



**The Concise Handbook  
of Linux for Embedded  
Real-Time Systems**

version 1.1

# **The Concise Handbook Of Linux for Embedded Real-Time Systems**

TimeSys Corporation

**Version 1.1**

1-888-432-TIME

©2002 TimeSys Corporation  
Pittsburgh, PA

**[www.timesys.com](http://www.timesys.com)**

<b>Embedded Real-Time Systems and Linux</b> .....	<b>5</b>
Managing Real-Time Systems .....	7
Requirements for Embedded Real-Time Systems .....	8
Common Problems with Using Linux for Embedded Real-Time Systems .....	9
Approaches to Designing a Linux-Based Real-Time OS .....	10
<b>About TimeSys Linux™</b> .....	<b>11</b>
TimeSys Linux: an Embedded Real-Time OS Based on Linux .....	12
- <i>What Real-Time Functionality does TimeSys Linux       Offer You?</i> .....	13
- <i>Why Use Linux for Real-Time?</i> .....	15
- <i>TimeSys Linux in Embedded Systems</i> .....	16
- <i>Basic Components of TimeSys Linux</i> .....	17
- <i>Key Features of TimeSys Linux</i> .....	18
Architecture of TimeSys Linux .....	19
- <i>Manually Loading and Unloading LKMs</i> .....	20
Reserves and Resource Sets .....	21
- <i>Behavior and Functioning of Reserves</i> .....	22
- <i>Real Reserves and Resource Guarantees</i> .....	23
Capabilities and Features of TimeSys Linux/Real-Time .....	24
Development Tools for TimeSys Linux .....	25
- <i>TimeWiz®: An Integrated Design and Simulation       Environment for Real-Time Systems</i> .....	26
- <i>TimeTrace®: A Real-Time Profiling Environment</i> .....	27
- <i>TimeStorm™: An Integrated Development       Environment for TimeSys Linux</i> .....	28
- <i>TimeSys Linux Utilities</i> .....	29
- <i>Real-Time Java</i> .....	30
<b>TimeSys Linux System Calls</b> .....	<b>31</b>
System Calls on CPU Reservations .....	32
System Calls on Network Reservations .....	34
System Calls on Resource Sets .....	36
System Calls on Clocks and Timers .....	38
System Calls on Periodic Real-Time Processes .....	39
System Calls on Mutexes .....	40

<b>Real-Time Extensions to POSIX (IEEE 1003.1d)</b> .....	<b>42</b>
Error Values .....	43
Primitive Functions .....	45
Signals and Timers .....	47
Process Identification .....	50
Files and Directories .....	52
Input and Output Primitives .....	56
Terminal Control .....	59
Language-Specific Services .....	61
Synchronization .....	62
Memory Management .....	65
Process Scheduling .....	67
Timers .....	69
Message Passing .....	71
Thread Management .....	73
<b>Reference</b> .....	<b>76</b>
Key Linux Commands .....	77
Key TimeSys Linux Commands .....	80
Glossary of Terms and Concepts .....	81
Some Key References on Real-Time Linux .....	87

# **Embedded Real-Time Systems and Linux**



## Managing Real-Time Systems

A real-time system is one in which it is possible to predict and control when computations take place. In real-time applications, the correctness of a computation depends not only upon its results, but also upon the time at which its outputs are generated. The measures of merit in a real-time system include:

- **Predictably fast response:** The system should respond quickly and predictably to urgent events.
- **High degree of schedulability:** The timing requirements of the system must be satisfied even at high degrees of resource usage.
- **Stability under transient overload:** When the system is overloaded by events, and it is impossible to meet all the deadlines, the deadlines of selected critical tasks must still be guaranteed.

In their pursuit of all of these objectives, real-time systems make use of a variety of scheduling algorithms. These methods of analysis allow engineers to assign priorities to different tasks, then spread the tasks out to ensure that the ones at the highest priority levels *always* meet their deadlines — no matter what else is going on within the system.

# Requirements for Embedded Real-Time Systems

Real-time software is most commonly found in embedded systems. Since the line between the operating system and the application in these computers is often blurred, embedded systems that must function under time constraints generally require that the operating system itself have real-time capabilities.

Other requirements for embedded systems may include:

- Small operating system footprint
- Diskless and/or headless operation
- Flash bootability
- Remote bootability
- Power monitoring and battery backup
- Integrated A/D, D/A, and DSP capabilities
- Power, reliability, safety, security, and maintainability constraints

The wide range of physical differences among embedded systems means that a successful system often will also incorporate one or more of the following features:

- Board support packages (BSPs) for specific hardware
- Chassis mount
- A wide range of processor power, from 8-bit microcontrollers to 64-bit processors
- Low cost



## **Common Problems with Using Linux for Embedded Real-Time Systems**

Linux has many strengths as an operating system. Yet those developing real-time systems with Linux must also overcome a number of problems with the standard Linux components, including:

- Limited number of fixed priority levels
- No support for priority inheritance
- Limited QoS (Quality of Service) support
- Lack of support for high-resolution timers
- No support for periodic tasks
- Potentially non-preemptible kernel with possibly long system calls
- Limited support for non-desktop systems

Commonly, extensions and additions to Linux are developed to handle the above problems.

