the password for your printer. Then you should send the edited file to your printer.

   When you download a font this way, the warning message:

   ```
   %%[ exitserver: permanent state may be changed ]%%
   ```

   is sent to the file */usr/spool/lp/log* on the machine to which the printer is attached.

   The permanent state of the printer is not really changed. Downloaded fonts disappear when you reset the printer by switching its power off and on. If there is not enough memory for additional fonts, you receive a message about a Virtual Memory (VM) error, and the font is not downloaded.

   If you again send the program that produces a list of available fonts to your printer, you should see the PostScript names of the fonts you downloaded on that list.

Chapter 9

Internationalization

Internationalization is the process of generalizing an application so that it can easily be customized—or *localized*—to run in more than one language environment. Internationalized software can be made to produce output in a user's native language, to format data (such as dates and currency values) according to the user's local customs, and to otherwise make the software easier to use for users from a culture other than that of the original software developer. As computers become more widely used in non−American cultures, it becomes increasingly more important that developers stop relying on the conventions of American programming and the English language in their programs; this chapter provides information on how to make your applications more widely accessible.

Overview

This chapter discusses internationalization standards, local−custom databases ("locales"), and character−encoding methods other than traditional ASCII. It also describes culture−specific output formatting; X11, Release 5 (X11R5) support for internationalization; internationalizing user input; and internationalizing applications that use Graphical User Interfaces (GUIs). The chapter concludes with a discussion of a few common non−ASCII encodings.

Terminology

This section defines some of the terms used in this chapter.

Locale

Locale refers to a set of local customs that determine many aspects of software input and output formatting, including natural language, culture, character sets and encodings, and formatting and sorting rules. The locale of a program is the set of such environmental parameters that are currently selected. For information on the method for selecting locales, see **"Locales,"** below.

Internationalization (I18n)

Internationalization is the process of making a program capable of running in multiple locales without recompiling. To put it another way, an internationalized program is one that can be easily localized without changing the program itself. (See **"Localization (L10n),"** below, for an explanation of the term "localization.")

Note: The word "internationalization" consists of an 'i' followed by 18 letters followed by an 'n.' It is thus commonly abbreviated "i18n." "I18n" is pronounced "internationalization," not "eye−eighteen−enn."

A program written for a specific locale may be difficult to run in a different environment. Rewriting such a program to operate in each desired environment would be tedious and costly.

Your goal as a developer should thus be to write *locale−independent* programs, programs that make no assumptions about languages, local customs, or coded character sets. Such internationalized applications can run in a user's native environment following native conventions with native messages, without recompiling or relinking. A single copy of an internationalized program can be used by a world of different users.

Localization (L10n)

Localization is the act of providing an internationalized application with the environment and data it needs to operate in a particular locale. For example, adding German system messages to IRIX is a part of localizing IRIX for the German locale.

Note: Localization is often abbreviated "l10n."

Nationalized Software

Nationalized programs run in only one language and are governed by one set of customs; in other words, in a nationalized program the locale is built into the application. Even if the application doesn't use ASCII or English, as long as it is a single−language program it is nationalized, not internationalized. Most older UNIX programs can be thought of as being nationalized for the United States.

Consider two applications, *hello* and *bonjour*. *hello* always produces the output

```
Hello, world.
```

and *bonjour* always produces

```
Bon jour, tout le monde.
```

Neither *hello* nor *bonjour* are internationalized; they are both nationalized.

There are no special requirements for writing or porting nationalized applications, whether they are text or graphics programs. Terminal−based programs work on suitable terminals, including internationalized terminal emulators. "Suitable" means that the terminal supports any necessary fonts and understands the encoding of the application output. Graphics programs simply do as they have always done. Applications using existing interfaces to operate in non−English or non−ASCII environments should continue to compile and run under an internationalized operating system.

Multilingual Software

A *multilingual* program is one that uses several different locales at the same time. Examples are described in **"Multilingual Support"** .

Areas of Concern in Internationalizing Software

Few developers will have to pay attention to more than a few items described in this chapter. Most will need to catalog their strings. Some will need to use library routines for character sorting or locale−dependent date, time, or number formatting. A few whose applications use the eighth bit of 8−bit characters inappropriately will need to stop doing so. The few applications that do arithmetic to manipulate characters will need to be cleaned up. Some GUI designers will have to spend just a little more time thinking. But for the large majority of developers, there isn't much to do.

In this chapter, we will attempt to address every internationalization issue that a developer might wonder about, so there may be sections that are not germane to your applications.

Standards

IRIX internationalization includes these standards−compliant features, among others:

- ANSI C and POSIX (ISO 9945−1): Locale
- *X/OPEN Portability Guide, Issue 3* (XPG/3): XPG/3 message catalogs, interpretation of locale strings
- AT&T UNIX™ System V Release 4: Multi−National Language Support (MNLS) message catalogs
- X11R5: Input methods, text rendering, resource files

Locales

An internationalized system is capable of presenting and receiving data understandably in a number of different formats, cultures, languages and character sets. An application running in an internationalized system must indicate how it wants the system to behave. IRIX uses the concept of a locale to convey that information.

A process can have only one locale at a time. Most internationalization interfaces rely on the locale of the current process being set properly; the locale governs the behavior of certain library routines.

Setting the Current Locale

Applications begin in the *C* locale. (*C* is the name used to indicate the system default locale; it usually corresponds to American English.) Applications should therefore call *setlocale()* as soon as possible to put the process into the desired locale. The syntax for *setlocale()* is:

```
#include <locale.h>
char *setlocale(int category, const char *locale);
```

The call almost always looks like this:

```
if (setlocale(LC_ALL, "") == NULL)
      exit_with_error();
```

Details of the two parameters are given in the next two sections.

Category

Applications need not perform every aspect of their work in the same locale. Although this approach is not recommended, an application could (for example) perform most of its activities in the English locale but use French sorting rules. You can use locale categories to do this kind of locale−mixing. (Mixing locale categories is not the same as multilingual support—see **"Multilingual Support."**)

The *category* argument is a symbolic constant that tells *setlocale()* which items in a locale to change. **Table 9−1** lists the available category choices.

Table 9−1 Locale Categories

Category	Affects
LC_ALL	All categories below
LC_COLLATE	Regular expressions, *strcoll()*, and *strxfrm()*
LC_CTYPE	Regular expressions and ctype routines (such as *islower()*)
LC_MESSAGES[a]	*gettxt()*, *pfmt()*, and *nl_langinfo()*
LC_MONETARY	*localeconv()*

LC_NUMERIC	Decimal−point character for formatted I/O and nonmonetary formatting information returned by *localeconv()*
LC_TIME	*ascftime()*, *cftime()*, *getdate()*, and *strftime()*

[a] LC_MESSAGES is supported by SVR4 but isn’t required by XPG/3.

Categories correspond to databases that contain relevant information for each defined locale. The locations of these databases are given in the **"Location of Locale−Specific Data"** section, below.

Locale

The *setlocale()* function attempts to set the locale of the specified category to the specified locale. You should almost always pass the empty string as the *locale* parameter to conform to user preferences.

On success, *setlocale()* returns the new value of the category. If *setlocale()* couldn’t set the category to the value requested, it returns NULL and does not change locale.

The Empty String

An empty string passed as the *locale* parameter is special. It specifies that the locale should be chosen based on environment variables. This is the way a user specifies a preferred locale, and that preference should almost always be honored. The variables are checked hierarchically, depending on category, as shown in **Table 9−2** ; for instance, if the category is LC_COLLATE, an empty−string locale parameter indicates that the locale should be chosen based on the value of the environment variable LC_COLLATE—or, if that value is undefined, the value of the environment variable LANG, which should contain the name of the locale that the user wishes to work in.

Table 9−2 Category Environment Variables

Category	First Environment Variable	Second Environment Variable
LC_COLLATE	LC_COLLATE	LANG
LC_CTYPE	LC_CTYPE	LANG
LC_MESSAGES	LC_MESSAGES	LANG
LC_MONETARY	LC_MONETARY	LANG
LC_NUMERIC	LC_NUMERIC	LANG
LC_TIME	LC_TIME	LANG

Specifying the category LC_ALL attempts to set each category individually to the value of the appropriate environment variable.

If no non−null environment variable is available, *setlocale()* returns the name of the current locale.

Nonempty Strings in Calls to *setlocale()*

Here are the possibilities for specifying a nonempty string as the *locale* parameter:

NULL string	Specifying a locale value of NULL—not the same as the empty string—causes *setlocale()* to return the name of the current locale.
"C"	Specifying a locale value of the single−character string "C" requests whatever locale the system uses as a default. (Note that this is a string and not just a character.)
Other nonempty strings	Requesting a particular locale to be used by specifying its name. Overrides any user preferences; this should only be done with good reason.

Location of Locale−Specific Data

Except for XPG/3 message catalogs, locale−specific data (that is, the "compiled" files containing the collation information, monetary information, and so on) are located in */usr/lib/locale/*locale/category, where *locale* and *category* are the names of the locale and category, respectively. For example, the database for the LC_COLLATE category of the French locale *fr* would be in */usr/lib/locale/fr/LC_COLLATE*.

There will probably be multiple locales symbolically linked to each other, usually in cases where a specific locale name points to the more general case. For example, */usr/lib/locale/En_US.ascii* might point to */usr/lib/locale/C*.

Locale Naming Conventions

A locale string is of the form:

`language[_territory[.encoding]][@modifier]...`

where:

- *language* is the two−letter ISO 639 abbreviation for the language name.
- *territory* is the two−uppercase−letter ISO 3166 abbreviation for the territory name. (For a list of these abbreviations, see the table in **Appendix B, "ISO 3166 Country Names and Abbreviations."**)
- *encoding* is the name of the character encoding (mapping between numbers and characters). For western languages, this is typically the codeset, such as 8859−1 or ASCII. For Asian languages, where an encoding may encode multiple codesets, the encodings themselves have names, such as UJIS or EUC (these encodings are described later in this chapter). Codesets and encodings are discussed under **"Character Sets, Codesets, and Encodings,"** below.
- *modifier*s are not actually part of the locale name definition; they give more specific information about the desired localized behavior of an application. For example, under X11R5, a user can select an input method with modifiers. (To use the Xsi wnn Japanese input method, for instance, one adds *@im=_XWNMO* to the locale string.) There are no standards for this part of a locale string.

Language data is implementation specific; databases for the language *en* (English) might contain British cultural data in England and American cultural data in the United States. If other than the default settings are required, the territory field may be used. For example, the above cases could be more strictly defined by setting LANG to *en_EN* or *en_US*. Full rigor might lead to *en_EN.88591* for England (the locale encoding specification for ISO 8859−1 is "88591") and *en_US.ascii* for the USA.

ANSI C has defined a special locale value of *C*. The *C* locale is guaranteed to work on all compliant systems and provides the user with the system's default locale. This default is typically American English and ASCII, but need not be. POSIX has also defined a special locale value, *POSIX*, which is identical to the *C* locale.

The length of the locale string may not exceed NL_LANGMAX characters (NL_LANGMAX is defined in */usr/include/limits.h*). However, XPG/3 recommends that this string (not counting modifiers) not exceed 14 characters.

Limitations of the Locale System

Multilingual Support

There can only be one locale at a time associated with any given process in an internationalized system. Therefore, although multilingual applications—which give the appearance of using more than one locale at a time—can be created, internationalization does not provide inherent support for them. Here are two examples of multilingual programs:

- An application creates and maintains windows on four different displays, operated by four different users. The program has a single controlling process, which is associated with only one locale at any given time. However, the application can switch back and forth between locales as it switches between users, so the four users may each use a different locale.
- In a sophisticated editing system with a complex user interface, a user may wish to operate the interface in one language while entering or editing text in another. For instance, a user whose first language is German may wish to compose a Japanese document, using Japanese input and text manipulation, but with the user interface operating in German. (There is no standard interface for such behavior.)

