POSIX 1.b Migration Guide

LynxOS Release 4

DOC-0416-00

Contents

Preface

The *POSIX 1.b Migration Guide* is intended to help developers to migrate code developed under POSIX.4 Draft 9 to POSIX.4 Draft 14 (POSIX.1b); it does not provide enough information to write code directly to the POSIX.1b standard. This Guide also assumes that the reader is thoroughly familiar with POSIX.4 Draft 9. For detailed information, consult the appropriate LynxOS man pages and the POSIX.1b standard.

For More Information

For more information on the features of LynxOS, refer to the following printed and online documentation.

- *Release Notes*

  This printed document contains late-breaking information about the current release.

- *LynxOS Installation Guide*

  This manual supports the initial installation and configuration of LynxOS and the X Windows System.

- *LynxOS User's Guide*

  This document contains information about basic system administration and kernel level specifics of LynxOS. It contains a "Quick Starting" chapter and covers a range of topics, including tuning system performance and creating kernel images for embedded applications.

- Online information

  Information about commands and utilities is provided online in text format through the **man** command. For example, a user wanting information about the GNU compiler would enter the following syntax, where **gcc** is the argument for information about the GNU compiler:

```
man gcc
```

More recent versions of the documentation listed here may also be found online.

Typographical Conventions

The typefaces used in this manual, summarized below, emphasize important concepts. All references to file names and commands are case sensitive and should be typed accurately.

Kind of Text	Examples
Body text; *italicized* for emphasis, new terms, and book titles	Refer to the *LynxOS User's Guide.*

Environment variables, file names, functions, methods, options, parameter names, path names, commands, and computer data
Commands that need to be highlighted within body text, or commands that must be typed as is by the user are **bolded**.
	`ls`
	`-l`
	`myprog.c`
	`/dev/null`
	`login:` **`myname`**
	`#` **`cd /usr/home`**

Text that represents a variable, such as a file name or a value that must be entered by the user	`cat` *`filename`* `mv` *`file1 file2`*

Blocks of text that appear on the display screen after entering instructions or commands	`Loading file /tftpboot/shell.kdi`
	`into 0x4000`
	`.....................`
	`File loaded. Size is 1314816`
	`Copyright 2000 LynuxWorks, Inc.`
	`All rights reserved.`
	
	`LynxOS (ppc) created Mon Jul 17`
	`17:50:22 GMT 2000`
	`user name:`

Keyboard options, button names, and menu sequences	**Enter, Ctrl-C**

Special Notes

The following notations highlight any key points and cautionary notes that may appear in this manual.

NOTE: These callouts note important or useful points in the text.

CAUTION! Used for situations that present minor hazards that may interfere with or threaten equipment/performance.

Technical Support

LynuxWorks Technical Support is available Monday through Friday (holidays excluded) between 8:00 AM and 5:00 PM Pacific Time (U.S. Headquarters) or between 9:00 AM and 6:00 PM Central European Time (Europe).

The LynuxWorks World Wide Web home page provides additional information about our products, Frequently Asked Questions (FAQs), and LynuxWorks news groups.

LynuxWorks U.S. Headquarters

Internet: `support@lnxw.com`
Phone: (408) 979-3940
Fax: (408) 979-3945

LynuxWorks Europe

Internet: `tech_europe@lnxw.com`
Phone: (+33) 1 30 85 06 00
Fax: (+33) 1 30 85 06 06

World Wide Web

`http://www.lynuxworks.com`

CHAPTER 1

Introduction

POSIX.1b Description

The POSIX.1b standard encompasses real-time extensions to the POSIX.1 standard. POSIX.1b functionality includes shared memory, messages, real-time signals, clocks and timers, scheduling, semaphores, memory locking, synchronized I/O, and asynchronous I/O.

This *Guide* acts as a tutorial and describes the differences in compiler flags between POSIX.4 Draft 9 and POSIX.1b.

NOTE: Throughout the example code, error checking is not performed; function calls for which error checking is not executed are assumed to be successful. The reader should not accept this as a programming style. Error checking is omitted in order to keep this document at a reasonable size and to place emphasis on the migration—the purpose of this document. In order to simplify the examples, obvious include files (i.e., `<stdio.h>`) have not been shown.

For a guide to POSIX.1b programming style with LynxOS, please refer to the on-line example programs provided with the distribution and to other related documents.

The following definitions clarify how the various POSIX specifications interrelate:

POSIX.1	The basic operating system standard, also known as POSIX 1003.1 - POSIX.1 was approved in 1988.
POSIX.1b	Amendments to POSIX.1 for real-time systems - POSIX.1b was approved in 1993. Also known as POSIX.4 Draft 14.

POSIX.4a (now POSIX.1c) Amendments to POSIX.1 defining thread primitives - thread creation, synchronization, destruction, etc. The POSIX.4 committee decided that keeping threads in the POSIX.1b standard would delay approval, due to the complexity of the threads issue. Also, a separate POSIX specification for the threads interface allows vendors the option to exclude the other real-time support, which POSIX.1b requires. POSIX.4a Draft 8 was approved and renamed POSIX.1c. LynxOS supports POSIX.4a Draft 4.

Draft 9 In this Guide, “Draft 9” refers to Draft 9 of the POSIX 1003.4 standard. Draft 9 is an intermediate draft leading to POSIX.1b. Draft 9 has been the target for LynxOS since version 2.0. However, LynxOS event handling is based on Draft 10.

Draft 10 The next intermediate draft after Draft 9, which, among other things, made some important and useful changes to event handling

The POSIX support in LynxOS has been targeted (mostly) for Draft 9 of the then-evolving POSIX.4 standard, and for the approved POSIX.1 standard. POSIX.4 changed significantly between Draft 9 and Draft 14; Draft 14 is the version which was approved as the POSIX.1b standard.

The POSIX.1 standard, when amended by POSIX.4, became the POSIX.1b standard, and included all of the facilities specified in both documents. LynxOS meets the POSIX.1b standard, and also supports Draft 4 of the POSIX.4a standard.

In this *Guide*, “POSIX.1b,” means the POSIX.1b standard as approved by the IEEE. “Draft *num*” refers to the *num* draft of the POSIX.4 standard.

Overview of Major Changes

- The compilation environment now defaults to POSIX.1b, rather than POSIX.4 Draft 9 (referred to as "Draft 9" or "P4D9"). To invoke Draft 9 functionality, you must specify the option **`-mposix4d9`** when compiling an application. For more information, see the *LynxOS Release Notes*.
- Significant library structure changes have been made in this release of LynxOS. Please see "Library Structure and Compiler Option Changes" on page 4 in this chapter for more information.
- It is not possible to mix the Draft 9 and POSIX.1b standard versions in the same application. Users must choose between the two with compile-time options. LynuxWorks recommends using the POSIX.1b standard for all future development.
- Not all Draft 9 features have corresponding equivalents in POSIX.1b. Some features have been discontinued.
- POSIX.1b does not include real-time files. Draft 9 programs using this facility have no migration equivalent in POSIX.1b. Draft 9 real-time file support still exists, and LynuxWorks will continue to support this interface as a proprietary feature in future releases.
- Message queues have changed significantly from Draft 9 to POSIX.1b. Some facilities, such as determining the ID of the sender of a message, and buffer management, are no longer available. The new message queue support is streamlined, with better performance.
- The events facility from Draft 9 has been abandoned. The equivalent to this interface is the real-time signals facility.
- Semaphores have changed from binary semaphores in Draft 9 to counting semaphores in POSIX.1b.
- Draft 9 support for named semaphores, shared memory, and message queues makes use of a file system. LynxOS's POSIX.1b support uses simple strings as names for these objects. There is no file system involvement.
- There are new facilities in POSIX.1b which do not exist in Draft 9. The **`mmap`** facility (which allows files, devices, and shared memory objects to be mapped into memory) is new. The `mmap` facility works only for shared memory objects in the first release.

Library Structure and Compiler Option Changes

In this release of LynxOS, both POSIX.1b and POSIX.4 Draft 9 (referred to as "4D9") are supported. However, POSIX.1b is now the default compilation environment. This change that involved modifying the library structure and compile time options from previous LynxOS releases.

Library Structure Changes

To use the POSIX library routine calls, compiler command line instructions had to contain the **`-mposix`** switch (for `gcc`) or the **`-X`** switch (for `cc`). These options instructed the linker to link with **`-lc_p`**.

The POSIX library routines for POSIX.1a and POSIX.1b have been merged into the library **`libc.a`**. The libc.a library exists in `/lib` and `/lib/thread` (for multithreads). Two new libraries have also been added:

- `liblynx.a`
  Contains some LynxOS-specific library calls (not conforming to any standard)
- `libposix4d9.a`
  Contains support for POSIX.4 Draft 9

Compiler Option Changes

In this new structure, to compile an application with 4D9 functionality, you must compile with the **`-mposix4d9`** option. This option signals the compiler to define **`-D__POSIX4_D9__`** and to link with the **`libposix4d9.a`** library. Thus, you do not need to change 4D9 source code with the new library structure, but you will need to recompile with the `-mposix4d9` option. Also, you cannot mix 4D9 functionality with POSIX.1b in a single application. Your application may fail to link or may exhibit unpredictable results at run-time.

The following table summarizes the changes to the compiler command line options.

Table 1-1: Compiler Option Changes in LynxOS

POSIX Specification	LynxOS	
	cc (x86)	gcc
POSIX 1003.1 C Language Standard	Default	Default
POSIX 1003.1b Final version of 1003.4	Default	Default
POSIX 1003.4a Draft 4 Thread Extensions	`-m`	`-mthreads`
POSIX 1003.4 Draft 9 Real-Time Extensions	`-mposix4d9`	`-mposix4d9`

New Library Structure Issues

Name Conflicts Between liblynx.a and libc.a

With the library reorganization, it has been necessary to remove from the library `libc.a` some non-POSIX functions that have the same names as POSIX functions. These functions are now contained in the **`liblynx.a`** library and each has an interface that is either LynxOS-specific or is BSD-compatible.

The following is a description of each of these functions and how its interface differs from the POSIX interface:

1. `getgroups()`

   This function differs in its arguments. The POSIX version (now in `libc.a`) has the following prototype:

   ```
   int getgroups(int, gid_t *);
   ```

   The LynxOS version (now in `liblynx.a`) is BSD-compatible:

   ```
   int getgroups(int *, gid_t *);
   ```

2. `getpgrp()`

   This function differs in its arguments. The POSIX version (now in `libc.a`) has the following prototype:

   ```
   pid_t getpgrp(void);
   ```

   The LynxOS version (now in `liblynx.a`) is BSD-compatible:

   ```
   pid_t getpgrp(pid_t);
   ```

3. LynxOS semaphore functions

   These functions include `sem_count()`, `sem_delete()`, `sem_get()`, `sem_nsignal()`, `sem_reset()`, `sem_signal()`, and `sem_wait()`. Only one of these functions, **`sem_wait()`**, has a name conflict with POSIX, but because the functions are expected to be used as a group, the whole set has been moved to `liblynx.a`. The POSIX version of `sem_wait()` (now in `libc.a`) has the following prototype:

   ```
   int sem_wait(sem_t *);
   ```

   The LynxOS version (now in `liblynx.a`) is LynxOS-specific:

   ```
   int sem_wait(int);
   ```

4. `sigaction()`

   This function differs in its functionality. When the POSIX version (in `libc.a`) is used to set up a signal handler for a particular signal, and that signal subsequently interrupts a system call in progress, then after returning from the handler, the system call aborts and `errno` sets to `EINTR`.

   When the LynxOS version (now in `liblynx.a`) is used in the same fashion, and the signal subsequently interrupts a system call in progress (as above), then after returning from the handler, the system call is resumed where it has been left off.

   Either version can be made to mimic the behavior of the other with the appropriate flag. The POSIX version can be made to behave like the LynxOS version by using the **`SA_NOABORT`** flag. The LynxOS version can be made to behave like the POSIX version by using the **`SA_ABORT`** flag.

BSD behavior is not supported by LynxOS. BSD differs from both POSIX and LynxOS in that under BSD, a system call is restarted after it is interrupted by a signal.

NOTE: Please see Appendix A, "Functions Callable from Signal Handlers" for a list of functions, including `sigaction()`, required in POSIX.1b to be callable by signal handlers. This is to prevent corruption of the state of a library or any other subtle failure.

---

5. `signal()`

   POSIX does not specify the **`signal()`** function, and strictly adhering POSIX applications should not use it. However, LynxOS does provide a "POSIX-like" `signal()` in `libc.a` that can be used if necessary.

   The LynxOS version of `signal()` (now in `/lib/liblynx.a`) differs in its functionality the same way that **`sigaction()`** does (see Item 4, above). Unlike `sigaction()`, however, there are no flags passed to `signal()`, so there is no way to make the "POSIX-like" version mimic LynxOS behavior or vice-versa.

6. `sleep()`, `susleep()`, `usleep()`

   LynxOS provides two versions of these functions. One is LynxOS-specific and is similar to BSD **`sleep()`**. (For more information about this version, see the man page for `sleep()`.)

   The other version complies with the POSIX.1 standard. The functions **`susleep()`** and **`usleep()`** are not specified in the POSIX.1 standard, but because they are closely related to `sleep()`, LynxOS provides two separate versions of them as well, found in `libc.a` and `liblynx.a`.

Of the above functions, the ones users need to be most concerned about are **`getgroups()`**, **`getpgrp()`**, and the LynxOS semaphore functions, because using unintended versions of these functions always produces incorrect results.

For `getgroups()`, the POSIX version provides the same level of functionality as the LynxOS version, so it may be easier to convert the source code to the POSIX version. For `getpgrp()`, the LynxOS version provides more functionality than the POSIX version, because it allows the user to obtain the process group of a given process ID. If this usage is not needed, however, then users may consider converting the code to the POSIX version.

For `sigaction()` and `signal()`, the differences are less noticeable, and in many cases do not matter. When converting to the POSIX version, be aware that using the POSIX version usually requires more error checking around system calls to handle the case when they are interrupted by a signal handler.

For `sleep()`, the differences between the LynxOS and POSIX versions often do not matter. Converting from LynxOS `sleep()` to POSIX `sleep()` depends on how a given application uses this function.

Identifying Function Usage in Applications

To identify the usage of these functions in executables or object files, use the **nm** and **grep** utilities as follows:

```
$ nm file | grep -w function
```

For an executable, the output line from the command above should show a **T**, meaning external text symbol. For an object file, it should show a **U**, meaning unresolved symbol.

To identify the function usage in source files, use the `grep` utility.

Other Functions in liblynx.a

The following functions are temporarily included in `liblynx.a` as well:

```
lsbrk()
mkcontig()
smem_create()
smem_get()
smem_remove()
vmtopm()
```

Users may need to link with the `liblynx.a` library if their applications use these functions, even though this has not been necessary in previous releases.