# Approaches to Designing a Linux-Based Real-Time OS

Real-time approaches to Linux have generally fallen into four major categories. These categories are:

1. Adding a new kernel below the Linux layer. RT-Linux from NMT and Real-Time Applications Interface (RTAI) from Milan Polytechnic typify this approach.

## **Benefits**

- Very high performance
- Smaller footprint
- Open source

## **Disadvantages**

- Task failure leads to system crash
- Requires custom device drivers
- Custom API for applications

2. Extending the existing kernel to provide real-time capabilities. TimeSys Linux exemplifies this approach.

## **Benefits**

- Embedding support
- Real-Time POSIX extensions
- QoS guarantees
- Uses Linux drivers as is
- Uses all Linux applications, utilities, and compilers
- Support for Real-Time Java™
- Excellent timer resolution
- Open source

## **Disadvantages**

- Performance not as high as Category I

3. Adding an OS server on top of a real-time microkernel. Running MK-Linux on the Mach microkernel is an example of this approach. For example, native Mach and Linux programs can both run simultaneously in such an environment.

4. Adding Linux binary compatibility to an existing RTOS.

# **About TimeSys Linux™**

## **TimeSys Linux™: an Embedded Real-Time OS Based on Linux**

TimeSys Linux™ is a Linux-based real-time operating system designed primarily for embedded systems. Linux meets the OS needs of many domains, including:

- Telecommunications systems
- Consumer electronics
- Automotive systems
- ISPs
- Multimedia and Web servers
- Medical electronics
- Process control
- Industrial automation
- Defense systems
- Avionics

## What Real-Time Functionality does TimeSys Linux Offer You?

TimeSys Linux is a complete real-time operating system with a full range of capabilities. The TimeSys Linux programmer benefits from such features as:

- **Real real-time Linux applications:** Any Linux process can now become a real-time process. You are no longer constrained to choose between a real-time OS and Linux. You do not have to embed a thin real-time OS layer below the Linux kernel; you just use Linux processes as is and grant them real-time capabilities as you wish.
- **POSIX (Portable Operating System Interface) support for your real-time needs:** TimeSys Linux provides complete support for the traditional real-time systems paradigm of using a fixed-priority preemptive scheduling policy. In fact, it supports 2048 priority levels. It also supports priority inheritance on mutexes to avoid the unbounded priority inversion problem.
- **QoS (Quality of Service) delivery:** TimeSys Linux provides direct and explicit support for QoS delivery to your real-time applications using the notation of CPU and network reservations (also called reserves).
- **Real-time support for legacy applications:** A convenient feature is that you can take existing legacy applications running on Linux and endow them with QoS guarantees.

## TimeSys Linux vs. Standard Linux

	TimeSys Linux	Standard Linux
Footprint	<b>500KB - 1.2MB</b>	<b>1 - 2MB</b>
Scheduler	<b>Fine-grained</b> (2048 priority levels) <b>Enforced CPU Reservations</b>	<b>Standard UNIX Scheduling</b>
Timer Resolution	<b>Geared To Clock Resolution</b> (10 $\mu$ s or less)	<b>Very Low</b> (10 ms or more)
Tools for Embedded/RT Support	<b>Comprehensive</b> TimeWiz TimeTrace TimeStorm	<b>Limited</b>
POSIX Real-Time Functionality	<b>Yes</b>	<b>No</b>
Priority Inheritance	<b>Yes</b>	<b>No</b>
Periodic Tasks	<b>Yes</b>	<b>No</b>

## Why Use Linux for Real-Time?

The power and features of Linux make it a natural base for real-time operating systems. The open source development model means that any Linux code is freely available for anyone to use and contribute to, and has led to an explosion of Linux development. Linux programmers can take advantage of the knowledgeable user base and multitude of software associated with Linux, as well as the years of fine-tuning that mean Linux can be trusted to run smoothly.

The chart below lays out some of the benefits of choosing Linux for your real-time operating system:

<b>TIMEsys LINUX VS. STANDARD RTOS</b>		
	<b>TimeSys Linux</b>	<b>Standard RTOS</b>
Run-Time Licenses	<b>Free</b>	<b>Expensive</b> (10¢ - \$50.00 per license)
Source Code License	<b>Free</b>	<b>Expensive</b> (>\$200,000 per seat)
Device Drivers	<b>50,000 +</b>	<b>5 to 5,000</b>
Applications and Utilities	<b>100,000 +</b>	<b>Limited</b>
Developers Exposed to API	<b>100,000 +</b>	<b>5,000 to 50,000</b>

## **TimeSys Linux in Embedded Systems**

TimeSys Linux was designed with the requirements of embedded systems in mind. The small footprint and reliability of TimeSys Linux make it a good choice for embedding in a wide range of devices, from small appliances to sophisticated transportation or defense systems. For more information about customizing TimeSys Linux for your embedded system, contact TimeSys or visit our website at **[www.timesys.com](http://www.timesys.com)**.