In writing a multilingual application, the first task is identifying the locales for the program to run in and when they apply. (There is no standard method for performing this task.) Once the application has chosen the desired locales, it must either one of the following:

- fork, and then call *setlocale()* differently in each process
- call *setlocale()* repeatedly as necessary to change from language to language

Misuse of Locales

The LANG environment variable and the locale variables provide the freedom to configure a locale, but they do not protect the user from creating a nonsensical combination of settings. For example, you are allowed to set LANG to *fr* (French) and LC_COLLATE to *ja* (Japanese). In such a case, string routines would assume text encoded in 8859−1—except for the sorting routines, which might assume French text and Japanese sorting rules. This would likely result in arbitrary−seeming behavior.

No Filesystem Information for Encoding Types

The IRIX filesystem does not contain information about what encoding should be associated with any given data. Thus, applications must assume that data presented to an application in some locale is properly encoded for that locale. In other words, a file is interpreted differently depending on locale; there is no way to ask the file what it thinks its encoding is.

For example, you may have created a file while in a Japanese locale using SJIS. Later, you might try printing it while in a French locale. The results will likely resemble a random collection of Latin 1 characters.

This problem applies to almost all stored strings. Most strings are uninterpreted sequences of nonzero bytes. This includes, for example, filenames. You can, if you want to, name your files using Chinese characters in a Chinese locale, but the names will look odd to anyone who runs */bin/ls* on the same filesystem using a non−Chinese locale.

Character Sets, Codesets, and Encodings

One major difference between nationalized and internationalized software is the availability in internationalized software of a wide variety of methods for encoding characters. Developers of internationalized software no longer have the convenience of always being able to assume ASCII.

A *character set* is an abstract collection of characters. A *codeset* is a character set with exactly one associated numerical encoding for each character. The English alphabet is a character set; ASCII is a codeset.

An *encoding* is also a set of characters and associated numbers, but the term is more general than "codeset." A single encoding may include multiple codesets; *Extended Unix Code (EUC)*, for instance, is an encoding that provides for four codesets in one data stream. (For details, see the **"EUC"** section later in this chapter.)

Eight−Bit Cleanliness

A program is *8−bit clean* if it does not use the high bit of any data byte to convey special information. ASCII characters are specified by the low seven bits of a byte, so some programs use the high bit of a data byte as a flag; such programs are not 8−bit clean. Internationalized programs must be 8−bit clean, because they cannot expect data to be in the form of ASCII bytes; non−ASCII character sets usually use all eight bits of each byte to specify the character. But a program must go out of its way to manipulate bytes based on the value of the high bit; and since changing data without cause is seldom desirable, most programs are already 8−bit clean.

The old *csh* (before this problem was fixed in the IRIX 5.0 release) was a good example of a program that was not 8−bit clean; it used the high bit in input strings to distinguish aliases from unaliased commands. An effect of this misuse was that *csh* stripped the 8th bit from all characters:

```
echo I know an architect named Mañosa
I know an architect named Maqosa
```

Another example is the specification of Internet messages, which calls for 7−bit data. Thus, if *sendmail* fails to strip the 8th bit from a character prior to sending it, it violates a protocol; if it does strip the bit, it could garble a non−ASCII message (this protocol problem is being addressed).

One of the simplest things to do to remove the American bias from a program is to replace the ASCII assumption with the assumption that the Latin 1 codeset will be used. This approach is not true internationalization, but it can make the application usable in most of Western Europe. Latin 1 uses only one byte per character, unlike some other codesets, so 8−bit clean ASCII software should work without modification using the Latin 1 codeset.

Ensuring that code is 8−bit clean is the single most important aspect of internationalizing software.

Note: Another caveat about 8−bit characters only applies to a particular set of circumstances: if you're not using a multibyte character type (see the next section), you should not declare characters as type *signed char*. (The default in IRIX C is for *char* to imply *unsigned char*.) If you try to cast a *signed char* to an *int* (as you must do to use the *ctype*(3C) functions) and the character's high bit is set (as it may be in an 8−bit character set), the high bit is interpreted as a sign bit and extends into the full width of the int.

Multiple−Byte Character Types

Western languages usually require only one byte for each character. Asian languages, however, often require two or even four bytes per character; and some Asian encodings allow a variable number of bytes per character.

The two kinds of encodings that allow more than one byte per character are wide (*WC* or *wchar*) characters (which are a fixed number of bytes long) and multibyte (*MB*) characters (which are of variable size). The application developer must decide where to use WC and MB characters and strings:

- Multibyte strings are almost the default: string I/O uses MB, MB code works for ASCII and ISO 8859, and MB characters use less space than do wide characters. However, manipulating individual characters within a multibyte string is difficult.
- Applications that do heavy string manipulation typically use WC strings for such activity, because manipulating individual WC characters in a string is much simpler than doing the same thing with MB characters. So wide characters are used as necessary to provide programming ease or runtime speed; however, they take up more space than MB characters.

Traditional ASCII (8859) Strings

Traditional strings are merely a special case of multibyte strings, where every character happens to be one byte long and there is only one codeset. All MB code, including conversion to and from wchars, works for traditional ASCII, or ISO 8859, strings.

Multibyte Characters

A multibyte character is a series of bytes. The character itself contains information on how many bytes long it is. Multibyte characters are referenced as strings (and are therefore of type *char **); before parsing, a string is indistinguishable from a multibyte character. The zero byte is still used as a string (and MB character) terminator.

A string of MB characters can be considered a null−terminated array of bytes, exactly like a traditional string. A multibyte string may contain characters from multiple codesets. Usually, this is done by incorporating special bytes that indicate that the next character (and only the next character) will be in a different codeset. Very little application code should ever need to be aware of that, though; you should use the available library routines to find out information about multibyte strings rather than look at the underlying byte structure, because that structure varies from one encoding to another. For one example of an encoding that allows characters from multiple codesets, see **"EUC."**

Use of Multibyte Strings

Multibyte strings are very easy to pass around. They efficiently use space (both data and disk space), since "extra" bytes are used only for characters that require them. MB strings can be read and written without regard to their contents, as long as the strings remain intact. Displaying MB strings on a terminal is done with the usual routines: *printf()*, *puts()*, and so on. Many programs (such as *cat*) need never concern themselves with the multibyte nature of MB strings, since they operate on bytes rather than on characters; so MB strings are often used for string I/O.

Manipulation of individual characters in an MB string can be difficult, since finding a particular character or position in a string is nontrivial (see **"Handling Multibyte Characters,"** below). Therefore, it is common to convert to WC strings for that kind of work.

Handling Multibyte Characters

Usually, multibyte characters are handled just like *char* strings. Editing such strings, however, requires some care.

You cannot tell how many bytes are in a particular character until you look at the character. You cannot look at the *n*th character in a string without looking at all the previous *n* – 1 characters, because you cannot tell where a character starts without knowing where the previous character ends. Given a byte, you don't know its position within a character. Thus, we say the string has *state* or is *context–sensitive*; that is, the interpretation we assign to any given byte depends on where we are in a character.

This analysis of characters is locale–dependent, and therefore must be done by routines that understand locale.

Conversion to Constant–Size Characters

Multibyte characters and strings are convertible to wchars via *mbtowc(*3*)* (individual characters) and *mbstowcs(*3*)* (strings).

How Many Bytes in a Character?

To find out how many bytes make up a given single MB character, use *mblen(*3*)*:

```
#include <stdlib.h>
. . .
size_t n;
int len;
char *pStr;
. . .
len = mblen(pStr, n); /* examine no more than n bytes */
```

It is the application's responsibility to ensure that *pStr* points to the beginning of a character, not to the middle of a character.

The maximum number of bytes in a multibyte character is MB_LEN_MAX, which is defined in *limits.h*. The maximum number of bytes in a character under the current locale is given by the macro MB_CUR_MAX, defined in *stdlib.h*.

How Many Bytes in an MB String?

Since *strlen()* simply counts bytes before the first NULL, it tells you how many bytes are in an MB string.

How Many Characters in an MB String?

When *mbstowcs()* coverts MB strings to WC strings, it returns the number of characters converted. This is the simplest way to count characters in an MB string.

Note: Many code segments that need to deal with individual characters within a string would be better suited by wide character strings. Since counting often involves conversion, such segments are often better served by working with a WC string, then converting back.

Getting the length without performing the conversion is straightforward, but not as simple. *mbtowc()* converts one character and returns the number of bytes used, but returns the same information without conversion if a NULL is passed as the address of the WC destination. Thus:

```
len = mblen(pStr, n);
```

is equivalent to

```
len = mbtowc((wchar_t *) NULL, pStr, n);
```

In fact, *mblen()* calls *mbtowc()* to perform its count. Therefore, counting characters in an MB string without converting would look like this:

```
int cLen;
char *tStr = pStr;

numChars = 0;
cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
while (cLen > 0) {
      tStr += cLen;
      numChars++;
      cLen = mbtowc((wchar_t *) NULL, tStr, MB_CUR_MAX);
      if (cLen == -1)
          numChars = cLen; /* invalid MB character */
}
```

Wide Characters

A wide character (*WC* or *wchar*) is a data object of type *wchar_t*, which is guaranteed to be able to hold the system's largest numerical code for a character. *wchar_t* is defined in *stdlib.h*. Under IRIX 4.0.x, *sizeof(wchar_t)* was 1. Under IRIX 5.1, it is 4. All wchars on a system are the same size, independent of locale, encoding, or any other factors.

Uses for *wchar* Strings

The single advantage of WC strings is that all characters are the same size. Thus, a string can be treated as an array, and a program can simply index into the array in order to modify its contents. Most applications' *char* manipulation routines work with little modification other than a type change to *wchar_t*, with appropriate attention to byte count and *sizeof()*.

So, when applications have significant string editing to perform, they typically keep the strings in WC format while doing that editing. Those WC strings may or may not be converted to or from MB strings at other points in the application.

Wide characters are often large and are not as space efficient as multibyte strings. Applications that do not need to perform string editing probably shouldn't use *wchar*s. If an application intends to both maintain and edit large numbers of strings, then the developer needs to make size/complexity trade−off decisions.

Support Routines for Wide Characters

Analogs to the routines defined in *string.h* and *stdio.h* are supplied in *libw.a* and defined in *widec.h*. This includes routines such as *getwchar()*, *putwchar()*, *putws()*, *wscpy()*, *wslen()*, and *wsrchr()*.

Conversion to MB Characters

Wide characters and strings are convertible to MB strings via *wctomb()* and *wcstombs()*, respectively.

Reading Input Data

Input can be divided into two categories: user events and other data. This section deals with non−user−originated data, which is assumed to come from file descriptors or streams. User events are discussed in **"User Input"** .

It is generally fair to assume that unless otherwise specified, data read by an application is encoded suitably for the current locale. Text strings typically are in MB format.

Streams can be read in WC format by using routines defined in *widec.h*.

Cultural Items

This section discusses several aspects of a locale that may differ between locales.

String Collation

Different locales can have different rules governing collation of strings, even within identical encodings.

The Issue

In English, sorting rules are extremely simple: each character sorts to exactly one unique place. Under ASCII, the characters are even in numeric order. However, neither of those statements is necessarily true for other languages and other codesets. Furthermore:

- Sorting order for a language may be completely unrelated to the (numerical) order of the characters in a given encoding.
- Even with a correctly sorted list of the characters in a character set, you may not be able to sort words properly.
- Locales using identically encoded character sets may use very different sorting rules.

Programs using ASCII can do simple arithmetic on characters and directly calculate sorting relationships; such programs frequently rely on truisms such as the fact that:

```
'a' < 'b'
```

in ASCII. But internationalized programs cannot rely on ASCII and English sorting rules. Consider some non–English collation rule types:

- *One–to–Two* mappings collate certain characters as if they were two. For example, the German *ß* collates as if it were "ss."
- *Many–to–One* mappings collate a string of characters as if they were one. For example, Spanish sorts "ch" as one character, following "c" and preceding "d." In Spanish, the following list is in correct alphabetical order: *calle*, *creo*, *chocolate*, *decir*.
- *Don't–Care Character* rules collate certain characters as if they were not present. For example, if "–" were a don't–care character, "co–op" and "coop" would sort identically.
- *First–Vowel* rules sort words based first on the first vowel of the word, then by consonants (which may precede or follow the vowel in question).
- *Primary/Secondary* sorts consider some characters as equals until there is a tie. For example, in French, a, á, à, and â all sort to the same primary location. If two strings (such as "tache" and "tâche") collate to the same primary order, then the secondary sort distinguishes them.
- Special case sorts exist for some Asian languages. For example, Japanese *kanji* has no strict sorting rules. *Kanji* strings can be sorted by the strokes that make up the characters, by the *kana* (phonetic) spellings of the characters, or by other agreed–upon rules.