Using Parts of liblynx.a in an Application

Sometimes an application needs to use a mixture of the LynxOS and POSIX versions of functions discussed in this section (for example, needs LynxOS **getpgrp()**, but needs POSIX **sem_wait()**). To achieve this, users must extract the object files that contain the needed LynxOS functions from `liblynx.a` and link with these objects directly instead of `liblynx.a`. Alternatively, users may put the extracted objects into their own smaller library. Users should then link the application with this smaller library instead of `liblynx.a`.

NOTE: Some of the objects in `liblynx.a` contain more than one function (for example `sleep()`, `susleep()`, and `usleep()` are combined into one object), so it is not possible to use the LynxOS version of one such function and the POSIX version of another.

Other General Changes

Refer to the **`sysconf()`** man pages for a list of new parameters which can be passed to the `sysconf()` function. Also, refer to the **`pathconf()`** and **`fpathconf()`** man pages for a list of new parameters that can be passed to these two functions.

The lists of run-time invariant values and compile-time symbolic constants have changed. Due to their length, these lists are not reproduced here. Refer to Table 2.5 and Table 2.10 of the POSIX.1b specification (IEEE 1003.1b) for these lists.

Table 1-2: Changes in Errno Values

Draft 9	POSIX.1b
EINPROG	EINPROGRESS
EFTYPE	EINVAL
No Equivalent	EBADMSG
No Equivalent	EMSGSIZE
EFAIL	No Equivalent
ENWAIT	No Equivalent

Table 1-3: Change in Compile Time Symbolic Constant

Draft 9	POSIX.1b
_POSIX_BINARY_SEMAPHORES	_POSIX_SEMAPHORES

CHAPTER 2 *Scheduling*

Scheduler Priority

The main difference in scheduling functionality is the way scheduling priorities are handled. In Draft 9, scheduling priorities are defined as type **`int`**. However, POSIX.1b defines a new structure, **`sched_param`**, which encloses the priority field as type **`sched_priority`**. A pointer to the structure `sched_param` must be passed to all scheduling functions. An example of these changes is shown below.

Draft 9 code

```
#include <sys/sched.h>

main()
{
        int prio;
        pid_t pid;
        :
        prio = getprio(pid);
        :
}
```

Equivalent POSIX.1b code

```
#include <sched.h>

main()
{
        int prio;
        struct sched_param parameter;
        pid_t pid;
        :
        sched_getparam(pid, &parameter);
        prio = parameter.sched_priority;
        :
}
```

Changes to Macros

Another minor change is that a number of Draft 9 macros are replaced by functions in POSIX.1b as explained in the table below.

Table 2-1: Scheduling Interface Changes

Draft 9	POSIX.1b
`<sys/sched.h>`	`<sched.h>`
No Equivalent	`struct sched_param`
`setscheduler()`	`sched_setscheduler()`
`getscheduler()`	`sched_getscheduler()`
`setprio()`	`sched_setparam()`
`getprio()`	`sched_getparam()`
`yield()`	`sched_yield()`
`PRIO_??_MAX macros`	`sched_get_priority_max()`
`PRIO_??_MIN macros`	`sched_get_priority_min()`
`RR_INTERVAL macro`	`sched_rr_get_interval()`

The following examples illustrates these changes:

Draft 9 code

```
#include <sys/sched.h>

main()
{
        printf("FIFO min prio = %d\n", PRIO_FIFO_MIN);
}
```

Equivalent POSIX.1b code

```
#include <sched.h>

main()
{
        printf("FIFO min prio = %d\n",
                sched_get_priority_min(SCHED_FIFO));
}
```

Macros vs. Functions

Draft 9 defines a number of macros for certain scheduler parameters. Six of these macros defined the minimum and maximum scheduler priorities for the three scheduling policies:

```
PRIO_FIFO_MIN
PRIO_RR_MIN
PRIO_OTHER_MIN
PRIO_FIFO_MAX
PRIO_RR_MAX
PRIO_OTHER_MAX
```

These were replaced by two functions in POSIX.1b. The function **`sched_get_priority_min()`** takes a scheduling policy as input and returns the minimum priority for it. The function **`sched_get_priority_max()`** takes a scheduling policy as input and returns the maximum priority for it.

Draft 9 defines the macro **`RR_INTERVAL`** for the interval for the **`SCHED_RR`** policy. This is replaced by a new function **`sched_rr_get_interval()`** in POSIX.1b.

yield ()

The Draft 9 **`yield()`** function does not return anything, and sets no error numbers. The equivalent **`sched_yield()`** function in POSIX.1b returns an `int` and sets an error number on failure.

SCHED_OTHER

The behavior for the **`SCHED_OTHER`** scheduling policy has not changed. This is the same as **`SCHED_DEFAULT`**, which is the LynxOS proprietary scheduling policy.

Non-Preemptible Scheduling Policy

LynxOS implements a non-preemptible scheduling policy called SCHED_NONPREEMPT. A process running under this policy cannot be preempted by any other process until it voluntarily sleeps or blocks waiting for a semaphore or mutex object. An example of this policy is a garbage collector running with low priority and performing critical work so that it must not be interrupted. The sample code of the application is as follows:

```
void garbage_collector(arg)
void *arg;
{
  ...
  while (1) { /* endless loop */
    if (waste_ratio() > max_waste_ratio) {
      /* garbage collection, critical area
       */
      ...
    }
    sleep(GC_TIMEOUT);
  } /* end of loop */
}

main()
{
  pthread_attr_t attr;
  struct sched_param prio;
  pthread_t tid;
  ...
  pthread_attr_create(&attr);
  pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);
  pthread_attr_setschedpolicy(&attr, SCHED_NONPREEMPT);
  prio.sched_priority = PRIO_NONPREEMPT_MIN;
  pthread_attr_setschedparam(&attr, &prio);
  pthread_create(&tid, &attr, garbage_collector, NULL);
  ...
}
```

LynxOS also defines 2 constants for the SCHED_NONPREEMPT policy designating the maximum and minimum priorities for this policy. These constants are:

- PRIO_NONPREEMPT_MAX
- PRIO_NONPREEMPT_MIN

These priorities can also be obtained using the sched_get_priority_max() and sched_get_priority_min() functions.

Interoperability

There have been no changes in the standard scheduling facilities, and, therefore, interoperability is preserved. Two processes, one using Draft 9 scheduling and another using POSIX.1b scheduling would get the CPU slices as if they used the same version of scheduling.

CHAPTER 3 *Real-Time Signals*

There are three basic types of signal functions available under LynxOS:

- Normal signals (as defined in POSIX.1)
- Events (Drafts 9 and 10)
- Real-time signals (POSIX.1b)

It is important to note that real-time signals may be thought of as an inter-process communication (IPC) tool. Real-time signals are only one possible IPC mechanism made available in POSIX.1b (e.g., messages, semaphores, and shared memory are also considered IPC mechanisms). Different IPC functions vary in functionality and performance.

Real-time signals are often *not* the best choice for IPC.

Normal Signals Versus Events

From one point of view, signals and events are slightly different user interfaces layered on top of the same underlying LynxOS support. In both cases, **`sigaction()`** notifies the operating system that the process is using a signal/event handler. A difference is that the signal handler and the event handler do not have the same calling sequence. The words "signal" and "event" are used almost interchangeably because the two interfaces are nearly identical.

There are a couple of key differences between events and signals. The default action for a signal is specified on the **`signal()`**man page. Most of the signals have already-defined names and functionality, such as **`SIGKILL`** and **`SIGCORE`**. Only a few user-defined signals are available. The events facility adds more "signals" to the list of possible signals, and all of these new signals are available for user-defined functions.

Events can also carry data; the event handler receives the event number and a small amount of data. The signal handler and event handler have different parameters. The main difference between Draft 9 and Draft 10 events is the handling function's calling sequence. Thus, signals, Draft 9 events, and Draft 10 events all have different-looking handlers.

Events are queued. Normal signals are not queued under LynxOS; the POSIX.1b standard does not define any particular queueing behavior for normal signals.

Events Versus Real-Time Signals

Events and real-time signals differ primarily in their default functionality. For Drafts 9 and 10, the default action for an event is for the process to ignore them. For POSIX.1b, the default action is to terminate the process.

Once again, the handler's calling sequence has changed. And as with all POSIX.1b functions, the functions and compile-time constants have new names. There are minor changes in the data structures, as well.

For real-time signals, users must call `sigaction()` to notify the operating system that a signal handler is being used. Events implementation requires a call to `sigaction()`; LynxOS users with applications coded under Drafts 9 and 10 already call `sigaction()`.

Both events and real-time signals are queued in FIFO order, and are delivered in that order.

The sigaction Structure

In order to add real-time signal handling to the pre-existing POSIX.1 **`sigaction`** structure, a new flag, **`SA_SIGINFO`**, is defined by POSIX.1b. This flag is in the **`sa_flags`** member of the `sigaction` structure.

The `SA_SIGINFO` flag specifies the signal handler that is desired. If `SA_SIGINFO` is set, it is possible to pass a small amount of data to the signal handler (see signal handler synopses below). If `SA_SIGINFO` is not set, then the signal handler does not receive data.

A new member, **`sa_sigaction`**, has been added to the `sigaction` structure. At signal delivery time, the `sa_sigaction` member is called if the `SA_SIGINFO` flag is set in `sa_flags`, otherwise **`sa_handler`** is called.

sigaction Structure Contents

The `sigaction` POSIX.1 structure contains at least the following members:

```
void (*sa_handler)();
sigset_t sa_mask;
int sa_flags;
```

The `sigaction` POSIX.1b structure now contains at least the following members:

```
void (*sa_sigaction)();                                     /*
SA_SIGINFO set */
void (*sa_handler)();                                       /*
SA_SIGINFO NOT set */
sigset_t sa_mask;
int sa_flags;
               /* NEW: set SA_SIGINFO flag if  */
               /* you want signals to have the */
               /* data queued and retrieved    */
```

NOTE: LynxOS does not implement the Draft 9 and 10 specification for the event handler. Instead, users are required to call `sigaction` to set up the event handler. LynxOS customers with existing Drafts 9 and 10 applications need not add the `sigaction` call; it should be in the application where needed.

NOTE: The default functionality for LynxOS is Draft 10 events. To use Draft 9 events, the user needs to set the `SA_D9EV` flag in the `sa_flags` member of the `sigaction` structure. When the `SA_D9EV` flag is set, be sure to remove it when porting an application to POSIX.1b.

The Event Structure

The new name of the event structure is now **`sigevent`**. The fields are similar. In both cases, a signal's value is an application-defined value, which is passed to the signal-catching function at the time of signal delivery (allowing the signal to pass a small amount of data).

event Structure Contents

The Drafts 9 and 10 event structure must include at least the following members:

```
evt_class_t evt_class;                                      /*
signal number */
```

```
void *evt_value;                                             /*
signal value */
void *evt_handler ();                                        /*
signal handler; not used by LynxOS*/
evtset_t  evt_classmask;  /* signal handler; not used by LynxOS*/
```

NOTE: **`evt_handler`** and **`evt_classmask`** are not used by LynxOS; the user is required to call `sigaction`.

The `sigevent` POSIX.1b structure must include at least the following members:

```
int sigev_signo;                                             /*
signal number */
union sigval sigev_value;                                    /*
signal value */
int sigev_notify;                                            /*
notification type*/
```

The **`sigval`** union must contain at least the following members:

```
int sival_int;                                               /*
integer signal value */
void *sival_ptr;                                             /*
pointer signal value */
```

The `sigval` union allows the user more flexibility in using the value passed by the signal sending code, because it is guaranteed to be large enough for an integer or a pointer, whichever is larger. The Draft 9 and 10 event handlers do not allow an integer, if it happens to be larger than a pointer on a given implementation.

The values for the **`sigev_notify`** member in the `sigval` union above are as follows:

`SIGEV_NONE`	*no signal will be sent*
`SIGEV_SIGNAL`	*signal will be sent*

This member has been added to the `sigevent` structure by POSIX.1b to allow different implementation-defined notification mechanisms.

NOTE: If the value in this field is not set to `SIGEV_SIGNAL`, the process does not receive a signal.

---

signal and event Handler Synopses

For reference, the following POSIX versions have the signal and event handler calling sequences listed below:

- The POSIX.1 signal handler calling sequence is as follows:

  ```
  signal_handler(int signo);
  ```

- The Draft 9 event handler sequence is as follows:

  ```
  event_handler(void *sigdata, int signo);
  ```

- The Draft 10 event handler sequence is as follows:

  ```
  event_handler(int signo, void *sigdata);
  ```

  (Please note the reversal of the **signo** and **sigdata** arguments.)

- The POSIX.1b signal handler sequence is as follows:

  - If SA_SIGINFO is not set in sa_flags

  ```
  signal_handler(int signo);
  ```

  - If SA_SIGINFO is set in sa_flags

  ```
  signal_handler(int signo, siginfo_t *info, void *context);
  ```

siginfo_t Structure

The **siginfo_t** structure must include at least these members:

```
int si_signo;                                                  /*
signal number      */
int si_code;                                                   /*
cause of signal    */
union sigval si_value;                                         /*
signal value       */
```

For both events and real-time signals, it is possible to pass a small amount of data along with the signal to the handler.

For Drafts 9 and 10, the data was put into the event structure as void *evt_value and received by the event handler as void *sigdata.

In POSIX.1b, the data is put into the **sigevent** structure as **sigev_value**. This value is received by the signal handler via the new siginfo_t structure.

The sigaction flag SA_SIGINFO must be set to access the data, because the signal handler used if SA_SIGINFO is not set, does not include siginfo_t.

Additionally, **si_signo** has the same value as the first argument, **signo**, in the signal_handler structure does.

Table 3-1: siginfo_t.si_code

Value	Meaning
SI_USER	Due to kill() function
SI_QUEUE	Due to sigqueue() function
SI_TIMER	Due to timer expiration
SI_ASYNCIO	Due to completion of asynchronous I/O
SI_MESGQ	Due to arrival of message on an empty message queue

The signal handler parameter, **context**, is not used in the LynxOS implementation.

- If SA_SIGINFO is set in sa_flags, use the sa_sigaction member of sigaction structure, taking the following information into consideration:
  - The signal number must be in the range of SIGRTMIN through SIGRTMAX.
  - A real-time signal is sent.

  Queued data is passed to the signal handler if the cause of the signal (passed in si_code) is due to one of any of the following members being called: SI_QUEUE, SI_TIMER, SI_ASYNCIO, or SI_MESGQ.

  The signal handler is the function specified in sa_sigaction.