TimeSys prepares specialized Linux distributions that are customized for the requirements and configuration of specific embedded development boards. These customized Linux distributions are known as Board Support Packages (BSPs).

A wide and continuously growing list of BSPs is available. Support for Pentium, PowerPC, ARM, StrongARM, and XScale process boards already exists. Other than the X86 desktop distribution, COMPACT PCI, VME, VME64, and PC104 backplanes are supported. Ports to S-H, MIPS, and NEC processors are under way.

TimeSys Linux can work with any standard Linux distribution, including Red Hat, Debian, SuSE, Linux-Mandrake, and TurboLinux.



## Basic Components of TimeSys Linux

TimeSys has incorporated a critical set of components into its Linux offering that, together, offer a highly innovative approach to meeting time constraints. These components can be combined in some critical ways to handle a wide variety of application requirements. The basic components of TimeSys Linux are:

- Linux kernel (TimeSys Linux™)
- Real-time extensions (TimeSys Linux/Real-Time™)
- CPU reservation modules (TimeSys Linux/CPU™)
- Network reservation modules (TimeSys Linux/NET™)
- TimeTrace™

## Key Features of TimeSys Linux

TimeSys Linux adds numerous features to the standard Linux base, resulting in a system with impressive real-time capabilities. Below are some of the most distinctive characteristics of TimeSys Linux.

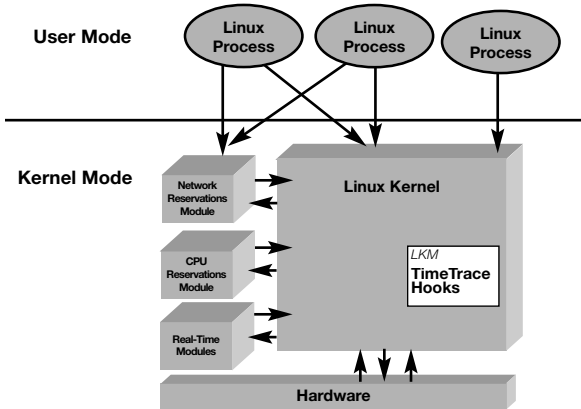
- TimeSys Linux differs from other real-time Linux systems in that the core kernel itself is modified to handle real-time applications. This means, among other things, that if a single real-time process crashes, the other processes and the kernel will continue safely along as if nothing happened.
- The TimeSys Linux implementation offers extremely accurate time management. The combination of a timestamp counter with a high-resolution timer contributes to significant improvement in the precision of resource management.
- QoS support is available in the form of CPU reservations.
- Support for 2048 priority levels co-exists with QoS and reservation guarantees.
- Support for priority inheritance in mutexes helps avoid unbounded priority inversion problems.
- Support for very-high-resolution timers and clocks.
- Support for periodic tasks is available.
- The Linux kernel supports the **proc** filesystem, which provides a consistent view of the current status of Linux kernel and running processes. TimeSys Linux uses the **proc** filesystem for providing information on the hardware platform and the status of resource sets and associated reservations.

# Architecture of TimeSys Linux

In the TimeSys Linux operating system, the code that provides support for real-time capabilities and for CPU and network reservations resides in modules — binary code that can be inserted into the main kernel for added functionality. This reliance on modules enables TimeSys Linux to include all the features necessary for a complete real-time operating system while keeping the size of the kernel as small as possible.

The loadable kernel modules (LKM) used in TimeSys Linux are object modules that can be inserted into or removed from the kernel when you boot your system or whenever they are required by an application. From the application's point of view, the system calls made available by these modules are indistinguishable from the system calls in the kernel itself, since the module is run in kernel (as opposed to user) mode.

The diagram shown above illustrates how the different components of TimeSys Linux fit together.



## Manually Loading and Unloading LKMs

To load a TimeSys Linux module, all you need to do is type:

```
insmod <module>
```

**insmod** comes with a number of flags and options:

- f** Attempt to load the module, even if the version of the kernel currently running and the version for which the module was compiled do not match.
- k** Auto-clean; remove modules that have not been used in some period of time, usually one minute.
- m** Output a load map, making it easier to debug the module in the event of a kernel panic.
- o module\_name** Explicitly name the module, rather than deriving the name from the base name of the source object file.
- p** Probe the module to see if it could be successfully loaded. This includes locating the object file in the module path, checking version numbers, and resolving symbols.
- s** Output everything to **syslog** instead of the terminal.
- v** Be verbose.

To unload a module, type:

```
rmmod module_name
```

**rmmod** also supports a few flags:

- a** Remove all unused modules.
- s** Output everything to **syslog** instead of the terminal.