It should be clear that a programmer cannot hope to collate strings by simple arithmetic or by traditional methods.

The Solution

Locale–specific collation should be performed with *strcoll()* and *strxfm()*. These are table–driven functions; the tables are supplied as part of locale support. The value of LC_COLLATE determines which ordering table to use. *strcoll()* has the same interface as *strcmp()*; it can be directly substituted into code that uses *strcmp()*.

Specification of Numbers and Money

Format of simple numbers differs from locale to locale. Characters used for decimal radix and group separators vary. Grouping rules may also vary. Even though we assume that decimal numbers are universal, there are some eighteen varying aspects of numeric formatting defined by a locale. Many of these are details of monetary formatting.

For example, Germany uses a comma to denote a decimal radix and a period to denote a group separator. English reverses these. India groups digits by two except for the last three digits before the decimal radix. Many locales have particular formats used for money, some of which are shown in **Table 9–3** :

Table 9–3 Some Monetary Formats

Country	Positive Format	Negative Format

India	Rs1,02,34,567.89	Rs(1,02,34,567.89)
Italy	L.10.234.567	−L.10.234.567
Japan	¥10,234,567	−¥10,234,567
Netherlands	F10.234.567,89	F−10.234.567,89
Norway	Kr10.234.567,89	Kr10.234.567,89−
Switzerland	SFr10,234,567.89	SFr10,234,567.89C

printf()

printf() examines LC_NUMERIC and chooses the appropriate decimal radix. If none is available, it tries to use ASCII period. No further locale-specific formatting is done directly by *printf()*, but see **"Variably Ordered Referencing of printf() Arguments,"** later in this chapter, for a way to do your own locale-specific output formatting.

localeconv()

The *localeconv()* function can be called to find out about numeric formatting data, including the decimal radix (inappropriately called *decimal_point*), the grouping separator (inappropriately called *thousands_sep*), the grouping rules, and a great deal of monetary formatting information. Actual use of formatting information other than the decimal radix is left to the application; there aren't any special print routines that produce formatted numbers according to all of *localeconv()*'s data.

Dates and Times

All of these can mean the same date to different people:

```
92.1.4
4/1/92
1/4/92
```

All of these can mean the same time to different people:

```
2:30 PM
14:30
14h30
```

Dates and times can be easily formatted by using *strftime()*, which gives a host of options for displaying locale-specific dates and times. The *ascftime()* and *cftime()* functions give further options, but should be avoided because they do not conform to ANSI and XPG/3 specifications. The old *asctime()* and *ctime()* functions are now obsolete; use *strftime()* instead. For more information, see the *strftime(*3C*)* manual page.

Character Classification and *ctype*

The *ctype.h* header file is described in the *ctype*(3C) manual page and defines macros to determine various kinds of information about a given character: *isalpha()*, *isupper()*, *islower()*, *isdigit()*, *isxdigit()*, *isalnum()*, *isspace()*, *ispunct()*, *isprint()*, *isgraph()*, *iscntrl()*, and *isascii()*.

The Issue

When programmers knew that a character set was ASCII, some convenient assumptions could be made about characters and letters. It was common for programmers to do arithmetic with the ASCII code values in order to perform some simple operations. For example, raising a character to upper case could be done by subtracting the difference between the code for *a* and the code for *A*. Numeric characters could be identified by inspection: if they fell between *0* and *9*, they were numeric; otherwise, they weren't. You could tell if a character was (for instance) printable, a letter, or a symbol by comparing to known encoding values. Macros for such activity have long been available in *ctype.h*, but lots of programs did character arithmetic anyway. Since character encoding and linguistic semantics are completely independent, such arithmetic in an internationalized program leads to unpleasant results.

Furthermore, characters exist outside of ASCII that break some non-arithmetic assumptions. Consider the German character *ß* which is a lowercase alphabetic character (letter), yet has no uppercase.

the German character *ß* which is a lowercase alphabetic character (letter), yet has no uppercase. Consider also French (as written in France), where the uppercase of *é* is *E*, not *É*.

Clearly, the programmer of an internationalized application has no way of directly computing all the character associations that were available in English under ASCII.

The Solution

Strict avoidance of arithmetic on character values should remove any trouble in this area. The macros in *ctype.h* are table−driven and are therefore locale−sensitive. If you think of characters as abstract characters rather than as the numbers used to represent them, you can avoid a lot of pitfalls in this area.

Functions Instead of Macros

There is a function corresponding to each of the above−mentioned macros in *ctype.h*. In order to get a function instead of a macro, simply undefine the name, for example:

```
#undef toascii
char (*xlate)(char);
char a, b;
...
xlate = toascii;
a = (*xlate)(b);
```

Regular Expressions

XPG/3 specifies some extensions to traditional regular expression syntax for internationalized software. Few application developers do their own regular expression parsing and matching, however, so we do not include full details here. Briefly, the extensions provide the ability to specify matches based on:

- character class (such as *alpha*, *digit*, *punct*, or *space*)
- equivalence class (for instance: *a*, *á*, *à*, *â*, *A*, *Á*, *À*, and *Â* may be equivalent)
- collating symbols (allowing you to match the Spanish "ch" as one element because it is a single collating token)
- generalization of range specifications of the form $[c_1-c_2]$ to include the above

Programmers who process expressions themselves need the full description of internationalized regular expression grammar in Volume 3, Chapter 6, of XPG/3.

Strings and Message Catalogs

Message catalogs are compiled databases of strings. While a major role of message catalogs is to provide communications text in locale−specific natural language, the strings can be used for any purpose. The idea is that an application uses only strings from a catalog, thus allowing localizers to supply catalogs suitable for a given locale.

There are two different and incompatible interfaces to message catalogs in IRIX: *MNLS* and *XPG/3*. Developers working on SVR4 or other AT&T code, or related base−system utilities, are probably using MNLS. Developers working on independent projects are probably using XPG/3. Neither is a solid standard, but XPG/3 is closer to being one than MNLS, and its prognosis looks brighter. Thus applications developers who have to choose between the two interfaces are encouraged to use XPG/3 in order to maximize their portability.

Note: XPG/3 seems to be more popular in Europe. Asia seems to lean toward MNLS.

XPG/3 Message Catalogs

The XPG/3 message catalog interface requires that a catalog be opened before it is read, and requires that catalog references specify a catalog descriptor.

Since catalog references include a default to be used in case of failure, applications will work normally without a catalog when in the default locale. This means catalog generation is exclusively the task of localizers. But in order to inform the localizer as to what strings to translate and how they should comprise a catalog, the application developer should provide a catalog for the developer's locale.

Opening and Closing XPG/3 Catalogs

catopen() locates and opens a message catalog file:

```
#include <nl_types.h>
nl_catd catopen(char *name, int unused);
```

The argument *name* is used to locate the catalog. Usually, this is a simple, relative pathname that is combined with environment variables to indicate the path to the catalog (see **"XPG/3 Catalog Location"** for details). However, names that begin with "/ " are assumed to be absolute pathnames to the catalog. Use of a hard−coded pathname like this is strongly discouraged; it doesn't allow the user to specify the catalog's locale through environment variables.

When an application is finished using a message catalog, it should close the catalog and free the descriptor using *catclose()*:

```
int catclose(nl_catd);
```

Using an XPG/3 Catalog

Catalogs contain sets of numbered messages. The application developer must know the contents of the catalog in order to specify the set and number of a message to be obtained.

catgets() is used to retrieve strings from a message catalog:

```
#include <nl_types.h>
char *catgets (nl_catd catd, int set_num, int msg_num,
                 char *defaultStr);
```

catgets() retrieves message *msg_num* from set *set_num* from the catalog described by *catd*. If for any reason *catgets()* cannot do this, it returns *defaultStr*. **Example 9−1** shows a program which reads the first message from the first message set in the appropriate catalog, and displays the result.

Example 9−1 Reading an XPG/3 Catalog

```
#include <stdio.h>
#include <locale.h>
#include <nl_types.h>

#define SET1      1
#define WRLD_MSG  1

int main(){
      nl_catd msgd;
      char *message;
      setlocale(LC_ALL, "");

      msgd = catopen("hw",0);
      message = catgets(msgd, SET1, WRLD_MSG,"Hello, world\n");
      printf(message);
      catclose(msgd);
}
```

The above example uses *printf()* instead of *puts()* in order to make a point: the format string of *printf()* came from a catalog. Note that:

```
printf(catgets(msgd, set, num, defaultStr));
```

is very different from:

```
printf("%s", catgets(msgd, set, num, defaultStr));
```

because strings in catalogs can contain formatting strings of the kind used by *printf()*. For further discussion of issues relating to this important distinction, see **"Variably Ordered Referencing of printf() Arguments."**

XPG/3 Catalog Location

XPG/3 message catalogs are located using the environment variable NLSPATH. The default NLSPATH is */nlslib/%L/%N*, where *%L* is filled in by the LANG environment variable and *%N* is filled in by the *name* argument to *catopen()*. NLSPATH can specify multiple pathnames in ordered precedence, much like the PATH variable. A sample NLSPATH assignment:

```
NLSPATH=/usr/lib/locale/%L/%N:/usr/local/lib/locale/%L/%N:/usr/defaults/%N
```

Full details are in the *catopen(3)* manual page.

Creating XPG/3 Message Catalogs

Message catalogs are of this general form:

```
$set n   comment
a   message-a\n
b   message-b\n
c   message-c\n
$quote "
d " message-d "
$this is a comment
```

Each message is identified by a *message number* and a *set*. Sets are often used to separate messages into more easily usable groups, such as error messages, help messages, directives, and so on. Alternatively, you could use a different set for each source file, containing all of that source file's messages.

"*$set* n" specifies the beginning of set *n*, where *n* is a set identifier in the range from 1 to NL_SETMAX. All messages following the "*$set* n" statement belong to set *n* until either a *$delset* or another *$set* is reached. You can skip set numbers (for example, you can have a set 3 without having a set 2), but the set numbers that you use must be listed in ascending numerical order (and every set must have a number). Any string following the set identifier on the same line is considered a comment.

"*$delset* n" deletes the set *n* from a message catalog.

"$quote c" specifies a quote character, *c*, which can be used to surround message text so that trailing spaces or null (empty) messages are visible in a message source line. By default, there is no quote character and messages are separated by newlines. To continue a message onto a second line, add a backslash to the end of the first line:

```
$set 1
1 Hello, world.
2 here is a long \
string.\n
3 Hello again.
n   message-text-n
```

Message #2 in set #1 is "here is a long string.\n".

Compiling XPG/3 Message Catalogs

After creating the message catalog sources, you need to compile them into binary form using *gencat*, which has the following syntax:

```
gencat catfile   msgfile [msgfile ...]
```

where *catfile* is the target message catalog and *msgfile* is the message source file. If an old *catfile* exists, *gencat* attempts to merge new entries with the old. *gencat* "resolves" set and message number conflicts with new information replacing the old.

The *catfile* then needs to be placed in a location where *catopen()* can find it; see the **"XPG/3 Catalog Location"** section of this chapter, above.

SVR4 MNLS Message Catalogs

There are many ways to use strings from MNLS message catalogs. You can get strings directly and then use them, or you can use output routines that search catalogs.

Specifying MNLS Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified every time a string is needed.