  The signal handler calling sequence is as follows:

  ```
  signal_handler(int signo, siginfo_t *info, void *context);
  ```
- If SA_SIGINFO is not set in sa_flags, use the sa_handler member of the sigaction structure, taking into consideration that:
  - A normal style signal is sent
  - No data is passed to the signal handler.
  - The signal handler is the function specified in sa_handler

  The signal handler calling sequence is as follows:

  ```
  signal_handler(int signo);
  ```

NOTE: If `sigqueue()` is called to send a signal that is not within the range of `SIGRTMIN` to `SIGRTMAX`, the data is discarded, and the signal posts to the receiving process. If the signal is already pending, the process does not necessarily receive it more than once.

The events facility from Draft 9 (and Draft 10, which LynxOS also supports) is replaced by real-time signals in POSIX.1b. Draft 9 events and real-time signals are distinctly different. Real-time signals are integrated with user (non-real-time) signals. The primary distinction between Draft 9 events and real-time signals is the default behavior; events are ignored while real-time signals terminate the process.

Data Structures

The `event` structure from Drafts 9 and 10 is replaced by the `sigevent` structure in POSIX.1b with the following members:

Table 3-2: `sigevent` Structure

Type	Name	Description
`int`	`sigev_signo`	Signal number
`union sigval`	`sigev_value`	Signal value
`int`	`sigev_notify`	Notification type

The **`sigev_signo`** member specifies the signal to be generated. The **`sigev_value`** member is the application-defined value, which is passed to the signal-catching function at the time of signal delivery. This is a part of the **`siginfo_t`** structure in the signal-catching function, which is described in the table entitled "`siginfo_t`".

Table 3-3: `sigval` Union

Type	Name	Description
`int`	`sival_int`	Integer signal value
`void *`	`sival_ptr`	Pointer signal value

Either an application-defined value of the type `int` or a pointer can be passed through the `sigval` union. The `sigev_notify` member can have either of two values: SIGEV_SIGNAL or SIGEV_NONE. SIGEV_SIGNAL queues a signal when the event occurs. SIGEV_NONE delivers no asynchronous notification when the event occurs.

For the sigaction structure defined by POSIX.1, a new flag, **SA_SIGINFO**, is defined by POSIX.1b for the sa_flags member. This flag *must* be used when setting up a handler to queue a real-time signal from POSIX.1b.

Also, under POSIX.1b, a new member, **sa_sigaction**, is defined for the sigaction structure. This new member must be used for the signal handler instead of sa_handler whenever the SA_SIGINFO flag is set.

sa_handler and sa_sigaction should not be set simultaneously.

The following table shows the various cases of the sa_flags members and the features associated with them; the **SA_NOCLDSTOP** flag does not affect the flags in this table.

Table 3-4: sa_sigaction.sa_flags

Flag	Feature
None (with a valid POSIX.1 signal value)	POSIX.1 signal
SA_D9EV set	Draft 9 event
None (with signal number between EVTCLASS_MIN and EVTCLASS_MAX)	Draft 10 event
SA_SIGINFO	POSIX.1b real-time signal

POSIX.1b defines another structure, **siginfo_t**, which is used to contain code that identifies the cause of a signal. The address of this structure is used as an argument to the signal-catching function.

Table 3-5: siginfo_t

Type	Name	Description
int	si_signo	Signal number
int	si_code	Cause of signal
union sigval	si_value	Signal value

The **si_signo** member contains the signal number. It is the same as the signal number argument of the signal-catching function. The **si_code** member encodes the cause of the signal.

The **`si_value`** member is the same as the application-specified signal value when the `si_code` member is one of `SI_QUEUE`, `SI_TIMER`, `SI_ASYNCIO`, or `SI_MESGQ`; see the table entitled "`siginfo_t.si_code`".

Signal Handlers

The signal handler synopsis for POSIX.1b is different from what it was in Draft 9.

Table 3-6: Signal Handlers

Draft	Handler Synopses
Draft 9	`event_handler(void *sigdata, int signo)`
Draft 10	`event_handler(int signo, void *sigdata)`
POSIX.1b	`signal_handler(int signo)` if `SA_SIGINFO` not set in `sa_flags` for `signo`
	`signal_handler(int signo, siginfo_t *info, void *context)` if `SA_SIGINFO` set in `sa_flags` for `signo`

Table 3-7: POSIX.1b Signal Handlers

Argument	Meaning
`int signo`	Signal number of the signal being delivered
`siginfo_t *info`	Pointer to a `siginfo_t` structure that encodes the signal number, the cause of the signal, and an application-specified signal value. This data structure is explained above.
`void *context`	Unused in the LynxOS implementation

Use of the sigqueue Function

Under Drafts 9 and 10, there is no explicit mechanism to send an event to a process. Events were generated as a result of a timer expiration, completion of asynchronous I/O, etc.

LynxOS provides a proprietary **`ekill()`** function to explicitly send an event to a process. POSIX.1b provides a **`sigqueue()`** function to explicitly queue a real-time signal to a specific process.

The following program illustrates the use of this function and signal handlers from POSIX.1b; the use of the `siginfo_t` structure in the signal handler informs the process of the cause of the signal:

```
#include <signal.h>

void signal_handler(int signo, siginfo_t *info, void *context);

main()
{
       struct sigaction sa;
       union sigval sig_value;
       :
       sa.sa_sigaction = signal_handler;
       sigemptyset(&sa.sa_mask);
       sa.sa_flags = SA_SIGINFO;

       /* SIGRTMIN is chosen more or less randomly,
          but it’s in the required range */
       sigaction(SIGRTMIN, &sa, NULL);

       sig_value.sival_int = 1000;
       sigqueue(getpid(), SIGRTMIN, sig_value);
       :
}

void signal_handler(signo, info, context)
int signo;
siginfo_t *info;
void *context;
{
       :
       printf("In signal handler!!\n");
       printf("Signal number = %d\n", signo);
       printf("Signal value (int) = %d\n",
               info->si_value.sival_int);

       switch(info->si_code) {
       case SI_USER:
               printf("Here due to a kill() function!\n");
               break;
       case SI_QUEUE:
               printf("Here due to a sigqueue() function!\n");
               break;
       case SI_TIMER:
               printf("Here due to a timer expiration!\n");
               break;
       case SI_ASYNCIO:
               printf("Here due to completion of asynch I/O!\n");
               break;
       case SI_MESGQ:
               printf("Here due to arrival of a message!\n");
               break;
```

```
        default:
                printf("No idea why here!!\n");
        }
        :
}
```

Sending a Real-Time Signal to a Process

Drafts 9 and 10 provide an **`evtraise()`** function to generate an event for a process. This can be migrated to POSIX.1b with the `sigqueue()` function:

Draft 9 Code

```
#include <sys/events.h>

void event_handler(void *evt_value,
        evt_class_t evt_class, evtset_t evt_mask);

main()
{
        struct event ev;
        :
        ev.evt_handler = event_handler;
        ev.evt_value = NULL;
        ev.evt_class = EVTCLASS_MIN;
        evtemptyset(&ev.evt_classmask);

        evtraise(&ev);
        :
}

void event_handler(evt_value, evt_class, evt_mask)
void *evt_value;
evt_class_t evt_class;
evtset_t evt_mask;
{
        :
        :
}
```

Equivalent POSIX.1b Code

```
#include <signal.h>

void signal_handler(int signo, siginfo_t *info,
        void *context);

main()
{
        struct sigaction sa;
        union sigval value;
        :
        sa.sa_sigaction = signal_handler;
        sa.sa_flags = SA_SIGINFO;
        sigemptyset(&sa.sa_mask);
```

```
        sigaction(SIGRTMIN, &sa, NULL);

        value.sival_ptr = NULL;
        sigqueue(getpid(), SIGRTMIN, value);
        :
}

void signal_handler(signo, info, context)
int signo;
siginfo_t *info;
void *context;
{
        :
        :
}
```

Polling for a Real-Time Signal

The **evtpoll()** function from Drafts 9 and 10 is superseded by the **sigwaitinfo()** and **sigtimedwait()** functions. The following example illustrates a conversion of the `evtpoll()` facility to POSIX.1b:

Draft 9 Code

```
#include <sys/events.h>
#include <sys/timers.h>

main()
{
        evtset_t set;
        struct timespec timeout;
        void *value;
        evt_class_t class;
        :
        evtemptyset(&set);
        evtaddset(&set, EVTCLASS_MIN);
        evtaddset(&set, EVTCLASS_MAX);

        timeout.tv_sec = 2;
        timeout.tv_nsec = 0;

        if (evtpoll(&set, &timeout, &value, &class) != -1) {
                printf("Received event no. %d\n", *class);
                printf("Pointer to value = %d\n",
                        (int *) *value);
        }
        :
        evtpoll(&set, NULL, &value, &class);
        printf("Received event no. %d\n", *class);
        printf("Pointer to value = %d\n", (int *) *value);
        :
}
```

Equivalent POSIX.1b Code

```
#include <signal.h>

main()
{
        sigset_t set;
        siginfo_t info;
        struct timespec timeout;
        :
        sigemptyset(&set);
        sigaddset(&set, SIGRTMIN);
        sigaddset(&set, SIGRTMAX);

        timeout.tv_sec = 2;
        timeout.tv_nsec = 0;

        if (sigtimedwait(&set, &info, &timeout) != -1) {
                printf("Dequeued signal no. %d\n",\
                        info->si_signo);
                printf("Pointer to value = %d\n",
                        (int *) *info->si_value.sival_ptr);
        }
        :
        sigwaitinfo(&set, &info);
        printf("Dequeued signal no. %d\n",\
                info->si_signo);
        printf("Pointer to value = %d\n",
                (int *) *info->si_value.sival_ptr);
        :
}
```

Equivalence for Other Draft 9 Event Functions

There is no equivalent to the **evtsigclass()** function from Drafts 9 and 10 in POSIX.1b because there is no longer a need for it. Most of the other event-related functions have no specific equivalents in POSIX.1b. However, their functionality is provided by the appropriate signal functions from POSIX.1; refer to “Changes from Draft 9 to POSIX.1b” on page 30,” for more information, including differences between **evtsuspend()** of Drafts 9 and 10 and **sigsuspend()** of POSIX.1.

Timers, Message Queues, and Asynchronous I/O

With Drafts 9 and 10, it is possible to send events after a timer has expired, and when asynchronous I/O is completed. In POSIX.1b, real-time signals can be sent to a process without explicitly queuing them with a `sigqueue()` call. This can happen when a timer expires (see Chapter 6, “Clocks and Timers” on page 57), a message arrives on an empty message queue (see Chapter 4, “Message Queues” on page 37), or asynchronous I/O completion (see Chapter 9, “Asynchronous I/O” on page 79).

To use POSIX.1b real-time signals, the `sigevent` structure must be used, and the handlers must be set up according to POSIX.1b specification. The `sa_flags` for the `sigaction` structure must be set to `SA_SIGINFO`, and the `sigev_notify` member for the `sigevent` structure must be set to `SIGEV_SIGNAL`.

Changes from Draft 9 to POSIX.1b

The final interface for real-time extended signals differs from the events facility in Draft 9 (and Draft 10, which LynxOS also supports) as follows:

Table 3-8: Extended Signal Interface

Drafts 9 & 10	POSIX.1b
Default action: Ignore the event	Default action: Terminate the process
`<sys/events.h>`	`<signal.h>`
`EVTCLASS_MIN`	`SIGRTMIN`
`EVTCLASS_MAX`	`SIGRTMAX`
`struct event`	`struct sigevent`
No Equivalent	`struct siginfo_t`
32 event values	`RTSIG_MAX` signals
No Equivalent	`sigqueue()`
`evtraise()`	`sigqueue(getpid(),....)`
`evtpoll()`	`sigwaitinfo()`, `sigtimedwait()`
`evtsigclass()`	No Equivalent

The following are important points about POSIX.1b interface:

- There is a class of signals in the `SIGRTMIN` to `SIGRTMAX` range which are treated as "real-time signals." The default action for a real-time signal is to terminate the process, as opposed to ignoring an event in Drafts 9 and 10.
- It is possible to have multiple occurrences of the same signal queued in FIFO order to a process.

- There is no explicit mechanism under Drafts 9 and 10 to send an event to a given **pid**. LynxOS provides a proprietary `ekill()` function for this purpose. In POSIX.1b, a new function, `sigqueue()`, is used to queue a signal with a specified value to a process. Signals can also be queued as a result of asynchronous I/O completion, timer expirations, etc.
- Queuing is not supported for signals generated by the `kill()` function or by events such as timer expiration, hardware fault detection, etc. Such signals have no effect on signals already queued for the same signal number.
- When multiple, unblocked signals in the range of `SIGRTMIN` to `SIGRTMAX` are pending, the unblocked signal with the lowest signal number in that range is delivered. No other ordering of signal delivery is specified.
- The cause for signal generation can be communicated to the signaled process.

Data Structures

The `event` structure from Drafts 9 and 10 is replaced by one of the following `sigevent` POSIX.1b structures:

Table 3-9: `sigevent` Structures

Type	Name	Description
`int`	`sigev_signo`	Signal number
`union sigval`	`sigev_value`	Signal value
`int`	`sigev_notify`	Notification type

The `sigev_signo` member specifies the signal to be generated. The `sigev_value` member is the application-defined value to be passed to the signal-catching function at the time of signal delivery. This is a part of the `siginfo_t` structure in the signal-catching function detailed in the table "`siginfo_t` Structure".

Table 3-10: sigval Union

Type	Name	Description
int	sival_int	Integer signal value
void *	sival_ptr	Pointer signal value

Either an application-defined value of type int or a pointer can be passed through the sigval union. The sigev_notify member can have either of two values: SIGEV_SIGNAL or SIGEV_NONE. SIGEV_SIGNAL queues a signal when the event occurs. SIGEV_NONE delivers no asynchronous notification when the event occurs.

For the sigaction structure (from POSIX.1), a new flag, **SA_SIGINFO**, is defined by POSIX.1b for the sa_flags member. This flag must be used when setting up a handler to queue a real-time signal from POSIX.1b.

Also, under POSIX.1b, a new member, **sa_sigaction**, is defined for the sigaction structure. This new member *must* be used for the signal handler instead of sa_handler whenever the SA_SIGINFO flag is set.

sa_handler and sa_sigaction should not be set simultaneously

Table 3-11: sa_sigaction.sa_flags

Flag	Feature
None (and signal no. with a valid value)	POSIX.1 signal
SA_D9EV set	Draft 9 event
None (and signal no. between EVTCLASS_MIN and EVTCLASS_MAX)	Draft 10 event
SA_SIGINFO	POSIX.1b real-time signal

The table above does not consider the SA_NOCLDSTOP flag. It may or may not be set; but that does not affect this table.

Under POSIX.1b `siginfo_t` (a new structure) contains the code identifying the cause of the signal. The address of this structure is used as an argument to the signal-catching function.