## Reserves and Resource Sets

The primary abstractions behind the TimeSys Linux/Real-Time, TimeSys Linux/CPU, and TimeSys Linux/NET modules are the *resource capacity reservations* (*reserves* for short) and the *resource set*. TimeSys Linux/Real-Time adds these features to the standard Linux operating system to provide better controls over resource allocation, scheduling, and usage accounting.

A reserve represents a share of a single computing resource. Such a resource can be CPU time (as in TimeSys Linux/CPU), network processing bandwidth (as in TimeSys Linux/NET), physical memory pages, or a disk bandwidth. A certain amount of a resource is reserved for use by the programs. A reserve is implemented as a kernel entity; thus, it cannot be counterfeited. The kernel keeps track of the use of a reserve and will enforce its utilization when necessary. Appropriate scheduling and enforcement of a reserve by the resource kernel guarantees that the reserved amount is always allocated for it.

A resource set represents a set of reserves. A resource set is bound to one or more programs, and provides those programs with the exclusive use of its reserved amount of resources. A resource set groups together the resources that are necessary for an application to do its job; thus, it is easy to examine and compare the utilization of each resource in it. If the kernel or a QoS manager finds an imbalance in resource utilization, an application will be notified and will be able to change its QoS parameters in order to balance the utilization.

## Behavior and Functioning of Reserves

When a reserve uses up its allocated time units within an interval, it is said to be *depleted*. A reserve that is not depleted is said to be an *undepleted* reserve. At the end of the current interval, the reserve obtains a new quota and is said to be *replenished*. In our resource management model, the behavior of a reserve between depletion and replenishment can take one of two forms:

- **Hard reserves** - not scheduled for execution on depletion until they are replenished.
- **Soft reserves** - can be scheduled for execution on depletion without restriction

Reserves contain certain amounts of resources and control their utilization. A reserve may represent one of many different types of resources, such as CPU cycles, network bandwidth, or others. Different types of resources have their own accounting information and their own ways to deal with resource management. At the same time, reserves need to provide a uniform interface; otherwise, modifications are required each time a new resource type is added.

Each reserve can be broken down into two parts, each geared towards satisfying one of these two needs:

- An **abstract reserve** implements the functionality common across all reserves and provides a uniform interface.
- A **real reserve** implements resource-type-specific portions and exports functions that adhere to the uniform resource management interface.

Abstract and real reserves are always paired. When a reserve is created, both components are formed and are coupled with each other. The distinction is useful because only real reserves need to be implemented in the creation of a new resource type.

## Real Reserves and Resource Guarantees

Real reserves implement the following mechanisms to guarantee resource utilization based on reservation.

- **Admission control:** TimeSys Linux performs an admission control test on a new request to determine if it can be accepted or not. If the request can be admitted, TimeSys Linux creates a reserve based on the requested parameters.
- **Scheduling policy:** A scheduling policy controls dynamic resource allocation so that an application can receive its reserved amount of a resource.
- **Enforcement:** TimeSys Linux enforces the use of a resource by an application based on its allocated reserves. An enforcement mechanism prevents a resource from being used more than its reserved amount.
- **Accounting:** TimeSys Linux tracks how much of a resource an application has already used. This information is used by the scheduling policy and the enforcement mechanism. An application, such as a QoS manager or a real-time visualization tool, can also query this information for observation and/or dynamic resource allocation control purposes.

# Capabilities and Features of TimeSys Linux

TimeSys Linux provides the following capabilities:

- **Fixed-priority scheduling with 2048 priority levels:** You can use the standard POSIX-compliant calls to assign a priority to any Linux process.
- **Priority inheritance to avoid unbounded priority inversion:** Timing problems from potentially unbounded priority inversion can be eliminated by the use of priority inheritance protocols using the Real-Time POSIX threads library and kernel support provided by TimeSys Linux/Real-Time. The APIs used by TimeSys are the same as POSIX in this regard.
- **Quality of Service (QoS) support for resource reservation:** TimeSys Linux, through the CPU and NET modules, provides direct support for delivering guaranteed Quality of Service (QoS) to your real-time applications. An application can explicitly request and obtain CPU and timing guarantees.
- **High-resolution clocks and timers:** TimeSys Linux/Real-Time supports high-resolution clocks and timers. Resolutions of a few microseconds or better are available.
- **Periodic real-time tasks:** Periodic execution of tasks is a common requirement in real-time systems. TimeSys Linux/Real-Time allows Linux processes to be marked as periodic processes, in which case they will be executed in periodic fashion.
- **Memory wiring:** TimeSys Linux/Real-Time can “lock” the physical memory pages of a real-time process so that they are not swapped out by the paging system. The predictability of real-time processes can suffer significantly without this feature.