To specify the default message catalog to be used by subsequent calls to MNLS routines that reference catalogs (such as *gettxt()*, *tfmt()*, or *pfmt()*), use *setcat()*:

```
#include <pfmt.h>
char *setcat(const char *catalog);
```

where *catalog* is limited to 14 characters, with no character equal to zero or the ASCII codes for `/` (slash) or *:* (colon). *setcat()* doesn't check to see if the catalog name is valid; it just stores the string for future reference. For an example of use, see **"Getting Strings from MNLS Message Catalogs,"** below. The catalog indicated by the string can be found in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Getting Strings from MNLS Message Catalogs

MNLS message catalogs do not need to be specifically opened. The catalog of choice can be set explicitly once, or it can be specified in each reference call. Strings are read from a catalog via *gettxt()*:

```
#include <unistd.h>
char *gettxt(const char *msgid, const char *defaultStr);
```

msgid is a string containing two fields separated by a colon:

```
msgfilename:msgnumber
```

The *msgfilename* is a catalog name as described previously in the **"Specifying MNLS Catalogs"** section of this chapter. For example, to get message 10 from the *MQ* catalog, you could use either:

```
char *str = gettxt("MQ:10", "Hello, world.\n");
```

or

```
setcat("MQ");
str = gettxt(":10", "Hello, world.\n");
```

pfmt()

pfmt() is one of the most important routines dealing with MNLS catalogs, because it is used to produce most system diagnostic messages. *pfmt()* formats like *printf()* and produces standard error message formats. It can usually be used in place of *perror()*. For example,

```
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce, by default (such as when the Mozambique locale is unavailable),

```
ERROR: Permission denied.
```

The syntax of *pfmt()* is:

```
#include <pfmt.h>
int pfmt(FILE *stream, long flags, char *format, ... );
```

The *flags* are used to indicate severity, type, or control details to *pfmt()*. The format string includes information specifying which message from which catalog to look for. Flag details are discussed next in the **"Labels, Severity, and Flags"** section of this chapter, and the format is discussed in the **"Format Strings for pfmt()"** section.

Labels, Severity, and Flags

pfmt() flags are composed of several groups; specify no more than one from each group. Specify multiple flags by using OR.

The groups are:
output format control:
MM_NOSTD, MM_STD
catalog access control:
MM_NOGET, MM_GET
severity: MM_HALT, MM_ERROR, MM_WARNING, MM_INFO
action message specification:
MM_ACTION

pfmt() prints messages in the form *label:severity:text*. *Severity* is specified in the *flags*. The *text* comes from a message catalog (or a default) as specified in the *format*, and the *label* is specified earlier by the application.

In the example above, we get only:

```
ERROR: Permission denied.
```

if no label has been set. Typically, an application sets the label once early in its life; subsequent error messages have the label prepended. For example:

```
setlabel("UX:myprog");
...
pfmt(stderr, MM_ERROR, "MQ:64:Permission denied");
```

would produce (by default):

```
UX:myprog: ERROR: Permission denied.
```

For details, consult the *pfmt(*3*)* and *setlabel(*3*)* manual pages.

Format Strings for *pfmt()*

pfmt() format strings are of the form:

[[*catalog*:]*messagenum*:]*defaultstring*

The *catalog* field is in the format described above in the **"Specifying MNLS Catalogs"** section of this chapter. *messagenum* is the message number in the catalog to use as the format. *defaultstring* specifies the string to use if the catalog lookup fails for any reason.

An important feature of *pfmt()* is its ability to refer to format arguments in format−specified order just as *printf()* does. See **"Variably Ordered Referencing of printf() Arguments"** for details.

fmtmsg()

fmtmsg() is a comprehensive formatter using the MNLS catalogs and "standard" formats. You probably won't need to use it; most applications should get by with *pfmt()*, *gettxt()*, and *printf()*. Consult the *fmtmsg(*3*)* manual page for details.

Putting Strings into a Catalog

An MNLS catalog source file contains merely a list of strings, separated by new lines. For an empty string, an empty line is used. Strings are referenced by line number in the original source file.

When you add new strings to an existing catalog source, always append them to the end of the file—otherwise, other applications will no longer know where in the catalog their strings are. But if you want to modify a string already in the catalog, don't append a modified copy to the end of the file; just change the old version directly.

Tools exist that help you compile MNLS message catalogs. *exstr*, for instance, extracts strings directly from your source code and replaces them with calls to the message retrieval function.

When a file of strings is ready to be compiled, simply run *mkmsgs* and put the results in the directory */usr/lib/locale/localename/LC_MESSAGES*.

Internationalizing File Typing Rule Strings

You can internationalize the strings defined in the LEGEND and MENUCMD rules in the File Typing Rule (FTR) file. To internationalize these rules, precede the string with the following:

```
:catalogname:msgnumber:
```

where *catalogname* is optional and should be a valid MNLS catalog, *msgnumber* is the line number in *catalogname*. If you omit *catalogname*, the *uxsgidesktop* catalog is used by default.

You can use these rules to create your own FTR catalog. For example, an entry looks like this:

```
LEGEND :mycatalog:7:Archive 8mm Tape Drive
```

This entry uses line 7 from the catalog, *mycatalog*, as the LEGEND for this FTR. If *mycatalog* is not available, or line 7 is not accessible from *mycatalog*, "Archive 8mm Tape Drive" is used as the LEGEND.

```
LEGEND :7:Archive 8mm Tape Drive
```

This entry uses line 7 from the *uxsgidesktop* catalog, if available. Otherwise, "Archive 8mm Tape Drive" is used.

The next example,

```
MENUCMD \`mycatalog:9:Eject Tape\’ /usr/sbin/eject /dev/tape
```

displays line 9 from *mycatalog*, if available. Otherwise "Eject Tape" is displayed on the menu that pops up when you click on an icon using this FTR.

You can internationalize strings in the command part of MENUCMD and CMD rules by using *gettxt* or any other convenient policy detailed in this chapter. For example:

```
CMD OPEN xconfirm –t "Tape tool not available"
```

can be internationalized to:

```
CMD OPEN xconfirm –t "`gettxt mycatalog:376 ’Tape tool not available’`"
```

In this example, *gettxt* is invoked to access line 376 from the catalog, *mycatalog*, and the string returned by *gettxt* is passed to *xconfirm* for display. If line 376 from *mycatalog* is not accessible, then *gettxt* returns the string "Tape tool not available."

For more information about FTRs, see the *Indigo Magic Desktop Integration Guide*, **Chapter 5, "File Typing Rules."**

Variably Ordered Referencing of *printf()* Arguments

printf() and its variants can now refer to arguments in any specified order. Consider the following scenario: an application has chosen "house" from a list of objects and "white" from a list of colors. The application wishes to display this choice. The code might look like:

```
char *obj, *color;
... /* make choices */ ...
printf("%s %s\n", color, obj);
```

And the *printf()* yields:

```
white house
```

Even once we make sure that *obj* and *color* are localized strings, we are not quite finished. If our locale is Spanish, the *printf()* yields:

```
blanca casa
```

which is wrong; in Spanish, it should be:

```
casa blanca
```

The solution to this problem is *variably ordered referencing* of *printf()* arguments. The syntax of *printf()* format strings has been expanded.

If a format string used to contain %*T* (where *T* represents one of the *printf()* conversion characters), the string can now contain %*D*$*T*. The *T* is the same, but the *D* specifies which argument from the argument list should be used.

This means you can write:

```
printf("2nd parameter is %2$s; the 1st is %1$s", p1, p2)
```

and the *second* parameter is printed *first*, with the first parameter printed second. For example:

```
char *store = "Macy's";
char *obj = "a cup";

printf("At %1$s, I bought %2$s.\n", store, obj);
printf("I bought %2$s at %1$s.\n", store, obj);
```

yields:

```
At Macy's, I bought a cup.
I bought a cup at Macy's.
```

even though the parameters to *printf()* are in the same order.

In English, we are able to come up with strings suitable for either word order; in some other language, we might not be so lucky. Nor can we predict which order such languages might prefer. So the developer has no way of knowing how to create traditional *printf()* format strings suitable for all languages.

Developers should therefore use message catalogs for their *printf()* format strings that take linguistic parameters, and allow localizers to localize the format strings as well as text strings. This means that the localizer has much greater ability to create intelligible text. An internationalized version of the above might look like:

```
/* internationalized (XPG/3) version */
char *form = catgets(msgd, set, formNum,
                     "At %1$s, I bought %2$s.\n");
char *store = catgets(msgd, set, storeNum, "Macy's");
char *obj = catgets(msgd, set, objNum, "a cup");

printf(form, store, obj);
```

The unlocalized (default) version would produce:

```
At Macy's, I bought a cup.
```

and a localized version might produce:

```
Compré una tasa en Macy's.
```

In practice, variably ordered format strings are only found in message catalogs and not in default strings. The default string usually simply uses the parameters in the order they're given, without the new variable–order format strings.

Changes to Support Internationalization in X11R5

X11R5 internationalization support is provided on the X client side; that is, the application must take care of such support instead of relying on the X server. No server changes are necessary, and the protocol is unchanged. Full backward compatibility is preserved, so a new internationalized application can run on an old server.

X uses existing internationalization standards to do its internationalization support; there are no X-specific interfaces to set and change locale. Internationalized X applications receive no help from X when attempting multilingual support. No locales or special process states are peculiar to X.

Some Effects of X Considerations

Since X is locale-independent, there are some limitations on its ability to support internationalization. The X protocol and Xlib specification, together with ANSI C and POSIX restrictions, have led to the following choices being made in X11R5:

Vertical Text

There is no built-in support for vertical text. Applications may draw strings vertically only by laying out the text manually.

Character Sets

In previous releases of X, there was no general support for character sets other than Latin 1. X11R5, however, does allow other character sets.

X11R5 includes the definition of the *X Portable Character Set*, which is required to exist in all locales supported by Xlib. There is no encoding defined for this set; it is only a character set. The set, similar to printable ASCII, consists of these characters:

```
abcdefghijklmnoqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
<space> <tab> <newline>
```

The *Host Portable Character Encoding* is the encoding of the X Portable Character Set on the Xlib host. This encoding is part of X, and is thus independent of locale—the coding remains the same for all locales supported by the host.

Strings used or returned by Xlib routines are either in the Host Portable Character Encoding or a locale-specific encoding. The Xlib manual pages specify which encodings are used where. Some string constructs (such as *TextProperty*) contain information regarding their own encoding.

Xlib Interface Change

Full use of X11R5's internationalization features means calling some new routines supplied in the X11R5 Xlib. While all old Xlib applications work with the new Xlib, developers should change their code in places. These are described below.

Resource Names

Resource names are compiled into programs. Because of that, their encoding must be known independent of locale. Trying to add a level of indirection here results in a chicken-and-egg problem: you're always left with something compiled that can't be localized. Resource names therefore use the X Portable Character Set. The names may be anything; at least they'll mean something to the application author. (If the names were numbers, for example, they would be meaningless to everybody.)

Getting X Internationalization Started

Xlib's internationalization state, like that of *libc*, needs to be initialized.

Initialization for Xlib Programming

Initialize Xlib's internationalization state after calling *setlocale()*. Xlib is being initialized, not a server or server-specific object, so a server connection is not necessary:

```
if ( setlocale(LC_ALL, "") == NULL )
      exit_with_error();
if ( ! XSupportsLocale() )
      exit_with_other_error();
if ( XSetLocaleModifiers("") == NULL)
      give_warning();
```

Just as passing an empty string to *setlocale()* honors the user's environment, so does passing an empty string to *XSetLocaleModifiers()*.

Initialization for Toolkit Programming

setlocale() should be called as early as possible after execution begins. If the locale is set, standard toolkit initialization interfaces will perform required X initialization; no extra work is necessary.

Fontsets

In X11R5, unlike previous releases of X, a string may contain characters from more than one codeset. There are several methods for determining which codeset a given character is in; which method is appropriate depends on the locale and the encoding used.

Such multiple−codeset strings usually cannot be rendered using a single font. A *fontset* is a collection of fonts suitable for rendering all codesets represented in a locale's encoding. A fontset includes information to indicate which locale it was created in. Applications create fontsets for their own use; when a program creates a fontset, it is told which of the requested fonts are unavailable.

Example: EUC in Japanese

To render strings encoded in EUC in Japanese (see **"Asian Languages"**), an application would need fonts encoded in 8859−1, JIS X 208, and JIS X 201. The application doesn't need to know which characters in a string go with which font, since it doesn't get into locale specifics. So it creates a fontset that is made from a list of user−specified fonts (under the assumption that the localizer has provided an appropriate list). Rendering is then done using that fontset. The locale−aware rendering system chooses the appropriate fonts for each character being rendered, from the supplied list.

Specifying a Fontset

A fontset specification is just a string, enumerating XLFD names of fonts. (See *X Logical Font Description Conventions*, an MIT X Consortium standard.) This string can include wild card characters. For example, a specification of 15−point "fixed" fonts might be:

```
char *fontSetSpecString = "*fixed−medium−r−normal*150*";
```

A particular server might expand this to:

```
−jis−fixed−medium−r−normal−−16−150−75−75−c−160−jisx0208.1983−0
−sony−fixed−medium−r−normal−−16−150−75−75−c−80−iso8859−1
−sony−fixed−medium−r−normal−−16−150−75−75−c−80−jisx0201.1976−0
```

Specifying the fontset by simply enumerating the fonts is perfectly acceptable:

```
char *fontSetSpecString =
"−jis−fixed−medium−r−normal*150−75−75*jisx0208.1983−0,\
−sony−fixed−medium−r−normal*150−75−75*iso8859−1,\
−sony−fixed−medium−r−normal*150−75−75*jisx0201.1976−0";
```

A German locale would use this fontset with interest only in the ISO font; a Japanese locale might use all three; a Chinese locale would have trouble with this fontset.

The developer should specify a default fontset suitable for the default locale. Furthermore, developers should ensure that the application accepts localized fontset specifications via resources (or message catalogs) or command line options. Localizers are responsible for providing default fontset specifications suitable for their locales.

Creating a Fontset

Creating fontsets in X is simply a matter of providing a string that names the fonts, as described above.