Table 3-12: `siginfo_t` Structure

Type	Name	Description
`int`	`si_signo`	Signal number
`int`	`si_code`	Cause of signal
`union sigval`	`si_value`	Signal value

The `si_signo` member contains the signal number. It is the same as the signal number argument of the signal-catching function. The `si_code` member encodes the cause of the signal.

Table 3-13: `siginfo_t.si_code`

Value	Meaning
`SI_USER`	Due to `kill()` function
`SI_QUEUE`	Due to `sigqueue()` function
`SI_TIMER`	Due to timer expiration
`SI_ASYNCIO`	Due to completion of asynchronous I/O
`SI_MESGQ`	Due to arrival of message on an empty message queue

The `si_value` member is the same as the application-specified signal value, when the `si_code` member is one of `SI_QUEUE`, `SI_TIMER`, `SI_ASYNCIO`, or `SI_MESGQ`.

NOTE: The signal handler synopsis for POSIX.1b is different than Draft 9. LynxOS supports Draft 10 event handlers by default.

Table 3-14: Signal Handlers

Draft	Handler Synopses
Draft 9	`event_handler(void *sigdata, int signo)`
Draft 10	`event_handler(int signo, void *sigdata)`
POSIX.1b	`signal_handler(int signo)` if `SA_SIGINFO` not set in `sa_flags` for `signo`. This is the same as POSIX.1 signal handler
	`signal_handler(int signo, siginfo_t *info, void *context)` if `SA_SIGINFO` set in `sa_flags` for `signo`

Table 3-15: POSIX.1b Signal Handlers

Arguments	POSIX.1b Meanings
`int signo`	Signal number of the signal being delivered
`siginfo_t *info`	Pointer to a `siginfo_t` structure that encodes the signal number, cause of the signal and an application-specified signal value
`void *context`	Unused in the LynxOS implementation

Indefinite/Timed Wait

In Drafts 9 and 10 one function, `evtpoll()`, waits for events with and without a timeout. POSIX.1b provides a separate interface for these two operations – `sigwaitinfo()` and `sigtimedwait()`.

Ability to Send Arbitrary Data

The `sigevent` structure from POSIX.1b contains a `sigev_value` entry, which is a union. With this entry, it is possible to send either an integer value or a pointer to arbitrary data along with the signal.

Drafts 9 and 10 Event Functions

Functions `evtemptyset()`, `evtfillset()`, `evtaddset()`, `evtdelset()`, `evtismember()`, `evtprocmask()`, `evtsuspend()`, `evtsetjmp()`, and `evtlongjmp()` have no equivalents in POSIX.1b. The POSIX.1 signal functions (`sigemptyset()`, `sigfillset()`, etc.) provide corresponding functionality. However, there are slight differences. Drafts 9 and 10 `evtsuspend()` takes two arguments. The second argument is a `timespec` structure, which allows a timed wait. The POSIX.1 `sigsuspend()` takes only one argument; a timed wait is not possible. The effect of an `evtsuspend()` with a timed wait can be simulated with the new `sigtimedwait()` function from POSIX.1b. However, this is not a true equivalence. It is not possible to invoke a signal handler with `sigtimedwait()`; it can only return values.

Interoperability

Events and real-time signals are not inter-operable between Drafts 9, 10, and POSIX.1b. Users should not try to catch a Draft 9 or 10 event with a signal handler from POSIX.1b, or vice versa. The effects of such behavior are undefined.

CHAPTER 4 Message Queues

The message queue interface has changed extensively. A number of Draft 9 features are completely eliminated in favor of performance improvements and ease of use in POSIX.1b. The following is a list of Draft 9 functions with no equivalent in POSIX.1b; applications that depend on these functions may not migrate easily to POSIX.1b:

- `msgalloc()`
- `msgfree()`
- `mqpurge()`
- `mqgetpid()`
- `mqgetevt()`
- `mqputevt()`

Some of the functions above may be partially simulated by other functions in POSIX.1b. For example, **`mqgetpid()`** can be simulated by encoding the **`pid`** of the process in the message itself. The **`mqpurge()`** function can be simulated with **`mq_receive()`** in a **`while`** loop until it fails. The **`mqputevt()`** and **`mqgetevt()`** functions can be simulated with the **`sigqueue()`** and **`sigwaitinfo()`** functions from POSIX.1b, respectively. The **`msgalloc()`** and **`msgfree()`** functions have no corresponding functionality.

Three other features are no longer supported: asynchronous message sending and receiving; using the **`MSG_MOVE`** and **`MSG_USE`** flags; and selective removing messages from somewhere other than the head of the queue.

There are other important changes; Draft 9 and POSIX.1b differ with respect to persistence and how the names for message queues are implemented. This is similar to the differences for named semaphores as described in Chapter 7, "Semaphores" on page 67.

A comparison of the various message queue features is provided later in this chapter.

Creating Message Queues

POSIX.1b message queues are created with the **`mq_open()`** function and the **`O_CREAT`** flag as opposed to the **`mkmq()`** function from Draft 9, as per the following example:

Draft 9 Code

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mqueue.h>

main()
{
        int mq;
        :
        mkmq("message_queue", MQ_PERSIST | 0666);
        mq = open("message_queue", O_RDWR | O_NONBLOCK,
                MQ_PERSIST | 0666);
        :
        close(mq);
        unlink("message_queue");
        :
}
```

Equivalent POSIX.1b Code

```
#include <mqueue.h>

main()
{
        mqd_t mq;
        :
        mq = mq_open("message_queue", O_CREAT | O_RDWR |
                O_NONBLOCK, 0666, NULL);
        :
        mq_close(mq);
        mq_unlink("message_queue");
        :
}
```

Data Structure Changes

The two data structures, **msgcb** and **mqstatus**, from Draft 9 were combined into a single data structure, **mq_attr**, in POSIX.1b. Also, a number of entries from both of these structures were eliminated. For example, the time stamp and event fields from msgcb have no equivalent in POSIX.1b.

The message length and message data fields from msgcb are now separate arguments for the message sending and receiving functions. A comparison of data structures is provided later in this chapter in “Data Structures” on page 45.

Getting and Setting Message Queue Attributes

The following example illustrates the comparison of the interfaces to get and set message queue attributes. The **mq_maxmsg** and **mq_msgsize** attributes for a POSIX.1b message queue may only be set at creation time. “Changes from Draft 9 to POSIX.1b” on page 44 tabulates the restrictions on setting attributes for a POSIX.1b message queue.

The default Message Queue attributes are as follows:

- mq_flags = 0
- mq_maxmsg = 35
- mq_msgsize = 120

Draft 9 Code

```
#include <sys/mqueue.h>

#define SIZE 1024

main()
{
       int mq;
       struct mqstatus mqstat;
       :
       mkmq("message_queue", MQ_PERSIST | 0666);
       mq = open("message_queue", O_RDWR | O_NONBLOCK,
                MQ_PERSIST | 0666);
       :
       mqstat.mqmaxmsg = 200;
       mqstat.mqrsvmsg = 200;
       mqstat.mqmaxbytes = SIZE;
       mqstat.mqrsvbytes = SIZE;
       mqstat.mqwrap = MQNOWRAP;
       mqstat.mqmaxarcv = 20;
```

```
        mqsetattr(mq, &mqstat);
        :
        mqgetattr(mq, &mqstat);
        printf("mqmaxmsg: %d\n", mqstat.mqmaxmsg);
        printf("mqrsvmsg: %d\n", mqstat.mqrsvmsg);
        printf("mqmaxbytes: %d\n", mqstat.mqmaxbytes);
        printf("mqrsvbytes: %d\n", mqstat.mqrsvbytes);
        printf("mqcurmsgs : %d\n", mqstat.mqcurmsgs);
        printf("mqsendwait: %d\n", mqstat.mqsendwait);
        printf("mqrcvwait : %d\n", mqstat.mqrcvwait);
        printf("mqmaxarcv : %d\n", mqstat.mqmaxarcv);
        printf("mqwrap: %s\n",
                (mqstat.mqwrap==MQWRAP)?"MQWRAP":"MQNOWRAP");
        :
        close(mq);
        unlink("message_queue");
        :
}
```

Equivalent POSIX.1b Code

```
#include <mqueue.h>

#define SIZE 1024

main()
{
        msg_t mq;
        struct mq_attr mqattr;
        :
        mqattr.mq_maxmsg = 200;
        mqattr.mq_msgsize = SIZE;
        mq = mq_open("message_queue", O_CREAT | O_RDWR,
                0666, &mqattr);
        :
        mqattr.mq_flags = O_NONBLOCK;
        mq_setattr(mq, &mqattr, NULL);
        :
        mq_getattr(mq, &mqattr);
        printf("mq_flags : %s\n",
                (mqattr.mq_flags & O_NONBLOCK == O_NONBLOCK)
                        ? "Non-Blocking" : "Blocking");
        printf("mq_maxmsg  : %ld\n", mqattr.mq_maxmsg);
        printf("mq_msgsize : %ld\n", mqattr.mq_msgsize);
        printf("mq_curmsgs : %ld\n", mqattr.mq_curmsgs);
        printf("mq_sendwait: %ld\n", mqattr.mq_sendwait);
        printf("mq_rcvwait : %ld\n", mqattr.mq_rcvwait);
        :
        mq_close(mq);
        mq_unlink("message_queue");
        :
}
```

Sending and Receiving Messages

The following example compares the sending and receiving of messages between message queue facilities. The `fork()` call is only relevant to illustrate how messages may be sent and received between two processes.

Draft 9 Code

```
#include <sys/mqueue.h>

#define SIZE 1024

main()
{
        int mq;
        char buffer[SIZE];
        struct msgcb msgcbp;
        :
        mkmq("message_queue", MQ_PERSIST | 0666);
        mq = open("message_queue", O_RDWR,MQ_PERSIST | 0666);
        :
        switch(fork()) {
        case 0:
                :
                msgcbp.msg_flags = 0;
                msgcbp.msg_bufsize = SIZE;
                msgcbp.msg_data = buffer;
                msgcbp.msg_type = 0;
                mqreceive(mq, &msgcbp);
                :
        default:
                :
                msgcbp.msg_flags = MSG_COPY;
                msgcbp.msg_length = SIZE;
                msgcbp.msg_bufsize = SIZE;
                msgcbp.msg_data = buffer;
                msgcbp.msg_type = 0;
                mqsend(mq, &msgcbp);
                :
        }
        :
        close(mq);
        unlink("message_queue");
        :
}
```

Equivalent POSIX.1b Code

```
#include <mqueue.h>

#define SIZE 1024

main()
{
        msg_t mq;
```

```
        char buffer[SIZE];
        :
        mq = mq_open("message_queue", O_CREAT | O_RDWR,0666, NULL);
        :
        switch(fork()) {
                case 0:
                        :
                        mq_receive(mq, buffer, SIZE, NULL);
                        :
                default:
                        :
                        mq_send(mq, buffer, SIZE, MQ_PRIO_MAX-1);
                        :
        }
        :
        mq_close(mq);
        mq_unlink("message_queue");
        :
}
```

Notification of Message Availability

The new **mq_notify()** function in POSIX.1b is used to notify a process that a message is available on a message queue. This is done by sending a signal to the process when the message queue changes from empty to non-empty as illustrated in the following example.

When a notification request is attached to a message queue, another process may be blocked in **mq_receive()** waiting to receive a message. If a message arrives at the queue, mq_receive() is completed and the notification request remains pending. If there is no process blocked in mq_receive(), the specified signal handler is called.

The next example uses the flag **no_msg** to ensure that the notification request is satisfied, and that the same message is received with an mq_receive() call: The sa_flags flag is set to SA_SIGINFO, and the sigev_notify field is set to SIGEV_SIGNAL to ensure the use of a real-time signal:

```
#include <mqueue.h>
#include <signal.h>

#define SIZE 1024

void signal_handler(int signo, siginfo_t *info, void *context);
volatile int no_msg;

main()
{
        mqd_t mq;
        char buffer[SIZE];
        struct mq_attr mqattr;
        struct sigevent notification;
        struct sigaction sa;
```

```
        mqattr.mq_flags = 0;
        mqattr.mq_maxmsg = 200;
        mqattr.mq_msgsize = SIZE;
        mq = mq_open("message_queue", O_CREAT | O_RDWR,
                0666, &mqattr);
        :
        switch(fork()) {
                case 0:
                        :
                        notification.sigev_signo = SIGRTMIN;
                        notification.sigev_value.sival_int = 0;
                        notification.sigev_value.sigev_notify =
                                SIGEV_SIGNAL;
                        :
                        sa.sa_sigaction = signal_handler;
                        sa.sa_flags = SA_SIGINFO;
                        sigemptyset(&sa.sa_mask);