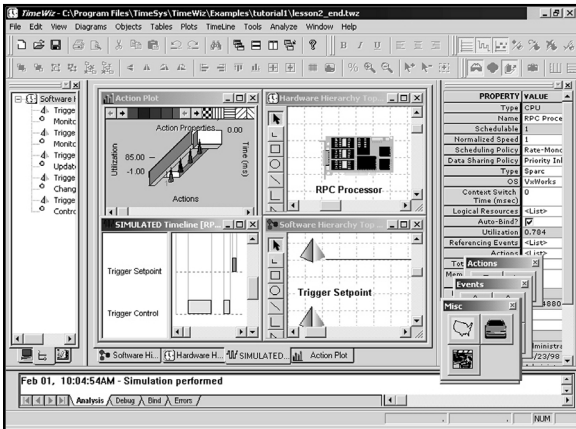


## Development Tools for TimeSys Linux

TimeSys has developed a complete line of tools for real-time analysis. TimeSys Linux supports the following TimeSys tools:

- **TimeWiz®**: a sophisticated system modeling, analysis, and simulation environment for real-time systems. A special version of TimeWiz customized for Rational Rose is also offered.
- **TimeTrace®**: provides the critical instrumentation needed to see inside your real-time system, collecting all the necessary timing data essential to the successful application of rate-monotonic analysis and average-case simulation studies.
- **TimeStorm™**: is a fully-featured IDE that lets you edit, compile, download, and debug TimeSys Linux programs on a remote Windows system.

## TimeWiz®: An Integrated Design and Simulation Environment for Real-Time Systems

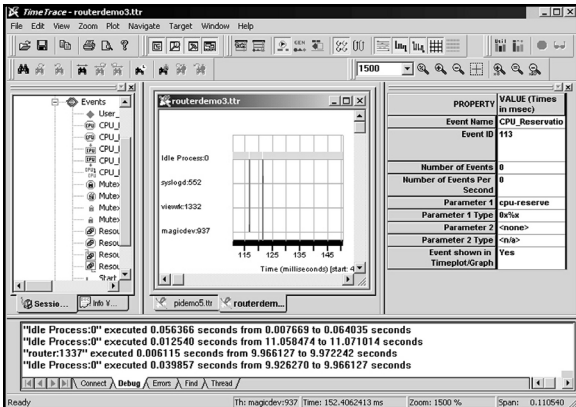


TimeWiz® is a TimeSys Corporation product specifically designed for the construction of simple or complex real-time systems with predictable timing behavior.

TimeWiz lets you:

- Represent your hardware and software configurations visually.
- Analyze the worst-case timing behavior of your system.
- Simulate its average-case timing behavior.
- Model processors and networks for end-to-end performance.
- Chart your system parameters and generate integrated system reports.

## TimeTrace®: A Real-Time Profiling Environment



PROPERTY	VALUE (Times in msec)
Event Name	CPU_Reservation
Event ID	113
Number of Events	0
Number of Events Per Second	0
Parameter 1	cpu-reserve
Parameter 1 Type	bool
Parameter 2	<none>
Parameter 2 Type	<n/a>
Event shown in Timeplot/Graph	Yes

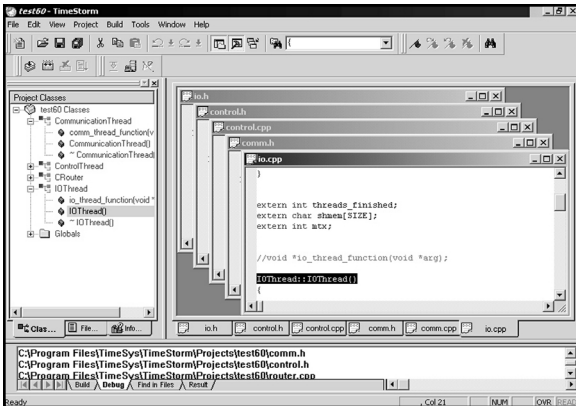
```
"Idle Process:0" executed 0.056366 seconds from 0.007669 to 0.064035 seconds
"Idle Process:0" executed 0.012540 seconds from 11.058474 to 11.071014 seconds
"router:1337" executed 0.006115 seconds from 9.966127 to 9.972242 seconds
"Idle Process:0" executed 0.039857 seconds from 9.926270 to 9.966127 seconds
```

TimeTrace® is a productivity enhancement tool from TimeSys Corporation that lets you profile your real-time OS target in real-time.

With TimeTrace, you can:

- Capture execution sequence on targets efficiently.
- Display target execution sequences visually to create a “software oscilloscope.”
- Monitor multiple targets simultaneously from a single workstation.
- Feed TimeTrace data into TimeWiz as execution time and period parameters for worst-case analysis and/or average-case simulation.

## TimeStorm™: An Integrated Development Environment for TimeSys Linux



TimeStorm™ is a gcc-based integrated development environment (IDE) that allows you to create, compile, and debug TimeSys Linux applications on a remote system.

With TimeStorm, you can:

- Write and edit code with a powerful editor that features search-and-replace functionality as well as language-specific syntax highlighting.
- Debug your applications with gdb.
- Navigate your project easily with control trees that let you view every file or every class, method, and variable in your project.
- Export applications to a variety of embedded systems running TimeSys Linux.
- Develop TimeSys Linux applications in a familiar, Windows-based environment.