```
XFontSet fontset;
char *base_name;  /* should get from resource */
char **missingCharsetList;
int missingCharsetCount;
char *defaultStringForMissingCharsets;

base_name = "*fixed-medium-r*150*"; /* use resources! */

fontset = XCreateFontSet(display, base_name,
                         &missingCharsetList,
                         &missingCharsetCount,
                         &defaultStringForMissingCharsets);
```

The locale in effect at create time is bound to the fontset. Fontsets are freed with *XFreeFontSet()*.

Using a Fontset

Fontsets are used when rendering text with X11R5 *Xmb* or *Xwc* text rendering routines. These routines are described in **"Text Rendering Routines,"** below.

Text Rendering Routines

X11R5 includes text rendering routines that understand multibyte and wide character strings. These routines are analogs to the X11R4 text rendering routines *XDrawText()*, *XDrawString()*, and *XDrawImageString()*. The old routines continue to operate, but do not take fontsets, and don't know how to handle characters longer than one byte.

- *XmbDrawText()* and *XwcDrawText()* take lists of *TextItems*, each of which contains (among other things) a string. The strings are rendered using fontsets. These routines allow complex spacing and fontset shifts between strings.
- *XmbDrawString()* and *XwcDrawString()* render a string using a fontset. These routines render in foreground only and use the raster operation from the current graphics context.
- *XmbDrawImageString()* and *XwcDrawImageString* also render a string using a fontset. These routines fill the background rectangle of the entire string with the background, then render the string in the foreground color, ignoring the currently active raster operation.

Consult the appropriate manual pages for more details on these routines.

New Text Extents Functions

X11R5 provides MB and WC versions of *width* and *extents* interrogation routines, supplying the maximum amount of space required to draw any character in a given fontset. These routines depend on fontsets to interpret strings and use locale–specific data.

The *XFontSetExtents* structure contains the two kinds of extents a string can have:

```
typedef struct {
      XRectangle max_ink_extent;
      XRectangle max_logical_extent;
} XFontSetExtents;
```

max_ink_extent gives the maximum boundaries needed to render the drawable characters of a fontset. It considers only the parts of glyphs that would be drawn, and gives distances relative to a constant origin. *max_logical_extent* gives the maximum extent of the *occupied space* of drawable characters of a fontset. The occupied space of a character is a rectangle specifying minimum distance from other graphical features; other graphics generated by a client should not intersect this rectangle. *max_logical_extent* is used to compute interline spacing and the minimum amount of space needed for a given number of characters.

Here are descriptions of a few of the new extents–related functions (consult the appropriate manual pages for details):

- *XExtentsOfFontSet()* returns an *XFontSetExtents* structure for a fontset.
- *XmbTextEscapement()* and *XwcTextEscapement()* take a string and return the distance in pixels (in the current drawing direction) to the origin of the next character after the string, if the string were drawn. Escapement is always positive, regardless of direction.
- *XmbTextExtents()* and *XwcTextExtents()* take a string and return information detailing the overall rectangle bounding the string's image and the space the string occupies (for spacing purposes).
- *XmbTextPerCharExtents()* and *XwcTextPerCharExtents()* take a string and return ink and logical extents for each character in the string. This can be used for redrawing portions of strings or for word justification. If the fontset might include context–dependent drawing, the client cannot assume that it can redraw individual characters and get the same rendering.
- *XContextDependentDrawing()* returns a Boolean telling whether a fontset might include context–dependent drawing.

User Input

Just as internationalized programs cannot assume that data is in ASCII, they cannot assume that user input will use any specific keyboard. Keyboards change from country to country and language to language; internationalized software should never assume that a certain position on the keyboard is bound to a certain character, or that a given character will be available as a single keystroke on all keyboards.

No useful physical keyboard—not even one specifically designed for multilingual work—could possibly contain a key for every character we would ever wish to type. Certainly there are characters commonly used in other areas of the world that are not present on most USA keyboards. So methods have been invented that provide for input of almost any known character on even the most naïve keyboards. These schemes are referred to as *input methods* (IMs).

Note: IMs should not be confused with the IRIS IM product, the Silicon Graphics port of the OSF/Motif product.

Input methods vary significantly in design, use, and behavior, but there is a single API that developers use to access them. The object is for the application simply to ask for an IM and let the system check the locale and choose the appropriate IM.

Some IMs are complex; others are very simple. The API is designed to be a low–level interface, like Xlib. Usually, only toolkit text object authors must deal with the IM interfaces. However, some applications developers are unable to use toolkit objects, so the concepts are described here.

Don't Use This Section If You Can Avoid It

This section (**"User Input"**) has to do with the translation of physical user events into programmatic character strings or special keyboard data (such as "backspace"). This kind of work should be done by toolkits. If you can use a toolkit to manage event processing for you, do so, and blissfully ignore this section. If you are writing a toolkit text object, or are writing a truly extraordinary application, then this section is for you.

Reuse Sample Code

A sample program demonstrating some of the concepts in this chapter is given in Chapter 11 of the *Xlib Programming Manual, Volume One*. Looking carefully at that code may be easier than starting from scratch.

GL Input

The old GL function *qdevice()* has a hard–coded view of a keyboard (see */usr/include/gl/device.h* for details). Some flexibility, particularly for Europe, is available if you queue KEYBD instead of individual keys, but the GL has no general solution to non–ASCII input. There is no supported way to input

Chinese (for instance) to the old GL.

The OpenGL does not contain input code but leaves that to the operating environment, which in IRIX means X.

In short, support for internationalized input means a departure from *qread()*. Under IRIX, that means using mixed−model input, all the more reason to use a toolkit.

Keys, Keycodes, and Keysyms

This section provides some background that may help make the following sections easier to understand.

X Keyboard Support: a Brief Tutorial

When a client connects to the X server, the server announces its range of *keycodes* and exports a table of *keysyms*. Each key event the client receives has a single byte *keycode*, which directly represents a physical key, and a single byte *state*, which represents currently engaged modifier keys, such as Shift or Alt.

Note: The mapping of state bits to modifiers is done by another table acquired from the server.

Keysyms are well defined, and there has been an attempt to have a keysym for every engraving one might possibly find on any keyboard, anywhere. (An *engraving* is the image imprinted on a physical key.) These are contained in */usr/include/X11/keysymdef.h*. Keysyms represent the engravings on the actual keys, but not their meanings. The server's idea of the keysym table can be changed by clients, and clients may receive *KeyMap* events when this remapping happens, but such events don't happen often.

When a client receives a Key event, it asks Xlib to use the keycode to index into its keysym table to find a list of keysyms. (This list is usually very short. Most keys have only one or two engravings on them.) Using the state byte, Xlib chooses a keysym from the list to find out what was engraved on the key the user pressed.

At this point, the client can choose to act on the keysym itself (if, for instance, it was a backspace) or it can ask for a character string represented by the keysym (or both). Generating such a string is tricky; it is discussed in **"Input Methods,"** below.

Details on X keyboard support can be found in *X Window System, Third Edition*, from Digital Press. Details on input methods are also available in that book, as well as in the *Xlib Programming Manual, Volume One*.

Composed Characters

There are two ways to compose characters that do not exist on a keyboard: explicit and implicit.

It is common for an application to be modal and switch between the two. For example, Japanese input of *kana* is often done via implicit composition. Users switch between a mode where input is interpreted as *romaji* (Latin characters) and a mode where strokes are all translated to *kana*.

Furthermore, both styles may operate simultaneously. While an application is supporting implicit composition of certain characters, other characters may be composable via explicit composition.

So remember that not every keystroke produces a character, even if the associated keysym would normally imply character text. The event−to−string translation routines (see, for example, **"XLookupString(), XwcLookupString(), and XmbLookupString()"** in this chapter) figure out what result a given set of keystrokes should produce.

Explicit Composition

Explicit composition is requested when the user presses the Compose key and then types a key sequence that corresponds to the desired character. For example, to compose the character 'ñ' under some keymaps, you might press the Compose key and then type "~n".

Note: The Compose key can be defined by using *xmodmap*(1) to map the XK_Multi_key keysym onto whatever key you want to use as Compose.

Implicit Composition

Implicit composition mimics many existing European typewriters that have "dead" keys: keys that type a character but do not advance the carriage. When a special "dead" key is struck, the system attempts to compose a character using the next character struck. For example, on a keyboard that had a diaeresis (¨) and an O, but no Ö, one would simply strike '¨' and then 'O' to compose 'Ö'.

Implicit composition support usually comes with some specified way to leave characters uncomposed.

Keyboards

IRIX currently supports 12 keyboard layouts: American, Belgian, Danish, English, French, German, Italian, Norwegian, Portuguese, Spanish, Swedish, and Swiss.

All are representable in Latin 1; the American keyboard needs only ASCII.

Input Methods

Input methods (IMs) are ways to translate keyboard−input events into text strings. You would use a different input method, for instance, to type on a USA keyboard in Chinese than to type on the same keyboard in English. Nobody would build a keyboard suitable for direct input of the roughly 80,000 distinct Chinese characters.

IMs come in two flavors, *front−end* and *back−end*. Both types can use identical application programming interfaces, so we lose no generality by using back−end methods for our examples here.

To use an IM, follow these steps:

1. Open the IM.
2. Find out what the IM can do.
3. Agree upon capabilities to use.
4. Create input contexts with preferences and window(s) specified (see **"Input Contexts (ICs),"** later in this chapter).
5. Set the input context focus.
6. Process events.

Although all applications go through the same setup when establishing input methods, the results can vary widely. In a Japanese locale, one might end up with networked communications with an input method and a *kanji* translation server, with circuitous paths for Key events. But in (say) a Swiss locale, it is likely that nothing would occur besides a flag or two being set in Xlib. Since operating in non−Asian locales ends up bypassing almost all of the things that might make input methods expensive, Western users are not noticeably penalized for using Asia−ready applications.

Opening an Input Method

XOpenIM() opens an input method appropriate for the locale and modifiers in effect when it is called. The locale is bound to that IM and cannot be changed. (But you could open another IM if you wanted to switch later.) Strings returned by *XmbLookupString()* and *XwcLookupString()* are encoded in the locale that was current when the IM was opened, regardless of current input context.

The syntax is:

```
XIM XOpenIM(Display *dpy, XrmDataBase db, char *res_name,
              char *res_class);
```

where *res_name* is the resource name of the application, *res_class* is the resource class, and *db* is the resource database that the input method should use for looking up resources private to itself. Any of these can be NULL.

So opening an IM is easy:

```
XIM im;
```

```
im = XOpenIM(dpy, NULL, NULL, NULL);
if (im == NULL)
      exit_with_error();
```

XOpenIM() finds the IM appropriate for the current locale. If *XSupportsLocale()* has returned good status (see the **"Initialization for Xlib Programming"** section of this chapter) and *XOpenIM()* fails, something is amiss with the administration of the system.

Note: The library routines are not prepared for the possibility of *XSupportsLocale()* succeeding and *XOpenIM()* failing, so it's up to application developers to deal with such an eventuality. (This circumstance could occur, for example, if the IM died after *XSupportsLocale()* was called.) This topic is under some debate in the MIT X consortium.

Most of the complexity associated with IM use comes from configuring an input context to work with the IM. Input contexts are discussed in **"Input Contexts (ICs),"** elsewhere in this chapter.

To close an input method, call *XCloseIM()*.

IM Styles

If the application requests it, an input method can often supply *status* information about itself. For example, a Japanese IM may be able to indicate whether it is in Japanese input mode or *romaji* input mode. An input method can also supply *pre−edit* information, partial feedback about characters in the process of being composed. The way in which an IM deals with status and pre−edit information is referred to as an IM *style*. This section describes styles and their naming.

Root Window

The *Root Window* style has a pre−edit area and a status area in a window owned by the IM as a descendant of the root. The application does not manage the pre−edit data, the pre−edit area, the status data, or the status area. Everything is left to the input method to do in its own window, as illustrated in **Figure 9−1** :

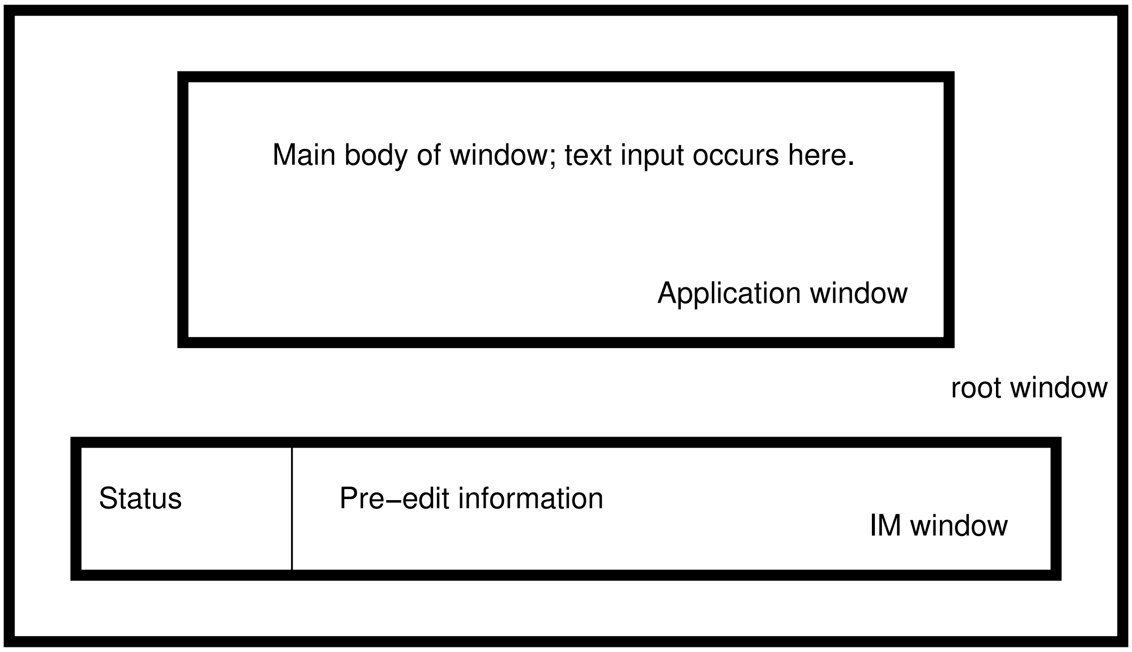

Figure 9−1 Root Window Input

Off−the−Spot

The *Off−the−Spot* style places a pre−edit area and a status area in the window being used, usually in reserved space away from the place where input appears. The application manages the pre−edit area and status area, but allows the IM to update the data there. (The application provides information regarding foreground and background colors, fonts, and so on.) A window using Off−the−Spot input style might look like that shown in **Figure 9−2** :

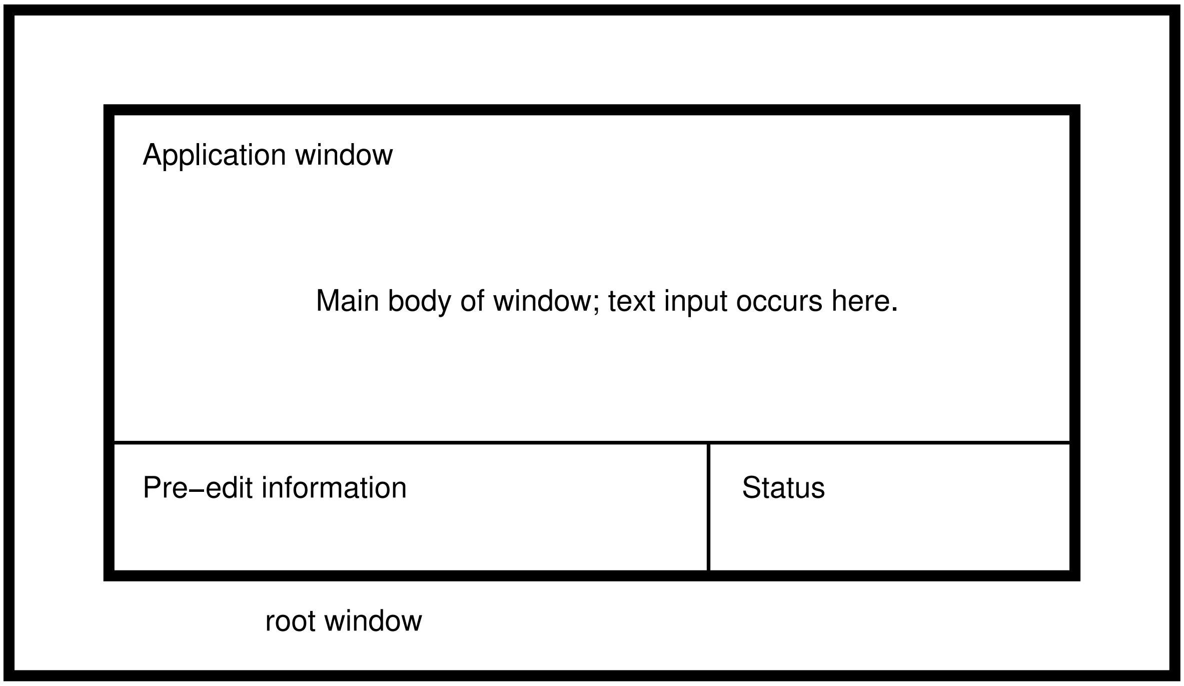

Figure 9−2 Off−the−Spot Input

Over−the−Spot

The *Over−the−Spot* style involves the IM creating a small, pre−edit window over the point of insertion. The window is owned and managed by the IM as a descendant of the root, but it gives the user the impression that input is being entered in the right place; in fact, the pre−edit window often has no borders and is invisible to the user, giving the appearance of On−the−Spot input. The application manages the status area as in Off−the−Spot, but specifies the location of the editing so that the IM can place pre−edit data over that spot.

On−the−Spot

On−the−Spot input is by far the most complex for the application developer. The IM delivers all pre−edit data via callbacks to the application, which must perform in−place editing—complete with insertion and deletion and so on. This approach usually involves a great deal of string and text rendering support at the input generation level, above and beyond the effort required for completed input. Since this may mean a lot of updating of surrounding data or other display management, everything is left to the application. There is little chance an IM could ever know enough about the application to be able to help it provide user feedback. The IM therefore provides status and edit information via callbacks.

Done well, this style can be the most intuitive one for a user.

Setting IM Styles

A style describes how an IM presents its pre−edit and status information to the user. An IM supplies information detailing its presentation capabilities. The information comes in the form of flags, OR’ed together. The flags to use with each style are:

Root Window	XIMPreeditNothing | XIMStatusNothing
Off−the−Spot	XIMPreeditArea | XIMStatusArea
Over−the−Spot	XIMPreeditPosition | XIMStatusArea

On−the−Spot	XIMPreeditCallbacks | XIMStatusCallbacks

For example, if you wanted a style variable to match an Over−the−Spot IM style, you could write:

```
XIMStyle over = XIMPreeditPosition | XIMStatusArea;
```

If an IM returns *XIMStatusNone* (not to be confused with *XIMStatusNothing*), it means the IM will not supply status information.

Using Styles

An input method supports one or more styles. It is up to the application to find a style that is supported by both the IM and the application. If there are several, the application must choose. If there are none, the application is in trouble.

Input Contexts (ICs)

An input method may be serving multiple clients, or one client with multiple windows, or one client with multiple input styles on one window. The specification of style and client/IM communication is done via *input contexts*. An input context is simply a collection of parameters that together describe how to go about receiving and examining input under a given set of circumstances.

To set up and use an input context:

1. Decide what styles your application can support.
2. Query the IM to find out what styles it supports.
3. Find a match.
4. Determine information that the IC needs in order to work with your application.
5. Create the IC.
6. Employ the IC.

Find an IM Style

The IM may be able to support multiple styles—for example, both Off−the−Spot and Root Window. The application may be able to do, in order of preference, Over−the−Spot, Off−the−Spot, and Root Window. The application should determine that the best match in this case is Off−the−Spot.

First, discover what the IM can do, then set up a variable describing what the application can do:

```
XIMStyles *IMcando;
XIMStyle  clientCanDo; /* note type difference */
XIMStyle  styleWeWillUse = NULL;

XGetImValues(im, XNQueryInputStyle, &IMcando, NULL);

clientCanDo =
/*none*/ XIMPreeditNone | XIMStatusNone |
/*over*/ XIMPreeditPosition | XIMStatusArea |
/*off*/  XIMPreeditArea | XIMStatusArea |
/*root*/ XIMPreeditNothing | XIMStatusNothing;
```

A client should always be able to handle the case of *XIMPreeditNone | XIMStatusNone*, which is likely in a Western locale. To the application, this is not very different from a *RootWindow* style, but it comes with less overhead.

Once we know what the application can handle, we look through the IM styles for a match:

```
for(i=0; i<IMcando->count_styles; i++) {
      XIMStyle tmpStyle;
      tmpStyle = IMcando->support_styles[i];
```

```
      if ( ((tmpStyle & clientCanDo) == tmpStyle) &&
          prefer(tmpStyle, styleWeWillUse) )
              styleWeWillUse = tmpStyle;
}

if (styleWeWillUse = NULL)
      exit_with_error();
XFree(IMcando);

/* styleWeWillUse is set, which is what we were after */
```

The *prefer()* routine simply applies some heuristic to application preference.

IC Values

There are several pieces of information an input method may require, depending on the input context and style chosen by the application. The input method can acquire any such information it needs from the input context, ignoring any information that does not affect the style or IM.

A full description of every item of information available to the IM is supplied in *X Window System, Third Edition*. We include only a brief list here:

- *XNClientWindow* specifies to the IM which client window it can display data in or create child windows in. It is set once and cannot be changed.
- *XNFilterEvents* is an additional event mask for event selection on the client window.
- *XNFocusWindow* specifies the focus window. This specifies which window gets the processed (composed) Key events.
- *XNGeometryCallback* specifies a geometry handler that is called if the client allows an IM to change the geometry of the window.
- *XNInputStyle* specifies the style for this IC.
- *XNResourceClass* and *XNResourceName* specify the resource class and name to use when the IM looks up resources that vary by IC.
- *XNStatusAttributes* and *XNPreeditAttributes* specify to an IM the attributes to be used for any status and pre−edit areas. The attributes are nested, variable−length lists.

Pre−edit and Status Attributes

When an IM is going to provide state, it needs some simple X information with which to do its work. For example, if an IM is going to draw status information in a client window in an Off−the−Spot style, it needs to know where the area is, what color and font to render text in, and so on. The application gives this data to the IC for use by the IM.

As with the **"IC Values"** section, full details are available in *X Window System, Third Edition*.

- *XNArea* specifies a rectangle to be used as a status or pre−edit area.
- *XNAreaNeeded* specifies the rectangle desired by the attribute writer. Either the application or the IM may provide this information, depending on circumstances.
- *XNBackgroundPixmap* specifies a pixmap to be used for the background of windows the IM creates.
- *XNColormap* specifies the colormap to use.
- *XNCursor* specifies the cursor to use.
- *XNFontSet* specifies the fontset to use for rendering text.
- *XNForeground* and *XNBackground* specify the colors to use for rendering.
- *XNLineSpacing* specifies the line spacing to be used in the pre−edit window if more than one line is used.
- *XNSpotLocation* specifies where the next insertion point is, for use by *XIMPreeditPosition* styles.
- *XNStdColormap* specifies that the IM should use *XGetRGBColormaps()* with the supplied property (passed as an Atom) in order to find out which colormap to use.

Creating an Input Context

Creating an input context is a simple matter of calling *XCreateIC()* with a variable−length list of parameters specifying IC values. A simple example:

```
XVaNestedList arglist;
```

```
XIC ic;

arglist = XVaCreateNestedList(0, XNFontSet, fontset,
                             XNForeground,
                             WhitePixel(dpy, screen),
                             XNBackground,
                             BlackPixel(dpy, screen),
                             NULL);

ic = XCreateIC(im, XNInputStyle, styleWeWillUse,
               XNClientWindow, window, XNFocusWindow, window,
               XNStatusAttributes, arglist,
               XNPreeditAttributes, arglist, NULL);
XFree(arglist);

if (ic == NULL)
      exit_with_error();
```

Using the IC

A multi–window application may choose to use several input contexts. But for simplicity, we assume that the application just wants to get to the internationalized input using one method in one window.

Using the IC is a matter of (a) making sure we check events the IC wants and (b) setting IC focus. If you are setting up a window for the first time, you know the event mask you want, and you can use it directly. If you are attaching an IC to a previously configured window, you should query the window and add in the new event mask.