                        sigaction(SIGRTMIN, &sa, NULL);
                        :
                        mq_notify(mq, &notification);
                        :
                        no_msg = 1;
                        while (no_msg) {
                                :
                                sched_yield();
                        }
                mq_receive(mq, buffer, size, NULL);
                :
                break;
        default:
                :
                mq_send(mq, buffer, size, MQ_PRIO_MAX-1);
                :
        }
        :
        mq_close(mq);
        mq_unlink("message_queue");
        :
}

void signal_handler(signo, info, context)
int signo;
siginfo_t *info;
void *context;
{
        :
        no_msg = 0;
        :
}
```

Changes from Draft 9 to POSIX.1b

Interface Changes

Message queues have changed in a fairly major way. A number of facilities from Draft 9 are no longer available; however, POSIX.1b offers a new facility for notification of message availability. The new implementation offers better performance. Speeds comparable to raw memory copy are attainable using the new, simple POSIX.1b functions

Table 4-1: Message Queue Interface

Draft 9	POSIX.1b
Message queue = special file	Independent of file system
Persistent as well as non-persistent message queues with the `MQ_PERSIST` flag	Persistent message queues
Queue wrapping with *MQWRAP*	No queue wrapping
Buffer management with `MSG_COPY`, `MSG_USE`, `MSG_MOVE`	All messages copied
Message transfer synchronization control with `MSG_WAIT`, `MSG_ASYNC`, `MSG_NOWAIT`	No message synchronization
Truncation control with `MSG_TRUNC`	Overlong messages rejected at the time of sending
Ability to send an event via a message queue	No event sending
Selective message receive order	No equivalent[1]
`<sys/mqueue.h>`	`<mqueue.h>`
`sender_t`	`mqd_t`
`open()`	`mq_open()`
`close()`	`mq_close()`
`mkmq()`	Done with `mq_open()`
`unlink()`	`mq_unlink()`

Table 4-1: Message Queue Interface (Continued)

Draft 9	POSIX.1b
`mqsend()`	`mq_send()`
`mqreceive()`	`mq_receive()`
`mqsetattr()`	`mq_setattr()`
`mqgetattr()`	`mq_getattr()`
`msgalloc()`	No Equivalent
`msgfree()`	No Equivalent
`mqpurge()`	No Equivalent
`mqgetpid()`	No Equivalent
`mqputevt()`	No Equivalent
`mqgetevt()`	No Equivalent
No Equivalent	`mq_notify()` Notify process that a message is available on a queue.
`struct msgcb` `struct mqstatus`	`struct mq_attr` Combines the flags of `msgcb` and `mqstatus`.

1. LynxOS provides the `mq_selective_receive()` function to support this behavior, even though it is not in the POSIX.1b standard.

Data Structures

The `msgcb` and `mqstatus` structures are combined into the single `mq_attr` structure in POSIX.1b.

Table 4-2: Data Structure Equivalence

Draft 9	POSIX.1b
`msgcb.msg_flags`	`mq_attr.mq_flags`
`msgcb.msg_type`	Message Priorities
`msgcb.msg_length`	No Equivalent
`msgcb.msg_bufsize`	No Equivalent

Table 4-2: Data Structure Equivalence (Continued)

Draft 9	POSIX.1b
msgcb.msg_data	No Equivalent
msgcb.msg_event	No Equivalent
msgcb.msg_errno	No Equivalent
msgcb.msg_timesent	No Equivalent
msgcb.msg_sender	No Equivalent
mqstatus.mqrsvmsg	No Equivalent
mqstatus.mqrsvbytes	No Equivalent
mqstatus.mqmaxmsg	mq_attr.mq_maxmsg
mqstatus.mqmaxbytes	mq_attr.mq_msgsize
mqstatus.mqwrap	No Equivalent
mqstatus.mqmaxarcv	No Equivalent
mqstatus.mqcurmsgs	mq_attr.mq_curmsgs
mqstatus.mqsendwait	No Equivalent
mqstatus.mqrcvwait	No Equivalent

Attributes

There are various restrictions on setting attributes for message queues from POSIX.1b. The table below shows the attributes that may be set, and when. All attributes may be queried at any time.

Table 4-3: Message Queue Attributes

Attribute	Set
mq_flags	Yes, any time after creation
mq_maxmsg	Yes, only at creation
mq_msgsize	Yes, only at creation
mq_curmsgs	No

Messages

Message Priorities

POSIX.1b provides a new concept of message priorities. A message is inserted into the queue, and received from the queue according to message priority. This priority is independent of the process priority.

Selective Receive

Draft 9 message queues allow the application to selectively remove queued messages by type. This facility has been replaced by the message priority facility described above. Note that priorities are somewhat less flexible than message typing, because only the highest priority message is retrievable. LynuxWorks has provided a proprietary extension, **`mq_selective_receive`**, to retain this functionality.

Process Priorities

Process priorities come into the picture when sending and receiving messages. If more than one process is blocked while sending to a full message queue (or receiving from an empty message queue) with priority scheduling, the highest-priority process, which has been waiting the longest, is unblocked first.

Synchronization Control

With Draft 9 message queues, it is possible to control how processes waited for each other by specifying the `MSG_WAIT`, `MSG_ASYNC`, and `MSG_NOWAIT` flags on a per-message basis. Such options are not available with POSIX.1b message queues. All messages in POSIX.1b are sent and received with behavior equivalent to the Draft 9 `MSG_NOWAIT`.

Buffer Management

With Draft 9 message queues, it is possible to control the use of buffers to achieve higher performance with the `MSG_MOVE`, `MSG_USE`, and `MSG_COPY` flags on a per-message basis. This feature is discontinued in POSIX.1b. All POSIX.1b message queues have behavior equivalent to the **`MSG_COPY`** flag from Draft 9.

Sending and Receiving Events

Draft 9 allows events to be sent along message queues. POSIX.1b does not provide this capability.

Purging, Data Buffer Allocation/Freeing

Draft 9 provides the following functions:

`mqpurge()`	purge a message queue
`msgalloc()`	allocate a message data buffer
`msgfree()`	free a message data buffer

The **`mqpurge()`** function can be simulated with **`mq_receive()`** in a **`while`** loop until it fails. The **`msgalloc()`** and **`msgfree()`** functions have no corresponding functionality.

Sender ID

The **`mqgetpid()`** function from Draft 9 allowed a process to determine the **`pid`** of the sender process. There is no equivalent function for this feature in POSIX.1b. However, the user can work around this by encoding the `pid` in the message itself.

Queue Wrap

With Draft 9 message queues, it is possible to specify the queue wrap behavior, so that older messages could be overwritten by newer ones as messages were sent to a full queue. This behavior was requested with the **`MQWRAP`** flag at message queue creation time. POSIX.1b does not support this capability.

Time-Stamping

It is possible to time-stamp Draft 9 messages. This feature is not supported by POSIX.1b. An application writer can work around this by encoding the time-stamp in the message.

Truncation Control

Draft 9 provides a **MSG_TRUNC** flag to truncate a message when receiving, if it is larger than the buffer. POSIX.1b does not allow a message to be sent if its length exceeds the message size of the queue, and does not allow **mq_receive** to be called with a buffer size smaller than the message size.

A Pointer-Worth of Data

Draft 9 provides a **MSG_OVERRIDE** flag to indicate receipt of a pointer-worth of data. POSIX.1b does not provide special support for such functionality, although 4-byte-long messages are supported.

Notification of Message Availability

POSIX.1b provides the new function, **mq_notify()**, to notify a process when a message queue changes from empty to non-empty.

exec() Behavior

With Draft 9, message queue file descriptors of a process remain open after **exec()**, except if the **FD_CLOEXEC** flag was set. With POSIX.1b, open message queue descriptors of a calling process are closed upon `exec()`.

New Utilities

LynxOS provides two new utilities, **lipcs** and **lipcrm**, to list and remove message queues (and other POSIX.1b IPC facilities), respectively; refer to the `lipcs` and `lipcrm` man pages for more information.

Interoperability

Draft 9 message queues and POSIX.1b message queues are distinct. There is no interoperability. For example, it is not possible to send a message on a Draft 9 queue and receive it on a POSIX.1b message queue.

CHAPTER 5 *Shared Memory*

Introduction

The **`mmap()`** function is fundamental to the POSIX.1b changes in the shared memory system. Additional changes affect the way shared memory object sizes are specified upon creation. The `mmap()` function from POSIX.1b is not specific to shared memory objects. With its full features, it is a powerful function allowing files and devices to be mapped into process address space.

Also, there are persistence-related differences between the Draft 9 and POSIX.1b specification of shared memory. Draft 9 provides persistent and non-persistent shared memory; persistent shared memory had to be requested explicitly with the `SHM_PERSIST` flag. In contrast, POSIX.1b only provides persistent shared memory and does not require an explicit flag.

Persistence of an object implies that the object and its state (for example, the value of a semaphore, data in a message queue, data for a shared memory object) are preserved once the object is no longer referenced by a process. If the user absolutely needs to migrate non-persistent behavior from Draft 9 to POSIX.1b, here is an alternative method: After all of the processes that wish to use non-persistent shared memory have opened the shared memory, `shm_unlink` the shared memory. The shared memory will be deleted when all references to it are removed, simulating non-persistent shared memory.

Creating and Deleting Shared Memory

The interface to create a shared memory object differs between Draft 9 and POSIX.1b. Draft 9 provides a **`mkshm()`** function which used a **`size`** argument to specify the size of the shared memory object. POSIX.1b shared memory is created with the **`shm_open()`** function with the `O_CREAT` flag. This function does not take a `size` argument; all shared memory objects are of zero size when created.

To specify size, a new POSIX.1b function, **`ftruncate()`**, which is not specific to shared memory, must be used. This function truncates a file to a specified size. If a file is expanded with `ftruncate()`, the expanded part is initialized to zero. When `ftruncate()` is used to expand a shared memory object, the expanded part is initialized to zero. The following example demonstrates the comparison.

NOTE: To delete a shared memory object under POSIX.1b, call `shm_unlink()`. The object is actually destroyed after the last process unmaps the object.

Draft 9 Code

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/shmmap.h>

#define SIZE 1024

main()
{
       int shm;
       off_t size = SIZE;
       :
       mkshm("shmem", SHM_PERSIST | 0666, size);
       shm = open("shmem", O_RDWR, SHM_PERSIST | 0666);
       :
       close(shm);
       unlink("shmem");
       :
}
```

Equivalent POSIX.1b Code

```
#include <sys/mman.h>

#define SIZE 1024

main()
{
       int shm;
       :
       shm = shm_open("shmem", O_CREAT | O_RDWR, 0666);
       ftruncate(shm, SIZE);
       :
       close(shm);
       shm_unlink("shmem");
       :
}
```

Mapping and Unmapping Shared Memory

This code compares mapping and unmapping a shared memory object between Draft 9 and POSIX.1b code; refer to "Changes from Draft 9 to POSIX.1b" on page 54 for flags specific to shared memory.

Draft 9 Code

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/shmmap.h>

#define SIZE 1024

main()
{
        int shm;
        void *mem_ptr;
        :
        mkshm("shmem",.....);
        shm = open("shmem",.....);
        :
        mem_ptr = shmmap(shm, NULL, SIZE, 0,
                SHM_READ | SHM_WRITE);
        :
        shmunmap(mem_ptr, 0);
        :
        close(shm);
        unlink("shmem");
        :
}
```

Equivalent POSIX.1b code

```
#include <sys/mman.h>

#define SIZE 1024

main()
{
        int shm;
        char *mem_ptr;
        :
        shm = shm_open("shmem",.....);
        ftruncate(shm,.....);
        :
        mem_ptr = mmap(NULL, SIZE, PROT_READ |
                PROT_WRITE, MAP_SHARED, shm, 0);
        :
        munmap(mem_ptr, SIZE);
        :
        close(shm);
        shm_unlink("shmem");
        :
}
```

Changes from Draft 9 to POSIX.1b

The following table summarizes the shared memory interface changes:

Table 5-1: Shared Memory Interface

Draft 9	POSIX.1b
Shared memory object = Special file	Independent of file system
`<sys/shmmap.h>`	`<sys/mman.h>`
SHM_READ	PROT_READ
SHM_WRITE	PROT_WRITE
SHM_EXEC	PROT_EXEC
No Equivalent	PROT_NONE
SHM_EXACT	MAP_FIXED
shmmap()	Done by mmap()
shmunmap()	Done by munmap()
mkshm()	Done by shm_open()
open()	shm_open()
close()	close()
unlink()	shm_unlink()
No Equivalent	ftruncate() Truncates a file to specified length

Persistence

Draft 9 supports persistent and non-persistent shared memory. POSIX.1b shared memory is persistent.

Size of Shared Memory Object

The interface changed to specify shared memory object size (when it is created). In Draft 9, **size** was specified as an argument to the **mkshm()** function. In POSIX.1b, a shared memory object is created with **shm_open()**, which does not take a size argument. All shared memory objects are of zero size when created.

The size is specified with a new `ftruncate()` function. This function is not specific to shared memory, and can be used to truncate any file to a specified size. When `ftruncate()` is used to expand a shared memory object, the expanded part is initialized to zero.

Shared/Private Changes

Currently, LynxOS does not support the **`MAP_PRIVATE`** flag. The **`MAP_SHARED`** flag can be used, and changes to a shared memory object change the underlying object. With the `MAP_PRIVATE` flag (which will be supported in a subsequent release), changes to a shared memory object change the private copy of that object for that process but not the underlying object.

fork() Behavior

In the absence of `MAP_PRIVATE`, there are no changes to the **`fork()`** behavior with respect to shared memory. Memory mappings created by the parent are retained by the child process. With the `MAP_PRIVATE` flag (when it is supported), mappings before `fork()` in the parent also appear in the child. After `fork()`, the parent and the child are independent with respect to private mappings. The semantics are copy-on-write.

Protection

POSIX.1b supports all protections supported by Draft 9 (read, write, execute). In addition a new **`PROT_NONE`** flag is provided to suppress the ability to access data.

msync() and mprotect() Functions

In addition to `mmap()`, POSIX.1b provides **`msync()`** and **`mprotect()`**, which are unrelated to shared memory. These functions correspond to the `_POSIX_MAPPED_FILES` and `_POSIX_MEMORY_PROTECTION` feature test macros. These functions are only relevant to map files and devices with the `mmap()` function.

Return Values

A notable difference exists between the return values of `shmmap()` in Draft 9 and `mmap()` in POSIX.1b. `shmmap()` returns `NULL` upon failure, while `mmap()`

returns `MAP_FAILED`. All successful `mmap()` returns are guaranteed not to return `MAP_FAILED`.

New Utilities

Lynx provides two new utilities, **`lipcs`** and **`lipcrm`**, to list and remove shared memory objects, respectively; refer to the `lipcs` and `lipcrm` man pages for more information.

Inter-Operability

There is no inter-operability between Draft 9 and POSIX.1b shared memory. Two processes, one using Draft 9 shared memory, and the other using POSIX.1b shared memory, cannot access the same underlying object.

CHAPTER 6 *Clocks and Timers*

Introduction

The most important change in the clock and timer interface is that the following Draft 9 functions have no equivalent in POSIX.1b:

- `resrel()`
- `resabs()`
- `ressleep()`

With non-trivial changes to the code, programs using the above functions can be migrated to POSIX.1b. Also, timer overrun counts are handled with a new function, **`timer_getoverrun()`**, instead of through the Draft 9 **`itimercb`** structure.

Refer to “Changes from Draft 9 to POSIX.1b” on page 63 for equivalence between function names from Draft 9 and POSIX.1b.

Resolution of a Clock

The Draft 9 **`resclock()`** function obtains the maximum value of a clock. The equivalent **`clock_getres()`** function from POSIX.1b does not allow this. The following code is a comparison:

Draft 9 Code

```
#include <sys/timers.h>

main()
{
        struct timespec res, maxval;
        :
```

```
        resclock(TIMEOFDAY, &res, &maxval);
        printf("Resolution: %ld sec %ld nsec\n",
                res.tv_sec, res.tv_nsec);
        printf("Max. val.: %ld sec %ld nsec\n",
                maxval.tv_sec, maxval.tv_nsec);
        :
}
```

Equivalent POSIX.1b Code

```
#include <time.h>

main()
{
        struct timespec res;
        :
        clock_getres(CLOCK_REALTIME, &res);
        printf("Resolution: %ld sec %ld nsec\n",
                res.tv_sec, res.tv_nsec);
        :
}
```

Creation and Deletion of a Timer

The timer creation interface has changed from Draft 9 to POSIX.1b. In Draft 9, the function **mktimer()** returned `timer_t`. In POSIX.1b, the function **timer_create()** returns an `int` with a result argument of the type `timer_t`.

Also, there are differences in how notification type is specified. In POSIX.1b code, the **sa_flags** flag is set to `SA_SIGINFO`, and the **sigev_notify** field is set to `SIGEV_SIGNAL` to ensure the use of a real-time signal. In addition, the `sa_sigaction` member is used to set the signal handler.

Timers are deleted with the **timer_delete()** function, instead of with **rmtimer()**.

Draft 9 Code