## TimeSys Linux Utilities

Besides such standard day-to-day Linux utilities as **make** and **gcc**, TimeSys Linux comes with an assortment of utilities that allow you to manipulate resource sets and other basic real-time concepts.

These utilities include:

- **clockfreq:** Allows you to retrieve the processor clock frequency at which the CPU is running.
- **rkattach:** Allows you to attach a process (specifying its pid) to an existing resource set. Remember to specify the resource set id using the hexadecimal format. You can attach any Linux process using this utility, even if the process was written without any knowledge of RK.
- **RKcleanRS:** A shell script that destroys all resource sets and their reserves in the processor.
- **rkdestroy:** Allows you to destroy a resource set (whose id is specified using the hexadecimal format).
- **rkdetach:** Allows you to detach a process (specified by its pid) from an existing resource set.
- **rkexec:** Allows you to create a new resource set with CPU reservation parameters and attach a new process to the resource set. Again, this allows any legacy process (written without any knowledge of TimeSys Linux) to be able to use and benefit from the Quality of Service guarantees provided by TimeSys Linux.
- **rklist:** Lists the current resource sets in the system and their parameters.

## Real-Time Java

The Real-Time Specification for Java (RTSJ), completed in 2001 under Sun Microsystems' Java Community Process, meets the need for a truly platform-independent real-time programming language. The RTSJ allows TimeSys Linux users to take advantage of real-time extensions to Java that include the following:

- Real-time threads. These threads offer more carefully defined scheduling attributes than standard Java threads.
- Tools and mechanisms that let developers write code that does not need garbage collection.
- Asynchronous event handlers, and a mechanism that associates asynchronous events with happenings outside the JVM.
- Asynchronous transfer of control, which provides a carefully controlled way for one thread to throw an exception into another thread.
- Mechanisms that let the programmer control where objects will be allocated in memory and access memory at particular addresses.

TimeSys developed the reference implementation for the RTSJ, which is available at [www.timesys.com](http://www.timesys.com). Further RTSJ information is available at [www.rttj.org](http://www.rttj.org).

# **TimeSys Linux System Calls**

## System Calls on CPU Reservations

**int rk\_cpu\_reserve\_create (rk\_resource\_set\_t rs, rk\_reserve\_t \*rs, cpu\_reserve\_attr\_t attr);**

Creates a CPU reservation and attaches to resource set rs. The amount of CPU reservation is specified with struct cpu reserve attr (defined in <rk/rk.h>). It permits the definition of computation time (C), period (T), deadline (D), blocking time (B, typically 0), and enforcement mode (hard or soft). Currently, TimeSys Linux supports **RSU HARD** and **RSU SOFT**.

- **RSU HARD:** guaranteed to receive the specified amount on success
- **RSU SOFT:** guaranteed to receive the specified amount on success. If resource is still available after using up guaranteed amount, it will compete against unreserved tasks for more.

**int rk\_cpu\_reserve\_ctl (rk\_resource\_set\_t rs, cpu\_reserve\_attr\_t attr);**

Changes the properties of existing CPU reservations (computation time, period, deadline, blocking time and enforcement mode). Typically, a failure return indicates that admission control has failed. In that case, the original values of the CPU reserve are restored and the reserve continues to be valid.

**int rk\_cpu\_reserve\_delete(rk\_resource\_set\_t rs);**

Deletes the CPU reserve associated with a resource set.

**int rk\_cpu\_reserves\_get\_num(void);**

Returns the number of CPU reserves currently in the system. This function is normally followed by a **rk\_cpu\_reserves\_get\_list** system call.

**int rk\_cpu\_reserves\_get\_list(rk\_reserve\_t \*buff, int size);**

Returns the list of CPU reserves in the system; count is the size in bytes of the buffer; **buff** must point to a buffer of sufficient size to hold all current CPU reserves. The number of resource sets stored into the **\*buff** buffer is returned. This system call is typically preceded by the **rk cpu reserves get num( )** system call. The number of CPU reserves actually stored in the **\*buff** buffer is returned by the call.



**int rk\_cpu\_reserve\_get\_attr(rk\_reserve\_t rsv, cpu\_reserve\_attr\_t attr);**

Returns the attributes of the specified CPU reserve rsv, which include the reserve's computation time (C), period (T), deadline (D), blocking time (B, typically 0), and enforcement mode (hard or soft).

**void rk\_inherit\_mode (int mode);**

Determines whether children created by this process inherit the resource set of the parent process. A mode value of 0 clears the flag, and a non-zero value sets the flag. This functionality enables programs like **make** to bind themselves and all their children like **gcc**, **ld**, etc. to the same fixed resource reservation.

## System Calls on Network Reservations

**int rk\_net\_reserve\_create (rk\_resource\_set\_t rs, rk\_reserve\_t \*rsu, net\_reserve\_attr\_t attr);**

Creates a network reservation with the specified parameters and attaches it to the specified resource set.