```
unsigned long imEventMask;

XGetWindowAttributes(dpy, win, &winAtts);
XGetICValues(ic, XNFilterEvents, &imEventMask, NULL);

imEventMask |= winAtts.your_event_mask;
XSelectInput(dpy, window, imEventMask);
XSetICFocus(ic);
```

At this point, the window is ready to be used.

Events Under IM Control

Processing events under input method control is almost the same in X11R5 as it was under R4 and before. There are two essential differences: *XFilterEvent()* and the *LookupString* routines.

XFilterEvent()

Every event received by your application should be fed to the IM via *XFilterEvent()*, which returns a value telling you whether or not to disregard the event. IMs asks you to disregard the event if they have extracted the data and plan on giving it to you later, possibly in some other form. All events (not just *KeyPress* and *KeyRelease* events) go to *XFilterEvent()*.

If we compacted the event processing into a single routine, a typical event loop would look something like this:

```
Xevent event;

while (TRUE) {
      XNextEvent(dpy, &event);

      if (XFilterEvent(&event, None))
          continue;
```

```
      DealWithEvent(&event);
}
```

XLookupString(), *XwcLookupString()*, and *XmbLookupString()*

When using an input method, you should replace calls to *XLookupString()* with calls to *XwcLookupString()* or *XmbLookupString()*. The *MB* and *WC* versions have very similar interfaces. In the examples below, we arbitrarily use *XmbLookupString()*, but the examples apply to both versions.

There are two new situations to deal with:

1. The string returned may be long.
2. There may be an interesting keysym returned, an interesting set of characters returned, both, or neither.

Dealing with the former is merely a matter of maintaining an arena, as in the example below.

To tell the application what to pay attention to for a given event, *XmbLookupString()* returns a status value in a passed parameter, equal to one of the following:

- *XLookupKeysym*, which indicates the keysym should be checked
- *XLookupChars*, which indicates a string has been typed or composed
- *XLookupBoth*, which means both of the above
- *XLookupNone*, which means neither is ready for processing
- *XBufferOverflow*, which means the supplied buffer is too small—*XmbLookupString()* should be called again with a bigger buffer

XmbLookupString() also returns the length of the string in question. Note that *XmbLookupString()* returns the length of the string in bytes, while *XwcLookupString()* returns the length of the string in characters.

The example below should help show how these functions work. Most event processors perform a *switch()* on the event type; assume we have done that, and we have received a *KeyPress* event:

```
case KeyPress:
{
      Keysym keysym;
      Status status;
      int buflength;
      static int bufsize = 16;
      static char *buf = NULL;

      if (buf == NULL) {
          buf = malloc(bufsize);
          if (buf < 0) StopSequence();
      }

      buflength = XmbLookupString(ic, &event, buf, bufsize,
                                 &keysym, &status);

      /* first, check to see if that worked */
      if (status == XBufferOverflow) {
          buf = realloc(buf, (bufsize = buflength));
          buflength = XmbLookupString(ic, &event, buf, bufsize,
                                     &keysym, &status);
      }

      /* We have a valid status. Check that */
      switch(status) {
      case XLookupKeysym:
      case XLookupBoth:
          DealWithKeysym(keysym);
          if (status == XLookupKeysym)
```

```
            break; /* wasn’t XLookupBoth */
        case XLookupChars:
            DealWithString(buf, buflength);
        case XLookupNone:
            break;
        } /* end switch(status) */

} /* end case KeyPress segment */
break; /* we are in a switch(event.type) statement */
```

GUI Concerns

It shouldn’t be significantly more difficult to internationalize an application with a graphical user interface than an application without such an interface, but there are a few further issues that must be addressed.

X Resources for Strings

Resource lookup mechanisms in Xlib as well as in toolkits monitor locale environment variables when locating resource files. For string constants that are used within toolkit objects, resources provide a simpler solution than do message catalogs.

Some common objects that should definitely get their labels from resources:

- Labels
- Buttons
- Menu items
- Dialog notices and questions

Any object that employs some sort of text label should be labeled via resources. Since the localizer wants to provide strings for the local version of the application, the app–defaults file for the application should specify every reasonable string resource. Manual pages should identify all localizable string resources.

Localizers of an application provide a separate resource file for each locale that the application runs in.

Layout

Layout management is of special interest when you cannot predict how large a button or other label might be. The nature of the problem of layout composition and management does not change, but one must construct the layout management without full knowledge of the final appearance.

It’s worth noting that localization efforts can be assumed to be "reasonable" in some sense. For example, X resources have always allowed a user to specify an extremely large font for buttons, but applications correctly choose to let such users live with the results. But it’s not always that clear what is reasonable and what isn’t; we don’t always know what will be difficult to translate succinctly in some locale. So while developers need not provide for all combinations of resource specifications, they must take the responsibility to make the application localizable.

There are three main approaches to the layout problem.

Dynamic Layout

Most toolkits provide *form*, *pane*, *rowcolumn*, *rubberboard*, or other layout objects that will calculate layout depending on the "natural" (localized) size of the objects involved. Most use some hints provided by the developer that can regulate this layout. For example, some IRIS IM widgets providing these services are *XmForm*, *XmPanedWindow*, and *XmRowColumn*.

Dynamic layout is probably the simplest way to prevent localization difficulties.

Note: The IRIS IM product is the Silicon Graphics port of the OSF/Motif product, and should not be confused with IM, the abbreviation for Input Methods.

Constant Layout

Under certain circumstances, an application may insist on having a predefined layout. When this is so, the application must provide objects that are so constructed as to allow localization. A "Quit" button that just barely allows room for the Latin 1 string "Quit" is not likely to suffice when localizers attempt to fit their translations into that small space.

In order to enforce constant layout, the developer incurs the heavy responsibility of making sure the objects are localizable. This means a lot of investigation; the "there, that ought to be enough" approach is chancy at best.

Localized Layout

Some toolkits provide for layout control by run−time reading of strings or other data files. Applications that use such toolkits can easily finesse the layout issue by providing capability for localization of the layout as well as the contents of the layout. This provides each localizer maximum freedom in presenting the application to the local users. The application developer is responsible for providing localizers with instructions and mechanisms necessary to produce layout data.

IRIS IM Localization with *editres*

IRIX provides an interactive method of laying out widgets for IRIS IM and Xaw (the Athena Widget Set): a utility called *editres*. With *editres*, you can construct and edit resources and see how your widgets will look on the screen; the program even generates a usable app−defaults file for you. But note that if you hard−code any resources into your IRIS IM code, you won't be able to edit them using this method.

Icons

Icons attempt to be fairly generic representations of their antecedents. Unfortunately, it is very difficult for a designer to know what is generic or recognizable in other cultures. Therefore, it is important that any pictographic representations used by an application be localizable.

Graphic representations can be stored as strings representing X bitmaps, as names of data files containing pictographs, or in whatever manner the developer thinks best, so long as the developer provides a way for the localizer to produce and deliver localized pictographs.

Some Popular Encodings

The ISO 8859 Family

American English is easily representable in 7−bit ASCII. Most other languages are not. For example, the character é is not in ASCII.

Most Western European languages are representable in 8−bit ISO 8859−1, which is commonly known as Latin 1. Latin 1 is a superset of ASCII including characters used by several Western European languages (such as ö, £, ñ, ç, ¿).

ISO 8859 comes in nine parts, many of which overlap;, all are supersets of ASCII.

The ISO 8859 Character Sets are shown in **Table 9−4** :

Table 9−4 ISO 8859 Character Sets

Character set	Common name	Languages supported
8859−1	Latin 1	Danish, Dutch, English, Faeroese, Finnish, French, German, Icelandic, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish
8859−2	Latin 2	Albanian, Czech, English, German, Hungarian, Polish,

		Rumanian, Serbo−Croatian, Slovak, Slovene
8859−3	Latin 3	Afrikaans, Catalan, Dutch, English, Esperanto, German, Italian, Maltese, Spanish, Turkish
8859−4	Latin 4	Danish, English, Estonian, Finnish, German, Greenlandic, Lapp, Latvian, Lithuanian, Norwegian, Swedish
8859−5	Latin/Cyrillic	Bulgarian, Byelorussian, English, Macedonian, Russian, Serbo−Croatian, Ukrainian
8859−6	Latin/Arabic	Arabic, English (see ISO 8859−6 specification)
8859−7	Latin/Greek	English, Greek (see ISO 8859−7 specification)
8859−8	Latin/Hebrew	English, Hebrew (see ISO 8859−8 specification)
8859−9	Latin 5	Danish, Dutch, English, Finnish, French, German, Irish, Italian, Norwegian, Portuguese, Spanish, Swedish, Turkish

IRIX contains over 500 Latin 1 fonts, as well as a few fonts for each of the other 8859−encoded character sets except 8859−6. Currently, IRIX contains no fonts for use with the 8859−6 character set.

Asian Languages

Asian languages are commonly pictographic and employ many thousands of characters for their representation. For example, Japanese and Korean can be practically encoded in 16 bits. Daily−use Chinese can also be, but archives and scholars frequently need more; Chinese is often encoded with up to four bytes per character.

Some Standards

Various Asian character sets have been developed, some of which are considered standard. Encodings for these sets are less standardized. Asian character sets usually require larger−than−byte character types like those described in **"Multiple−Byte Character Types."** **Table 9−5** lists some of these standard character sets. Note that some of these character sets have multiple associated codesets, usually designated by appending the year the codeset was adopted to the character set name. (For example, JIS X 208−1983 is different from JIS X 208−1990.)

Table 9−5 Character Sets for Asian Languages

Language	Character Set Standards	Support
Japanese	JIS X 0201	*Katakana*
	JIS X 0208	*Kanji*, *kana*, Latin, Greek, Cyrillic, symbols, others
	JIS X 0212	Supplemental *kanji*, others
Chinese	no specific standard yet	
Korean	KSC5601.1987−0	Hangul

EUC

EUC is *Extended Unix Code*, an encoding methodology that supports concurrent use of four codesets in one encoding. It employs two special "shift state" bytes:

```
ss1 = 0x8e
ss2 = 0x8f
```

These are used to identify codesets within a string. The EUC encoding scheme uses the following patterns to indicate which codeset is in use at any given time:

```
Codeset #1: 0xxxxxxx
Codeset #2: 1xxxxxxx [ 1xxxxxxx ...]
Codeset #3: ss1 1xxxxxxx [ 1xxxxxxx ...]
Codeset #4: ss2 1xxxxxxx [ 1xxxxxxx ...]
```

So if *ss1* appears in a string, it means that the next character—however many bytes long it is—should be interpreted as a character from codeset #3. (So if there are multiple characters in a row from codeset #3, each one is preceded by *ss1*.) Similarly, *ss2* indicates that the following character belongs to codeset #4. If any other byte whose high bit is 1 appears in the string (without being preceded by *ss1* or *ss2*), it is interpreted as part of a character from codeset #2.

In EUC, codeset #1 is always ASCII. The other codesets are implementation– or user–defined.

EUC implementations exist (but are not standardized) for all ideographic Asian languages.

Japanese

The codesets for the JIS character sets all use 7–bit codes. This allows Japanese to be sent in Internet messages without modification of the messages or the transport. Plain JIS requires imbedded escape sequences in order to switch between one– and two–byte character modes and character sets.

Shift–JIS (SJIS) and *UJIS* (a Japanese EUC encoding) use a string's numeric values to switch between codesets. Both encodings use the eighth bit of bytes.

Most Unix workstations that support Japanese use UJIS for internal code. UJIS codeset #2 is JIS X 208–1990, a 2–byte codeset. Codeset #3 is JIS X 201–1989, a one–byte codeset; since the *ss1* byte is needed for each character to indicate codeset #3, each character becomes 2 bytes. Codeset #4 is not used. Thus the largest characters in UJIS are 2 bytes long.

In Japan, most PCs use SJIS, which turns on the high bit of 7–bit JIS characters to shift them above ASCII, becoming compatible with ASCII at the expense of using the eighth bit of bytes.

The Japanese encodings are interconvertible. Since JIS uses a 7–bit code, SJIS and UJIS are commonly converted to JIS for network transmission.

ISO 10646 and Unicode

ISO and the Unicode Consortium have jointly developed a character set designed to cover almost every character normally used by any language in the world. ISO calls this *ISO IS 10646*. The Unicode Consortium embraces a subset of 10646, called the *Basic Multilingual Plane* (BMP) of 10646, and calls it *Unicode*. The only characters defined in either standard are the characters in the BMP. The characters have 2– and 4–byte representations.

It appears that ISO 10646 will grow significantly in acceptance, but widespread use is still some years away.

Appendix A

Position–Independent Coding in Assembly Language

Several new assembler directives have been added to support generation of PIC. You should also make yourself familiar with the MIPS ABI Supplement and the PIC coding model it describes.

The assembler generates PIC if either of two things occur:

- the directive *.option pic2* appears in the assembler file
- the assembler, *as*, is invoked with the *-KPIC* argument in the absence of an explicit *.option pic0* directive in the file

Unless PIC is being generated, the other options in this section are ignored by the assembler, with the exception of *.gpword*, which becomes *.word*. Thus, you may easily use the same assembler file for generating PIC and non−PIC (i.e., non−shared) objects by *not* placing *.option pic0* or *.option pic2* in the assembler file and invoking the assembler without *-KPIC* (for non−shared) or with *-KPIC* (for PIC).

- *.option pic2*

  This directive forces the assembler to mark the output object file "PIC" and activates the following directives. It overrides the command line argument. Normally, you don't need to specify this directive. Instead, you should use the *-KPIC* or *-non_shared* options to toggle between generating PIC or non−PIC.

  Note that even though *-KPIC* will be made the default for the high−level language drivers (such as *cc*, *f77*, and *pc*) in future releases, it will *not* be the default for assembly sources. You must explicitly specify *-KPIC* for compiling .s files.

- *.cpload reg*

  This directive expands into three instructions that sets the gp register to the context pointer value for the current function. The three instructions are:

  ```
  lui     gp,_gp_disp
  addui   gp,gp,_gp_disp
  addu    gp,gp,reg
  ```

  _gp_disp is a reserved symbol that the linker sets to the distance between the lui instruction and the context pointer. This directive is required at the beginning of each subroutine that uses the *gp* register.

  You must add this directive at the beginning of every procedure, with the exception of leaf−procedures that do not access any global variables, and procedures that are static (i.e., not marked *.globl* or *.extern*).

- *.cprestore offset*

  This directive causes the assembler to issue

  ```
  sw     gp,offset(sp)
  ```

  at the point where it appears. Additionally, it causes the assembler to emit

  ```
  lw     gp,offset(sp)
  ```

  after every jump−and−link (jal) operation (but *not* after a branch−and−link (bal) operation), thereby restoring the *gp* register after function calls. The programmer is responsible for allocating the stack space for the *gp*. This space should be in the saved register area of the stack frame to remain consistent with MIPS' calling and debugger conventions.

- *.gpword local−sym*

  <Default Para Font>This directive is similar to .word except that the relocation entry for local−sym has the R_MIPS_GPREL32 type. After linkage, this results in a 32−bit value that is the distance between local−sym and the context pointer (that is, the gp). local−sym must be local. It is currently used for PIC switch tables.

- *.cpadd reg*

  This adds the value of the context pointer (*gp*) to reg.

Examples

This following is a simplified version of the "hello world" program:

```
          .option pic2
          .data
          .align      2
$$5:
          .ascii      "hello world\X0A\X00"
          .text
          .align      2
main:
```

```
.set        noreorder
.cpload    $25
.set       reorder
subu       $sp, 40
sw         $31, 36($sp)
.cprestore 32
la         $4, $$5
jal        printf
move       $2, $0
lw         $31, 36($sp)
addu       $sp, 40
j          $31
```

The actual instructions generated by the assembler are:

```
lui     gp,0            #
addiu   gp,gp,0         # generated by .cpload
addu    gp,gp,t9        #
lw      a0,0(gp)        # gp-relative addressing used
lw      t9,0(gp)        # t9 is used for func. call
addiu   sp,sp,-40
sw      ra,36(sp)
sw      gp,32(sp)       # from .cprestore
jalr    ra,t9           # jal is changed to jalr
addiu   a0,a0,0
lw      ra,36(sp)
lw      gp,32(sp)       # activated by .cprestore
move    v0,zero
jr      ra
addiu   sp,sp,40
nop
```

Note: The MIPS ABI requires that register t9 ($25) be used for indirect function call, so .cpload should always use $25. No reorder mode should also be used. Also, programmers should make sure that t9 is dead before any function call.

If your program uses an indirect jump (*jalr*), you must also use *t9* as the jump register.

If you have an unconditional jump to an external label:

```
j    _cerror
```

you have to rewrite it into indirect jump via *t9*:

```
la   t9,_cerror
j    t9
```

If you use branch–and–link (*bal*) instruction, and if the target procedure begins with a *.cpload*, you have to specify an alternate entry point:

```
foo: .set     noreorder # callee
       .cpload  $25
       .set     reorder
$$1:          ...       # alternate entry point
       ...
       j        $31       # foo returns
bar:          ...       # caller
       ...
bal  $$1                # by-pass the .cpload
       ...
```

This is very important because *.cpload* assumes register *$25* contains the address of foo, but in this case *$25* is not set up. Note that since both *foo* and *bar* reside in the same file, they must have the same value for *$gp*. So the *.cpload* instructions can be and must be bypassed. However, since *foo* can still be called from outside, the *.cpload* is still required.

Alternatively, if you don't want to have an alternate entry point, you can set up register *$25* before the *bal* :

```
        la      t9,foo
        bal     foo
```

but this is less efficient.

.gpword and *.cpadd* are used together to implement position−independent jump table (or any table of text addresses). Entries of the address table created by .gpword are converted into displacement from the context pointer. To get the correct text address, *.cpadd* should be used to add the value of *gp* back to them. Since the *gp* is updated by the run−time linker, the correct text address can be reconstructed regardless of the location of the DSO.

Appendix B

ISO 3166 Country Names and Abbreviations

Table B−1 lists the ISO 3166 country codes, alphabetized by country name (the table reads from left to right, and top to bottom).

Table B−1 ISO 3166 Country Codes

Country Name	Code	Country Name	Code	Country Name
Afghanistan	AF	Albania	AL	Algeria
American Samoa	AS	Andorra	AD	Angola
Anguilla	AI	Antarctica	AQ	Antigua and Barbuda
Argentina	AR	Aruba	AW	Australia
Austria	AT	Bahamas	BS	Bahrain
Bangladesh	BD	Barbados	BB	Belgium
Belize	BZ	Benin	BJ	Bermuda
Bhutan	BT	Bolivia	BO	Botswana
Bouvet Island	BV	Brazil	BR	British Indian Ocean Territory
Brunei Darussalam	BN	Bulgaria	BG	Burkina Faso
Burma	BU	Burundi	BI	Byelorussia
Cameroon	CM	Canada	CA	Cape Verde
Cayman Islands	KY	Central African Republic	CF	Chad
Chile	CL	China	CN	Christmas Island
Cocos Islands	CC	Colombia	CO	Comoros
Congo	CG	Cook Islands	CK	Costa Rica
Cote D'Ivoire	CI	Cuba	CU	Cyprus
Czech Republic	CS	Denmark	DK	Djibouti
Dominica	DM	Dominican Republic	DO	East Timor
Ecuador	EC	Egypt	EG	El Salvador
Equatorial Guinea	GQ	Ethiopia	ET	Falkland Islands
Faroe Islands	FO	Fiji	FJ	Finland
France	FR	French Guiana	GF	French Polynesia
French Southern Territories	TF	Gabon	GA	Gambia
Germany	DE	Ghana	GH	Gibraltar
Greece	GR	Greenland	GL	Grenada
Guadalupe	GP	Guam	GU	Guatemala
Guinea−Bissau	GW	Guinea	GN	Guyana
Haiti	HT	Heard and McDonald Islands	HM	Honduras

Hong Kong	HK	Hungary	HU	Iceland
India	IN	Indonesia	ID	Iran
Iraq	IQ	Ireland	IE	Israel
Italy	IT	Jamaica	JM	Japan
Jordan	JO	Kampuchea	KH	Kenya
Kiribati	KI	Korea	KP or KR	Kuwait
Laos	LA	Lebanon	LB	Lesotho
Liberia	LR	Libya	LY	Liechtenstein
Luxembourg	LU	Macau	MO	Madagascar
Malawi	MW	Malaysia	MY	Maldives
Mali	ML	Malta	MT	Marshall Islands
Martinique	MQ	Mauritania	MR	Mauritius
Mexico	MX	Micronesia	FM	Monaco
Mongolia	MN	Montserrat	MS	Morocco
Mozambique	MZ	Namibia	NA	Nauru
Nepal	NP	Netherlands Antilles	AN	Netherlands
Neutral Zone	NT	New Caledonia	NC	New Zealand
Nicaragua	NI	Nigeria	NG	Niger
Niue	NU	Norfolk Island	NF	Northern Mariana Islands
Norway	NO	Oman	OM	Pakistan
Palau	PW	Panama	PA	Pangaea
Papua New Guinea	PG	Paraguay	PY	Peru
Philippines	PH	Pitcairn	PN	Poland
Portugal	PT	Puerto Rico	PR	Qatar
Quebec	QC	Reunion	RE	Romania
Rwanda	RW	Saint Kitts and Nevis	KN	Saint Lucia
Saint Vincent and the Granadines	VC	Samoa	WS	San Marino
Sao Tome and Principe	ST	Saudi Arabia	SA	Senegal
Seychelles	SC	Sierra Leone	SL	Singapore
Solomon Islands	SB	Somalia	SO	South Africa
Spain	ES	Sri Lanka	LK	St. Helena
St. Pierre and Miquelon	PM	Sudan	SD	Suriname
Svalbard and Jan Mayen Islands	SJ	Swaziland	SZ	Sweden
Switzerland	CH	Syrian Arab Republic	SY	Taiwan
Tanzania	TZ	Thailand	TH	Togo
Tokelau	TK	Tonga	TO	Trinidad and Tobago
Tunisia	TN	Turkey	TR	Turks and Caicos Islands
Tuvalu	TV	Uganda	UG	Ukraine
United Arab Emirates	AE	United Kingdom	GB	United States Minor Outlying Islands
Uruguay	UY	Vanuatu	VU	Vatican City State
Venezuela	VE	Viet Nam	VN	Virgin Islands (British)
Virgin Islands (USA)	VI	Wallis and Futuna Islands	WF	Western Sahara
Yemen	YE or YD	Yugoslavia (Former)	YU	Zaire
Zambia	ZM	Zimbabwe	ZW	