```
#include <sys/timers.h>
#include <sys/events.h>

void event_handler(void *evt_value,
        evt_class_t evt_class, evtset_t evt_mask);

main()
{
        timer_t timer1, timer2;
        struct itimercb itimercbp;
        :
        timer1 = mktimer(TIMEOFDAY, DELIVERY_SIGNALS, NULL);
```

```
        :
        rmtimer(timer1);
        :
        itimercbp.itcb_event.evt_handler = event_handler;
        itimercbp.itcb_event.evt_value = NULL;
        itimercbp.itcb_event.evt_class = EVTCLASS_MIN;
        evtemptyset(&itimercbp.itcb_event.evt_classmask);
        itimercbp.itcb_count = 0;

        timer2 = mktimer(TIMEOFDAY, DELIVERY_EVENTS,
                &itimercbp);
        :
        rmtimer(timer2);
        :
}

void event_handler(evt_value, evt_class, evt_mask)
void *evt_value;
evt_class_t evt_class;
evtset_t evt_mask;
{
        :
        :
}
```

Equivalent POSIX.1b Code

```
#include <time.h>
#include <signal.h>

void signal_handler(int signo, siginfo_t *info,
        void *context);

main()
{
        timer_t timer1, timer2;
        struct sigevent se;
        struct sigaction sa;
        :
        timer_create(CLOCK_REALTIME, NULL, &timer1);
        :
        timer_delete(timer1);
        :
        se.sigev_signo = SIGRTMIN;
        se.sigev_value.sival_ptr = NULL;
        se.sigev_notify = SIGEV_SIGNAL;

        sa.sa_sigaction = signal_handler;
        sigemptyset(&sa.sa_mask);
        sa.sa_flags = SA_SIGINFO;

        sigaction(SIGRTMIN, &sa, NULL);

        timer_create(CLOCK_REALTIME, &se, &timer2);
        :
        timer_delete(timer2);
        :
}

void signal_handler(signo, info, context)
```

```
int signo;
siginfo_t *info;
void *context;
{
        :
        :
}
```

Setting a Timer

Draft 9 provides two functions, **abstimer()** and **reltimer()**, to set the value of a timer. These are used to set the absolute and the relative value, respectively. POSIX.1b provides only one function, **timer_settime()**, and requires an extra flag argument to choose between absolute and relative countdowns.

Draft 9 Code

```
#include <sys/timers.h>

main()
{
        timer_t timer;
        struct itimerspec value, ovalue;
        struct timespec now;
        :
        timer = mktimer(TIMEOFDAY, DELIVERY_SIGNALS, NULL);
        :
        value.it_value.tv_sec = 2;
        value.it_value.tv_nsec = 0;
        value.it_interval.tv_sec = 0;
        value.it_interval.tv_nsec = 0;

        reltimer(timer, &value, &ovalue);
        :
        getclock(TIMEOFDAY, &now);

        value.it_value.tv_sec = now.tv_sec + 5;
        value.it_value.tv_nsec = 0;
        value.it_interval.tv_sec = 0;
        value.it_interval.tv_nsec = 0;

        abstimer(timer, &value, &ovalue);
        :
}
```

Equivalent POSIX.1b Code

```
#include <time.h>

main()
{
        timer_t timer;
```

```
        struct itimerspec value, ovalue;
        struct timespec now;
        :
        timer_create(CLOCK_REALTIME, NULL, &timer);
        :
        value.it_value.tv_sec = 2;
        value.it_value.tv_nsec = 0;
        value.it_interval.tv_sec = 0;
        value.it_interval.tv_nsec = 0;

        timer_settime(timer, 0, &value, &ovalue);
        :
        clock_gettime(CLOCK_REALTIME, &now);

        value.it_value.tv_sec = now.tv_sec + 5;
        value.it_value.tv_nsec = 0;
        value.it_interval.tv_sec = 0;
        value.it_interval.tv_nsec = 0;

        timer_settime(timer, TIMER_ABSTIME, &value,
                &ovalue);
        :
}
```

Determining Timer Overrun Count(s)

The following example shows how Draft 9 code, which determines the timer overrun count(s), can be migrated to the POSIX.1b:

Draft 9 Code

```
#include <sys/timers.h>
#include <sys/events.h>

void event_handler(void *evt_value,
        evt_class_t evt_class, evtset_t evt_mask);

main()
{
        timer_t timer;
        struct itimercb itimercbp;
        :
        itimercbp.itcb_event.evt_handler = event_handler;
        itimercbp.itcb_event.evt_value = NULL;
        itimercbp.itcb_event.evt_class = EVTCLASS_MIN;
        evtemptyset(&itimercbp.itcb_event.evt_classmask);
        itimercbp.itcb_count = 0;

        timer = mktimer(TIMEOFDAY, DELIVERY_EVENTS,
                &itimercbp);
        :
        printf("Overrun count = %d\n",
                itimercbp.itcb_count);
        :
}
```

```
void event_handler(evt_value, evt_class, evt_mask)
void *evt_value;
evt_class_t evt_class;
evtset_t evt_mask;
{
       :
       :
}
```

Equivalent POSIX.1b Code

```
#include <time.h>

void signal_handler(int signo, siginfo_t *info,
       void *context);

int overrun;
timer_t timer;

main()
{
       struct sigevent se;
       struct sigaction sa;
       :
       se.sigev_signo = SIGRTMIN;
       se.sigev_value.sival_ptr = NULL;
       se.sigev_notify = SIGEV_SIGNAL;

       sa.sa_sigaction = signal_handler;
       sigemptyset(&sa.sa_mask);
       sa.sa_flags = SA_SIGINFO;

       sigaction(SIGRTMIN, &sa, NULL);

       timer_create(CLOCK_REALTIME, &se, &timer);
       :
       printf("Overrun count = %d\n", overrun);
       :
}

void signal_handler(signo, info, context)
int signo;
siginfo_t *info;
void *context;
{
       :
       overrun = timer_getoverrun(timer);
       :
}
```

Changes from Draft 9 to POSIX.1b

Almost all Draft 9 timer functionality has an equivalent in POSIX.1b. Where there is no equivalent function, it can be emulated with a series of other functions. There are some notable differences between the two interfaces.

Table 6-1: Clock And Timer Interface

Draft 9	POSIX.1b
`<sys/timers.h>`	`<time.h>`
`TIMEOFDAY`	`CLOCK_REALTIME`
`getclock()`	`clock_gettime()`
`setclock()`	`clock_settime()`
`resclock()`	`clock_getres()`
`mktimer()`	`timer_create()`
`rmtimer()`	`timer_delete()`
`gettimer()`	`timer_gettime()`
`reltimer()`	`timer_settime(timerid,0,...)`
`abstimer()`	`timer_settime(timerid, TIMER_ABSTIME,..)`
`itimercb.itcb_count`	`timer_getoverrun()`
`resabs()`	Documentation
`resrel()`	Documentation
`ressleep()`	Documentation

Overrun Count

The overrun count is handled differently in the Draft 9 and POSIX.1b interfaces. In Draft 9, the structure **`itimercb`** encloses the overrun count for the timer. In POSIX.1b, no separate structure item is used. The overrun count is accessed through the new **`timer_getoverrun()`** function.

Signal/Event Associated with a Timer

In Draft 9, the structure `itimercb` encloses an event associated with a timer. In POSIX.1b, this structure no longer exists. Instead, a `sigevent` structure for a real-time signal is used as an argument to **`timer_create()`**.

Signal Number

In Draft 9, the flag `DELIVERY_SIGNALS` for `notify_type` in `mktimer()` delivers the `SIGALRM` signal. If the flag is `DELIVERY_EVENTS`, an event is associated with the timer. In POSIX.1b if the `sigevent` structure passed to `timer_create()` is `NULL`, the default signal is used (which is `SIGALRM` for `CLOCK_REALTIME`). If the `sigevent` structure for `timer_create()` specifies `SIGEV_SIGNAL` and if the `SA_SIGINFO` bit is set for any real-time signal between `SIGRTMIN` and `SIGRTMAX`, then that signal is queued.

Relative and Absolute Times

Draft 9 provides two functions, `reltimer()` and `abstimer()`, to set a timer either with a relative offset or the absolute value, respectively. In POSIX.1b, the `timer_settime()` function does both jobs. The new `TIMER_ABSTIME` flag specifies the choice of an absolute timer instead of a relative timer.

Resolutions

With non-trivial changes to the code, programs using `resrel()`, `resabs()`, and `ressleep()` can be migrated to POSIX.1b.

Get Timer Value

The `gettimer()` function from Draft 9 always returns the `it_interval` last set by `reltimer()` or `abstimer()`. On the other hand, the `timer_gettime()` function from POSIX.1b always returns how much time remains on the timer.

Create Timer

The `mktimer()` function from Draft 9 returns a `timer_t`, and has a `notify_type` argument. The `timer_create()` function from POSIX.1b has a `timer_t` as a result argument, and returns an `int`. The notification type is handled through the `sigev_notify` entry in the `sigevent` structure, which is an argument.

Clock Resolution

The `resclock()` function from Draft 9 provides an extra argument to obtain maximum possible time value for a clock. The `timer_getres()` function from POSIX.1b does not provide such an argument.

nanosleep()

In Draft 9, the second argument to the `nanosleep()` function is updated to contain the unslept time. In POSIX.1b, this is done only if that argument is non-`NULL`.

Pending Signals/Events

Draft 9 specifies that deleting a timer would cancel any pending events for that timer. However for POSIX.1b, even after deleting a timer, signals queued from it continue to be queued. Also with POSIX.1b, signals queued from a timer continue to be queued, even after disarming or resetting a timer.

Interoperability

There is no inter-operability between the Draft 9 and final standard versions of clocks and timers. These are distinct and separate features. However, the system-wide time-of-day clock is the same for all processes. Two processes, one a Draft 9 process using `TIMEOFDAY`, and another POSIX.1b process using `CLOCK_REALTIME` can access the same clock.

CHAPTER 7 Semaphores

Introduction

Most of the old functionality of semaphores has an equivalent under the new standard. The most important change is that Draft 9 provides binary semaphores, whereas POSIX.1b provides counting semaphores. Also, Draft 9 provides only named semaphores, whereas POSIX.1b provides named as well as unnamed semaphores. Two new functions, **`sem_init()`** and **`sem_destroy()`**, were introduced for unnamed semaphores.

Draft 9 semaphores were special files and relied on the underlying file system. POSIX.1b semaphores are independent of the file system. Names for POSIX.1b named semaphores are implemented as simple strings without file system involvement. These strings are processed with a new, efficient name service. This difference between Draft 9 and POSIX.1b may be experienced in other ways. For example, the **`ls`** and **`rm`** utilities could access the Draft 9 semaphores. This cannot be done for POSIX.1b named semaphores.

LynxOS provides two new utilities, **`lipcs`** and **`lipcrm`**, to list and remove named IPC objects – semaphores, shared memory objects, and message queues. Refer to the `lipcs` and `lipcrm` man pages for more information.

Also, there are persistence-related differences between the Draft 9 and POSIX.1b semaphores. Draft 9 provides persistent and non-persistent semaphores. Persistent semaphores were requested explicitly with the `SEM_PERSIST` flag. POSIX.1b, on the other hand, only provides persistent semaphores.

Persistence of an object implies that the object and its state (e.g., value of a semaphore, data in a message queue, data for a shared memory object) are preserved once the object is no longer referenced by a process. If the user absolutely needs to migrate non-persistent behavior from Draft 9 to POSIX.1b, here is an alternative method: After all the processes that wish to use a non-persistent semaphore have opened the semaphore, `sem_unlink` the semaphore.

The semaphore is deleted when all references to it are removed, simulating a non-persistent semaphore.

Unnamed Semaphores

The following example illustrates the use of POSIX.1b unnamed semaphores. Notice that for two processes to use an unnamed semaphore, it must reside in shared memory.

```
#include <sys/mman.h>
#include <semaphore.h>

main()
{
        struct shared_info *sp;
        :
        fd = shm_open(shmname, oflags, mode);
        :
        sp = mmap(0, SHMSIZE, PROT_READ|PROT_WRITE,
                 MAP_SHARED, fd, 0);
        close(fd);
        shm_unlink(shmname);
        :
        sem_init(&sp->sem, TRUE, 0);
        :
        pid = fork();

        if (pid) {
                 /* Parent */
                 :
                 sem_post(&sp->sem);
                 :
        }
        else {
                 /* Child */
                 :
                 sem_wait(&sp->sem);
                 :
        }
}
```

Creating a Named Semaphore

Named semaphores for POSIX.1b are created using the function **sem_open()** with the O_CREAT flag, instead of the **mksem()** function from Draft 9.

The following example compares the creation of a named semaphore. Note the use of the **SEM_PERSIST** flag in Draft 9 code to request creation of a persistent semaphore. For POSIX.1b, no special flag is necessary, because POSIX.1b only provides persistent semaphores.

Draft 9 Code

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/sem.h>

main()
{
        int sem;
        :
        mksem("semaphore", SEM_PERSIST | 0666,
                 STATE_LOCKED);
        sem = open("semaphore", O_RDWR, SEM_PERSIST | 0666);
        :
        close(sem);
        unlink("semaphore");
        :
}
```

Equivalent POSIX.1b Code

```
#include <semaphore.h>

main()
{
        sem_t *sem;
        :
        sem = sem_open("semaphore", O_CREAT, 0666, 0);
        :
        sem_close(sem);
        sem_unlink("semaphore");
        :
}
```

Posting and Waiting on Semaphores

Refer to “Changes from Draft 9 to POSIX.1b” on page 72 later in this chapter, for equivalence of function names to post to and wait on semaphores. The following example compares Draft 9 and POSIX.1b for this functionality.