**int rk\_net\_reserve\_ctl (rk\_resource\_set\_t rs, net\_reserve\_attr\_t attr);**

Changes the behavior of the network reserves in the specified resource set to the behavior in the specified **net\_reserve\_attr\_t**.

**int rk\_net\_reserve\_delete (rk\_resource\_set\_t rs);**

Delete the network reserve associated with the specified resource set.

**int rk\_net\_reserve\_get\_attr (rk\_reserve\_t rsu, net\_reserve\_attr\_t attr);**

Fills in a **net\_reserve\_attr\_t** struct with the values applicable to the specified network reserve.

**int rk\_net\_reserves\_get\_list (rk\_reserve\_t \*buff, int size);**

Puts a list of the system's current network reserves into the **buff** buffer.

**int rk\_net\_reserves\_get\_num ();**

Returns the number of network reserves currently in the system.

**int rk\_nettr\_reserve\_create (rk\_resource\_set\_t rs, rk\_reserve\_t \*rsu nettr\_reserve\_attr\_t attr);**

Creates a network reservation with attributes.

**int rk\_nettr\_reserve\_ctl (rk\_resource\_set\_t rs, net\_reserve\_attr\_t attr);**

Changes the behavior of the network reserves attached to **rs** to the behavior defined in **attr**.

**int rk\_nettr\_reserve\_delete (rk\_resource\_set\_t rs);**

Deletes the network reserve associated with the resource set **rs**.

```
int rk_netreserve_get_attr (rk_reserve_t rsv,  
net_reserve_attr_t attr);
```

Reads the attributes of **rsv** and returns them in **attr**.

```
int rk_netreserves_get_list (rk_reserve_t *buff, int size);
```

Puts a list of the system's current network reserves into the buffer **buff**. A maximum of **size** bytes will be returned.

```
int rk_netreserves_get_num ();
```

Returns the number of network reserves currently in the system.

## System Calls on Resource Sets

**rk\_resource\_set\_t rk\_proc\_get\_rset (pid\_t pid);**

Returns the resource set associated with the process **pid**.

**int rk\_resource\_set\_attach\_process(rk\_resource\_set\_t rs,  
pid\_t pid);**

**int rk\_resource\_set\_detach\_process(rk\_resource\_set\_t rs,  
pid\_t pid);**

Attaches the process identified by **pid** to and detaches it from the resource set.

**int rk\_resource\_set\_attach\_socket (rk\_resource\_set\_t rs, int  
socket\_fd, rk\_reserve\_type\_t reserve\_type);**

**int rk\_resource\_set\_detach\_socket (rk\_resource\_set\_t rs, int  
socket\_fd, rk\_reserve\_type\_t reserve\_type);**

The **rk\_resource\_set\_attach\_socket** system call attaches the socket **socket\_fd** to the resource set **rs**. The socket will be used by the network reserve of type **reserve\_type**, which must have been previously attached to the resource set.

The **rk\_resource\_set\_detach\_socket** call detaches the socket **socket\_fd** from the network reserve of type **reserve\_type**.

**rk\_resource\_set\_t rk\_resource\_set\_create (char \*name);**

Creates a resource set and associates the label **name** with it.

**int rk\_resource\_set\_destroy (rk\_resource\_set\_t rs);**

Destroys the resource set **rs**. All reserves in the resource set are deleted and any attached processes are detached.

**rk\_reserve\_t rk\_resource\_set\_get\_cpu\_rsv (rk\_resource\_set\_t rs);**

Returns the cpu reserve associated with the resource set **rs**.

**int rk\_resource\_set\_get\_name (rk\_resource\_set\_t rs, char \*name);**

**int rk\_resource\_set\_set\_name (rk\_resource\_set\_t rs, char \*name);**

The **rk\_resource\_set\_get\_name** system call gets the name of the resource set **rs** and copies it to **name**.

The **rk\_resource\_set\_set\_name** system call assigns the string **NAME** as the new name of the resource set **rs**.

**rk\_reserve\_t rk\_resource\_set\_get\_net\_rsv (rk\_resource\_set\_t rs);**  
Returns the net reserve associated with the resource set **rs**.

**rk\_reserve\_t rk\_resource\_set\_get\_net\_rsv (rk\_resource\_set\_t rs);**  
Returns the netr reserve associated with the resource set **rs**.

**int rk\_resource\_set\_get\_num\_procs (rk\_resource\_set\_t rs);**  
Returns the number of processes attached to the resource set **rs**.

**int rk\_resource\_set\_get\_proclist (rk\_resource\_set\_t rs, pid\_t \*procs, int count);**  
Examines the resource set **rs** and creates a list of process IDs in **procs** corresponding to the processes currently attached to the resource set. The **procs** buffer must have space for **count** pids. Typically the size required for the **procs** buffer is determined by calling **rk\_resource\_set\_get\_num\_procs** first to determine the number of processes attached to **rs**.