Draft 9 Code

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/sem.h>

main()
{
       int sem;
       :
       mksem("semaphore",.....);
       sem = open("semaphore",....);
       :
       /* this could also be semwait() */
       semifwait(sem);
       :
       sempost(sem);
       :
}
```

Equivalent POSIX.1b Code

```
#include <semaphore.h>

main()
{
       sem_t *sem;
       :
       sem = sem_open("semaphore",.....);
       :
       /* this could also be sem_wait() */
       sem_trywait(sem);
       :
       sem_post(sem);
       :
}
```

Conditional Posting to Semaphores

The Draft 9 facility for conditional posting to a semaphore with the **semifpost()** function (only if a process is waiting for it), was removed. However, POSIX.1b offers a new function, **sem_getvalue()**, to allow the user to obtain the value of a semaphore at any time.

If the semaphore is locked, sem_getvalue() returns a zero or a negative number. The absolute value of this number indicates the number of processes waiting for the semaphore. This value is sampled at an unspecified time inside the sem_getvalue() call. The following example illustrates the use of sem_getvalue() and sem_post() to simulate the effect of semifpost() from Draft 9. As explained below, the equivalence in this example does not always hold.

Draft 9 Code

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/sem.h>

main()
{
       int sem;
       :
       mksem("semaphore",.....);
       sem = open("semaphore",....);
       :
       semifpost(sem);
       :
}
```

Equivalent POSIX.1b Code

```
#include <semaphore.h>

main()
{
       sem_t *sem;
       int value;
       :
       sem = sem_open("semaphore",......);
       :
       sem_getvalue(sem, &value);
       if (value < 0)
               sem_post(sem);
       :
}
```

The `semifpost()` function from Draft 9 has an inherent race condition. If a process is about to sleep on the semaphore, `semifpost()` would never wake that process up. Therefore, programs that use `semifpost()` have a race condition upon wake-up. (For more details on why `sem_ifpost()` has been removed from POSIX.1b, refer to the rationales of the POSIX.1b standard.)

Also note that `sem_getvalue()` samples the value of the semaphore, and (because there is no locking built into semaphores) does not give a reliable value for heavily-used semaphores. POSIX.1b semaphores are not POSIX.1c condition variables.

Changes from Draft 9 to POSIX.1b

Semaphores changed, although not in a major way. The most important change is that Draft 9 provides binary semaphores, while POSIX.1b provides counting semaphores. Also, Draft 9 provides only named semaphores and they are special files. POSIX.1b provides named and unnamed semaphores; the interface is independent of any file system.

Table 7-1: Semaphore Interface

Draft 9	POSIX.1b
Binary semaphores	Counting semaphores
Semaphores = Special files	Independent of file system
Only named semaphores	Named and unnamed semaphores
Persistent as well as non-persistent semaphores with the flag `SEM_PERSIST`	Persistent semaphores
`<sys/sem.h>`	`<semaphore.h>`
No Equivalent	`sem_init()` Initializes unnamed semaphore
No Equivalent	`sem_destroy()` Destroys unnamed semaphore
`mksem()`	Done by `sem_open()`
`open()`	`sem_open()`
`close()`	`sem_close()`

Table 7-1: Semaphore Interface (Continued)

Draft 9	POSIX.1b
`unlink()`	`sem_unlink()`
`semwait()`	`sem_wait()`
`semifwait()`	`sem_trywait()`
`sempost()`	`sem_post()`
`semifpost()`	No Equivalent
No Equivalent	`sem_getvalue()` Get semaphore value

Conditional Posting

Draft 9 provides a `semifpost()` function to do a conditional post to a semaphore if a process is waiting for it. This functionality is discontinued in POSIX.1b. It can be simulated with a combination of `sem_getvalue()` and `sem_post()`, but is not an atomic operation.

Permission Checking

Since Draft 9 semaphores are special files, there is the overhead of complete file permission checking. In POSIX.1b, this is replaced by an efficient name service for the named semaphores. The new service does not need to do directory traversals or complicated permission checking. User ID and standard POSIX.1 permission checking is performed on a per-object-name basis.

New Utilities

LynxOS provides two new utilities, **`lipcs`** and **`lipcrm`**, to respectively list and remove named semaphores. Refer to the `lipcs` and `lipcrm` man pages for more information.

Interoperability

There is no semaphore interoperability between Draft 9 and POSIX.1b.

CHAPTER 8 *Memory Locking*

The memory locking interface has changed from Draft 9 to POSIX.1b. Some Draft 9 facilities have been discontinued (see below), while a new feature has been introduced to restrict memory locks to the current address space.

For additional information, refer to “Changes from Draft 9 to POSIX.1b” on page 77 later in this chapter.

Locking the Specific Address Space

The ability to restrict memory locking for data, text, or stack segments of a process under Draft 9 (`DATALOCK`, `TXTLOCK`, `STKLOCK` flags) no longer exists. The only behavior supported is the ability to lock a specified address range (`REGLOCK` flag in Draft 9) and the entire process address space (`PROLOCK` flag in Draft 9).

Two new functions have been introduced to lock and unlock the entire address space, instead of using the **`PROLOCK`** flag. The following example illustrates their use and comparison to Draft 9 code.

Draft 9 Code

```
#include <sys/memlk.h>

#define SIZE 1024

main()
{
       void *addr;
       :
       addr = malloc(SIZE);
       memlk(REGLOCK, addr, SIZE);
       :
       memunlk(REGLOCK, addr, SIZE);
       :
}
```

Equivalent POSIX.1b Code

```
#include <sys/mman.h>

#define SIZE 1024

main()
{
        void *addr;
        :
        addr = malloc(SIZE);
        mlock(addr, SIZE);
        :
        munlock(addr, SIZE);
        :
}
```

Locking Future Growth

When locking the entire address space, Draft 9 guaranteed that subsequent growth would also be locked. POSIX.1b provides two flags, **MCL_CURRENT** and **MCL_FUTURE**. These flags request locking current pages or future pages, respectively. Under LynxOS, MCL_CURRENT locks current as well as future pages.

The MCL_FUTURE flag by itself locks only future pages, not current ones. It would be unusual for an application writer to request this flag by itself.

Table 8-1: Memory Locking Flags

Flag	LynxOS Semantics
MCL_CURRENT	Lock current as well as future pages
MCL_CURRENT | MCL_FUTURE	Same as MCL_CURRENT
MCL_FUTURE	Locks only future pages, not current

Draft 9 Code

```
#include <sys/memlk.h>

main()
{
        :
        memlk(PROLOCK, NULL, 0);
        :
        memunlk(PROLOCK, NULL, 0);
        :
}
```

Equivalent POSIX.1b Code

```
#include <sys/mman.h>

main()
{
        :
        mlockall(MCL_CURRENT | MCL_FUTURE);
        :
        munlockall();
        :
}
```

Changes from Draft 9 to POSIX.1b

Table 8-2: Memory-Locking Interface

Draft 9	POSIX.1b
`<sys/memlk.h>`	`<sys/mman.h>`
`MEMLK_BOUNDSIZE`	`PAGESIZE`
`memlk(REGLOCK,addr, size)`	`mlock(addr,size)`
`memunlk(REGLOCK,addr, size)`	`munlock(addr,size)`
`memlk(TXTLOCK|DATALOCK|STKLOCK,..)`	No Equivalent
`memunlk (TXTLOCK|DATALOCK|STKLOCK,..)`	No Equivalent
`memlk(PROLOCK,NULL,0)`	`mlockall(MCL_CURRENT|MCL_FUTURE}` or `mlockall(MCL_CURRENT)`
`memunlk(PROLOCK,NULL,0)`	`munlockall()`
No Equivalent	`mlockall(MCL_FUTURE)`

Locking Flags

Draft 9 allows processes to lock their data, text, or stack segments. There is no support for such functionality in POSIX.1b. The flags used are:

- `TXTLOCK`
- `DATALOCK`
- `STKLOCK`

Multiple Locks

Multiple memory locks can be set for a given region under Draft 9. There are no semantics for multiple locking in POSIX.1b. Memory locking functions can be invoked multiple times for a given address range, but still act as a single lock and are removed by a single unlock.

Locking/Unlocking the Entire Process

In Draft 9, the `PROLOCK` flag was used with `memlk()` and `memunlk()` to specify the entire process address space. In POSIX.1b, there are two new functions, **`mlockall()`** and **`munlockall()`**, for this purpose.

Current/Future Locking

In Draft 9, future memory growth was automatically locked with the `PROLOCK` flag. POSIX.1b provides a flag to request whether current or future pages be locked, with the `MCL_CURRENT` and `MCL_FUTURE` values, respectively. Under the LynxOS implementation, the `MCL_CURRENT` flag locks current as well as future pages. The following table shows the meaning of these flags.

Table 8-3: Memory Locking Flags

Flag	LynxOS Semantics
MCL_CURRENT	Lock current as well as future pages
MCL_CURRENT | MCL_FUTURE	Same as MCL_CURRENT
MCL_FUTURE	Lock only future pages, not current

Interoperability

The memory locking facilities for both Draft 9 and POSIX.1b are based on the same code in LynxOS.

CHAPTER 9 *Asynchronous I/O*

Changes in asynchronous I/O features center primarily around reordered data structure entries and function parameters. In addition, POSIX.1b introduces a new idea of asynchronous I/O priority. The priority of an asynchronous I/O operation can be lowered, but not raised, with respect to the process scheduling priority.

Refer to “Changes from Draft 9 to POSIX.1b” on page 87 for a comparison of data structures and flags specific to asynchronous I/O.

Data Structure Changes

The **aiocb** data structure changed has significantly. Because of these changes, the **liocb** data structure has been eliminated. The following is a comparison of the **aiocb** data structures for Draft 9 and POSIX.1b.

Table 9-1: aiocb Structure

Draft 9	POSIX.1b
aio_offset	aio_offset
aio_event	aio_sigevent
aio_prio	aio_reqprio
aio_whence	No Equivalent (always SEEK_SET)
aio_flag	No Equivalent
aio_errno	No Equivalent
aio_nobytes	No Equivalent
No Equivalent	aio_nbytes
No Equivalent	aio_fildes

Table 9-1: aiocb Structure (Continued)

Draft 9	POSIX.1b
No Equivalent	aio_buf
No Equivalent	aio_lio_opcode

POSIX.1b structure includes the file descriptor, buffer, and listio opcode fields. The new aio_nbytes field has the same semantics as defined by the **read()** and **write()** synopses.

As a result, the synopses for the asynchronous I/O functions have changed as follows:

- File descriptor, buffer, and number of bytes are not passed as separate arguments to aio_read() and aio_write().
- The listio opcode is not passed as a separate argument to lio_listio().
- The aio_whence field has been eliminated. The aio_offset argument is treated as offset from the beginning of the file. The effect is as if aio_whence is always SEEK_SET.
- The aio_errno field has been eliminated. Instead, a new function, aio_error(), with the aiocb argument does the same job.
- The aio_flag field has been eliminated. It is superseded by the aio_sigevent field.
- The afsync() function has been renamed to aio_fsync().
- The aio_nobytes field has been eliminated. The new aio_return() function retrieves the return status from an aiocb structure. aio_return() can be called exactly once per structure; this structure may not be passed to aio_error() or aio_return() again.

There is no relation between the new **aio_nbytes** field and the old **aio_nobytes** field. The aio_return() function may be used on the same **aiocb** structure more than once, as a proprietary extension from LynuxWorks. This can be disabled by a new proprietary library call **aio_setparam()**. Refer to the aio_setparam() man page for more information.

The aio_prio field from Draft 9 was unused and has been replaced with the **aio_reqprio** field. With this field, POSIX.1b asynchronous I/O can be queued in a priority order.

The priority of an asynchronous process is the process priority minus the `aio_reqprio` value. Priority of asynchronous I/O can be lowered, but not raised, with respect to the process priority. However, as a proprietary feature from LynuxWorks, the priority of an asynchronous I/O operation can be raised with respect to the process priority. This is achieved by the **`aio_setparam()`** library call, which is specific to LynxOS. Refer to the `aio_setparam()` man page for more information.

Asynchronous Read and Write

The following example shows a code comparison for an asynchronous write operation. Also, it shows the use of two new functions for POSIX.1b; namely, `aio_return()` and `aio_error()`.

In POSIX.1b code the `sa_flags` flag is set to `SA_SIGINFO`, and the `sigev_notify` field is set to `SIGEV_SIGNAL` to ensure the use of a real-time signal.

Draft 9 Code

```
#include <errno.h>
#include <sys/aio.h>

#define SIZE 256

void event_handler(void *evt_value,
       evt_class_t evt_class, evtset_t evt_mask);

main()
{
       int fd;
       char buf[SIZE];
       struct aiocb cb;
       struct sigaction sa;
       :
       fd = open(.......);
       :
       sa.sa_handler = event_handler;
       sa.sa_flags = SA_D9EV;
       sigemptyset(&sa.sa_mask);

       sigaction(EVTCLASS_MIN, &sa, NULL);

       cb.aio_event.evt_handler = event_handler;
       cb.aio_event.evt_value = NULL;
       cb.aio_event.evt_class = EVTCLASS_MIN;
       evtemptyset(&cb.aio_event.evt_classmask);
       cb.aio_flag = AIO_EVENT;
       cb.aio_offset = 0;
       cb.aio_whence = 0;
```

```
        cb.aio_prio = 0;

        awrite(fd, buf, SIZE, &cb);

        while (cb.aio_errno == EINPROG) {
                :
                :
        }

        printf("Errno = %d, No. of bytes = %d\n",
                cb.aio_errno, cb.aio_nobytes);
        :
}

void event_handler(evt_value, evt_class, evt_mask)
void *evt_value;
evt_class_t evt_class;
evtset_t evt_mask;
{
        :
        :
}
```

Equivalent POSIX.1b Code

```
#include <aio.h>
#include <errno.h>

#define SIZE 256

void signal_handler(int signo, siginfo_t *info, void *context);

main()
{
        int fd;
        char buf[SIZE];
        struct aiocb cb;
        struct sigaction sa;
        int err, ret;
        :
        fd = open(.......);
        :
        sa.sa_sigaction = signal_handler;
        sa.sa_flags = SA_SIGINFO;
        sigemptyset(&sa.sa_mask);

        sigaction(SIGRTMIN, &sa, NULL);

        cb.aio_sigevent.sigev_signo = SIGRTMIN;
        cb.aio_sigevent.sigev_value.sival_ptr = NULL;
        cb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
        cb.aio_offset = 0;
        cb.aio_reqprio = 0;
        cb.aio_fildes = fd;
        cb.aio_buf = buf;
        cb.aio_nbytes = SIZE;

        aio_write(&cb);

        while (aio_error(&cb) == EINPROGRESS) {
```

```
                :
                :
        }

        err=aio_err (&cb);
        ret=aio_return (&cb);
        :
}
void signal_handler(signo, info, context)
int signo;
siginfo_t *info;
void *context;
{
        :
        :
}
```

List Directed I/O

The following example illustrates how a Draft 9 program doing list-directed I/O can be migrated to POSIX.1b. In Draft 9, `LIO_NOWAIT` ignores the final event argument while `LIO_ASYNC` ensures a final event delivery. In POSIX.1b, **`LIO_NOWAIT`** ensures a final signal delivery on completion of the last **`listio`** job. If, however, the final signal argument is `NULL`, no signal is sent.

Draft 9 Code

```
#include <sys/aio.h>

#define SIZE 1024

void evt_handler1(void *evt_value,
        evt_class_t evt_class, evtset_t evt_mask);
void evt_handler2(void *evt_value,
        evt_class_t evt_class, evtset_t evt_mask);
void evt_final_handler(void *evt_value,
        evt_class_t evt_class,evtset_t evt_mask);

main()
{
        int fd1, fd2;
        char buf1[SIZE], buf2[SIZE];
        struct liocb list1, list2, *lcb[2];
        struct sigaction sa;
        struct event final_evt;
        :
        fd1 = open(......);
        fd2 = open(......);
        :
        sa.sa_handler = evt_handler1;
        sa.sa_flags = SA_D9EV;
        sigemptyset(&sa.sa_mask);

        sigaction(EVTCLASS_MIN, &sa, NULL);
```

```
        sa.sa_handler = evt_handler2;
        sa.sa_flags = SA_D9EV;
        sigemptyset(&sa.sa_mask);

        sigaction(EVTCLASS_MIN+1, &sa, NULL);
        :
        list1.lio_opcode = LIO_WRITE;
        list1.lio_fildes = fd1;
        list1.lio_buf = buf1;
        list1.lio_nbytes = SIZE;
        list1.lio_aiocb.aio_event.evt_data = NULL;
        list1.lio_aiocb.aio_event.evt_class = EVTCLASS_MIN;
        list1.lio_aiocb.aio_event.evt_handler =
                evt_handler1;
        evtemptyset(&list1.lio_aiocb.aio_event.
                evt_classmask);
        list1.lio_aiocb.aio_flag = AIO_EVENT;
        list1.lio_aiocb.aio_offset = 0;
        list1.lio_aiocb.aio_whence = 0;
        list1.lio_aiocb.aio_prio = 0;

        list2.lio_opcode = LIO_READ;
        list2.lio_fildes = fd2;
        list2.lio_buf = buf2;
        list2.lio_nbytes = SIZE;
        list2.lio_aiocb.aio_event.evt_data = NULL;
        list2.lio_aiocb.aio_event.evt_class =
                EVTCLASS_MIN+1;
        list2.lio_aiocb.aio_event.evt_handler =
                evt_handler2;
        evtemptyset(&list2.lio_aiocb.aio_event.
                evt_classmask);
        list2.lio_aiocb.aio_flag = AIO_EVENT;
        list2.lio_aiocb.aio_offset = 0;
        list2.lio_aiocb.aio_whence = 0;
        list2.lio_aiocb.aio_prio = 0;

        lcb[0] = &list1;
        lcb[1] = &list2;

        sa.sa_handler = evt_final_handler;
        sa.sa_flags = SA_D9EV;
        sigemptyset(&sa.sa_mask);

        sigaction(EVTCLASS_MIN+2, &sa, NULL);

        final_evt.evt_value = NULL;
        final_evt.evt_class = EVTCLASS_MIN+2;
        final_evt.evt_handler = evt_final_handler;
        evtemptyset(&final_evt.evt_classmask);
        :
        listio(LIO_ASYNC, lcb, 2, &final_evt);
        :
}

void evt_handler1(evt_value, evt_class, evt_mask)
void *evt_value;
evt_class_t evt_class;
evtset_t evt_mask;
{
        :
        :
```

```
}

void evt_handler2(evt_value, evt_class, evt_mask)
void *evt_value;
evt_class_t evt_class;
evtset_t evt_mask;
{
       :
       :
}

void evt_final_handler(evt_value, evt_class, evt_mask)
void *evt_value;
evt_class_t evt_class;
evtset_t evt_mask;
{
       :
       :
}
```

Equivalent POSIX.1b Code

```
#include <aio.h>

#define SIZE 1024

void signal_handler1(int signo, siginfo_t *info,
       void *context);
void signal_handler2(int signo, siginfo_t *info,
       void *context);
void signal_final_handler(int signo, siginfo_t *info,
       void *context);

main()
{
       int fd1, fd2;
       char buf1[SIZE], buf2[SIZE];
       struct aiocb cb1, cb2, *cbs[2];
       struct sigaction sa;
       struct sigevent final_se;
       :
       fd1 = open(......);
       fd2 = open(......);
       :
       sa.sa_sigaction = signal_handler1;
       sa.sa_flags = SA_SIGINFO;
       sigemptyset(&sa.sa_mask);

       sigaction(SIGRTMIN, &sa, NULL);

       sa.sa_sigaction = signal_handler2;
       sa.sa_flags = SA_SIGINFO;
       sigemptyset(&sa.sa_mask);

       sigaction(SIGRTMIN+1, &sa, NULL);
       :
       cb1.aio_sigevent.sigev_signo = SIGRTMIN;
       cb1.aio_sigevent.sigev_value.sival_ptr = NULL;
       cb1.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
       cb1.aio_offset = 0;
```

```
        cb1.aio_reqprio = 0;
        cb1.aio_fildes = fd1;
        cb1.aio_buf = buf1;
        cb1.aio_nbytes = SIZE;
        cb1.aio_lio_opcode = LIO_WRITE;

        cb2.aio_sigevent.sigev_signo = SIGRTMIN+1;
        cb2.aio_sigevent.sigev_value.sival_ptr = NULL;
        cb2.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
        cb2.aio_offset = 0;
        cb2.aio_reqprio = 0;
        cb2.aio_fildes = fd2;
        cb2.aio_buf = buf2;
        cb2.aio_nbytes = SIZE;
        cb2.aio_lio_opcode = LIO_READ;

        cbs[0] = &cb1;
        cbs[1] = &cb2;

        sa.sa_sigaction = signal_final_handler;
        sa.sa_flags = SA_SIGINFO;
        sigemptyset(&sa.sa_mask);

        sigaction(SIGRTMIN+2, &sa, NULL);

        final_se.sigev_signo = SIGRTMIN+2;
        final_se.sigev_value.sival_ptr = NULL;
        final_se.sigev_value.sigev_notify = SIGEV_SIGNAL;
        :
        lio_listio(LIO_NOWAIT, cbs, 2, &final_se);
        :
}

void signal_handler1(signo, info, context)
int signo;
siginfo_t *info;
void *context;
{
        :
        :
}

void signal_handler2(signo, info, context);
int signo;
siginfo_t *info;
void *context;
{
        :
        :
}

void signal_final_handler(signo, info, context)
int signo;
siginfo_t *info;
void *context;
{
        :
        :
}
```

Changes from Draft 9 to POSIX.1b

All Draft 9 functionality has an equivalent in POSIX.1b, but there are differences in the data structure entries and the way parameters are passed to functions.

Table 9-2: Asynchronous I/O Interface

Draft 9	POSIX.1b
`<sys/aio.h>`	`<aio.h>`
`struct liocb`	Provided by structure `aiocb`
`AIO_EVENT`	No Equivalent
`LIO_ASYNC`	No Equivalent
`AIO_PRIO_DFL`	No Equivalent
`AIO_PRIO_MAX`	No Equivalent
`AIO_PRIO_MIN`	No Equivalent
`AIO_LISTIO_MAX`	No Equivalent
No Equivalent	`AIO_PRIO_DELTA_MAX`
`aread()`	`aio_read()`
`awrite()`	`aio_write()`
`listio()`	`lio_listio()`
`acancel()`	`aio_cancel()`
`iosuspend()`	`aio_suspend()`
`afsync()`	`aio_fsync()`
No Equivalent	`aio_error()` Retrieves the error status from an `aiocb` structure
No Equivalent	`aio_return()` Retrieves the return status from an `aiocb` structure

Data Structures

The data structure **`aiocb`** changed a lot from Draft 9 to POSIX.1b. Also, the **`liocb`** data structure has been eliminated because of the changes to `aiocb`.

The following is a comparison of the aiocb data structures for Draft 9 and POSIX.1b.

Table 9-3: aiocb Structure

Draft 9	POSIX.1b
aio_offset	aio_offset
aio_event	aio_sigevent
aio_prio	aio_reqprio
aio_whence	No Equivalent
aio_flag	No Equivalent
aio_errno	No Equivalent
aio_nobytes	No Equivalent
No Equivalent	aio_nbytes
No Equivalent	aio_fildes
No Equivalent	aio_buf
No Equivalent	aio_lio_opcode

POSIX.1b structure includes the file descriptor, buffer, and listio opcode fields. The new **aio_nbytes** field has the same semantics as defined by the **read()** and **write()** synopses. Therefore, the synopses for the **aio** functions have changed as follows:

- File descriptor, buffer, and number of bytes are not passed as separate arguments to aio_read() and aio_write().
- listio opcode is not passed as a separate argument to lio_listio().

The aio_whence field has been eliminated. The aio_offset argument is treated as an offset from the beginning of the file. The effect is as if aio_whence is always SEEK_SET.

The aio_errno field has been eliminated. Instead, a new function, aio_error() with the aiocb argument does the same job.

The aio_flag field has been eliminated. It is superseded by the aio_sigevent field.

The aio_nobytes field has been eliminated. The new aio_return() function retrieves the return status from an aiocb structure. aio_return() can be called only once per structure; this structure may not be passed to aio_error() or aio_return() again.

There is no relation between the new aio_nbytes field and the old aio_nobytes field. The aio_return() function may be used on the same **aiocb** structure more than once, as a proprietary extension from LynuxWorks. This can be disabled by a new proprietary library call aio_setparam(). Refer to the aio_setparam() man page for more information.

Priority of asynchronous I/O can be lowered, but not raised, with respect to the process priority. However, a proprietary feature of LynxOS allows the priority of an asynchronous I/O operation to be raised with respect to the process priority. This is done by the aio_setparam() library call. Refer to the aio_setparam() man page for more information.

Timed Suspension

The Draft 9 iosuspend() function suspends the process until the completion of I/O. The aio_suspend() function from POSIX.1b adds an option for timed suspension. It takes an extra timespec argument for timeout. If this argument is NULL, the behavior is the same as suspension until the completion of I/O.

Cancellation Notification

With the acancel() function from Draft 9, no event notification is given when an asynchronous I/O function is successfully cancelled. However, with the aio_cancel() function from POSIX.1b, normal signal delivery occurs for all asynchronous I/O functions that are cancelled.

listio Signal Delivery

POSIX.1b provides only two mode values as opposed to the three values from Draft 9. The LIO_ASYNC value has been removed. The LIO_NOWAIT argument ensures a final signal delivery, and is equivalent to LIO_ASYNC from Draft 9. If the final signal parameter passed to lio_listio() is NULL, a final signal is not sent. This is equivalent to the LIO_NOWAIT behavior from Draft 9.

aio_fsync()

The POSIX.1b `aio_fsync()` function (equivalent to the Draft 9 `afsync()` function) provides `fsync()` behavior with the `O_SYNC` flag, and `fdatasync()` behavior with the `O_DSYNC` flag. The difference is that for synchronized I/O file integrity completion, the `O_FSYNC` flag is used in Draft 9, while the `O_SYNC` flag is used in POSIX.1b. Refer to the synchronous I/O section for the semantics of these functions.

Interoperability

Asynchronous I/O is fully inter-operable. A process using Draft 9 asynchronous I/O is compatible with a process performing POSIX.1b asynchronous I/O to the same file.

NOTE: Due to a rare condition in the Draft 9 specification, multiple processes accessing a file during asynchronous I/O can produce unexpected results. Avoid using Draft 9 asynchronous I/O if the file will be accessed by multiple processes.

APPENDIX A Functions Callable from Signal Handlers

Because of their asynchronous nature, signals can interrupt any library function, and many system calls. If the signal handler calls the active function again, it may corrupt the state of the library, or fail in some subtle way.

POSIX.1 (POSIX.1b and POSIX.1c) specifies a list of functions that are required to be callable by signal handlers. The following is a list POSIX.1b-specific functions required to be callable by signal handlers.

Table A-1: List of Callable Functions

access()	fdatasync()	read()	tcdrain()
aio_error()	fork()	rename()	tcflow()
aio_return()	fstat()	rmdir()	tcflush()
aio_suspend()	fsync()	sem_post()	tcgetattr()
alarm()	getegid()	setgid()	tcgetpgrp()
cfgetispeed()	geteuid()	setpgid()	tcsendbreak()
cfgetospeed()	getgid()	setsid()	tcsetattr()
cfsetispeed()	getgroups()	setuid()	tcsetpgrp()
cfgetospeed()	getpgrp()	sigaction()	time()
chdir()	getpid()	sigaddset()	timer_getoverrun()
chmod()	getppid()	sigdelset()	timer_gettime()
chown()	getuid()	sigemptyset()	timer_settime()
clock_gettime()	kill()	sigfillset()	times()
close()	link()	sigismember()	umask()
creat()	lseek()	sigpending()	uname()
dup()	mkdir()	sigprocmask()	unlink()

Table A-1: List of Callable Functions (Continued)

dup2()	mkfifo()	sigqueue()	utime()
execle()	open()	sigsuspend()	wait()
execve()	pathconf()	sleep()	waitpid()
_exit()	pause()	stat()	write()
fcntl()	pipe()	sysconf()	

APPENDIX B *Mapping Between Drafts*

This Appendix correlates chapters in the POSIX.4 Draft 9 and POSIX.4 Draft 14 (POSIX.1b) specifications. POSIX.1b chapters are organized differently than those in POSIX.4 Draft 9. The POSIX.1b is now organized as an amendment to the POSIX.1 standard.

Table B-1: Draft 9 to POSIX.1b Chapter Mapping

Draft 9 Chapter	Location
2. Constants and vars	General Section 2, `sysconf`, Section 4, “Process Environment”
	`fpathconf/pathconf` Section 5, “Files and Directories”
3. Binary Semaphores	Semaphores Section 11, “Synchronization” (also `fork/exit/exec` in Section 3)
4. Memory Locking	Memory Locking Section 12, “Memory Management”
5. Shared Memory	Shared Memory Section 12, “Memory Management” (also `fork/exec/exit/close`, Sections 3 and 6)
6. Priority Scheduling	Scheduling Section 13, “Execution Scheduling” (also `fork/exec` in Section 3)
7. Asynchronous Events	Real-Time Signals Extension, Section 3, “Process Primitives”
8. Clocks and Timers	Section 14, “Clocks and Timers”

Table B-1: Draft 9 to POSIX.1b Chapter Mapping (Continued)

Draft 9 Chapter	Location
9. IPC Message Passing	Section 15, "Message Passing" (also `open/fork/exec`, Section 3)
10. Synchronized I/O	Section 6 "Input and Output Primitives"
11. Asynchronous I/O	Section 6 "Input and Output Primitives"
12. Real-Time Files	No Equivalent
No Equivalent	File Mapping (`mmap`) Section 12, "Memory Management" (also `ftruncate`, Section 5, "Files and Directories")

Index

Symbols

A

B

C

D

E

M

N

S

T

U

V

W

Y