**int rk\_resource\_sets\_get\_list (rk\_resource\_set\_t \*rs, int count);**  
Scans the system for resource sets and creates a list in the buffer pointed to by **rs**. There must be room for **count** bytes in this buffer.

**int rk\_resource\_sets\_get\_num ();**  
Returns the number of resource sets currently defined in the system.

**int rk\_signal\_reserve\_enforce (pid\_t pid, rk\_reserve\_t rsv, sigevent\_t \*evp);**  
Allows a process to be registered to receive a signal when a reserve is enforced. If the signal pointer **evp** is **NULL**, the process will be unregistered and will no longer receive the signal.

## System Calls on Clocks and Timers

**unsigned long rt\_get\_clock\_frequency ();**

Returns the system clock frequency in Hz. On some architectures (x86), this is the same as the processor clock frequency. On other architectures, the system clock runs at a different frequency than the clock.

**int clock\_getres (clockid\_t clock\_id, const struct timespec \*res);**

Gets the resolution of the clock **clock\_id** and places it in **res**.

**int clock\_gettime (clockid\_t clock\_id, struct timespec \*tp);**

**int clock\_settime (clockid\_t clock\_id, const struct timespec \*tp);**

The **clock\_gettime** call will get the current time of the clock **clock\_id** and place it in **tp**.

The **clock\_settime** call will set the current time of the clock **clock\_id** with the value in **tp**.

**int timer\_create (clockid\_t clock\_id, struct sigevent \*evp, timer\_t \*timerid);**

create a timer on clock **clock\_id** that will raise event **evp** every expiry, and place the timer's id in **timerid**.

**int timer\_delete (timer\_t timerid);**

Deletes the timer specified by **timerid**.

**int timer\_getoverrun (timer\_t timerid);**

Returns the number of times the timer **timerid** has expired since the most recently generated signal was delivered. If no signal has been delivered, then the results are undefined.

**int timer\_gettime (timer\_t timerid, struct itimerspec \*value);**

Returns the amount of time remaining until the timer **timerid** will expire and the reload value in the location specified by **value**.

**int timer\_settime (timer\_t timerid, int flags, const struct itimerspec \*value, struct itimerspec \*ovalue);**

Sets the start and interval values for the timer **timerid** to the values contained in **value** and copy the old values into **ovalue** if **ovalue** is not **NULL**.

## System Calls on Periodic Real-Time Processes

**int rt\_make\_periodic(struct timespec \*period, struct timespec \*start);**

The calling thread is made periodic with the specified period parameter and its period will begin at time start, which represents an absolute point in time. The calling task will typically call the function **rt\_wait\_for\_start\_time**, described next, after calling this function.

This is not persistent across exec and fork system calls.

**int rt\_wait\_for\_start\_time(void);**

This function is called by a periodic task and allows the task to be delayed until the point in time when its periodicity starts. If the start time has already elapsed, the function will return immediately. This function is typically preceded by a call (but only once) to the **rt\_make\_periodic** function described above.

**int rt\_wait\_for\_next\_period(void);**

This function is called by a periodic task to wait for its next (possible) period boundary. The task is blocked until the next boundary.

**int rt\_process\_get\_period(pid\_t pid, struct timespec \*period);**

Obtains the period in the structure **period** of the real-time periodic process specified by **pid**.

**int rt\_process\_set\_period(pid\_t pid, struct timespec \*period);**

Sets the period of the real-time periodic process specified by **pid** to the period value in the structure **period**.

**int rk\_cpu\_reserves\_get\_scheduling\_policy(void);**

(Deadline monotonic scheduling is generally preferable in this context because of the higher schedulability it offers).

## System Calls on Mutexes

**int rt\_mutex\_create (int proto, int gid);**

**int rt\_mutex\_destroy (int mutex\_id);**

The **rt\_mutex\_create** call will create a mutex where **proto** specifies which priority inheritance model to use, and **gid** specifies which threads may lock this mutex.

The **rt\_mutex\_destroy** call will destroy the mutex referenced by **mutex\_id**.

**int rt\_mutex\_lock (int mutex\_id);**

**int rt\_mutex\_trylock (int mutex\_id);**

**int rt\_mutex\_unlock (int mutex\_id);**

The **rt\_mutex\_lock** call will attempt to lock the mutex **mutex\_id**. If the mutex is already locked the calling thread will block and, depending on what protocol for priority inheritance was specified at mutex creation, the process that holds the lock on the mutex may have its priority adjusted to run at a higher priority than normal while the mutex is held.

The **rt\_mutex\_trylock** call will attempt to lock the mutex **mutex\_id**. If the mutex is already locked, this call returns immediately. If successful, the same effects on priority occur as if it were **rt\_mutex\_lock**.

The **rt\_mutex\_unlock** call will unlock the mutex **mutex\_id**. The mutex may only be unlocked by the process that locked it, and any effects on priority caused by locking this mutex are reversed.

**int rt\_signal\_deadline (pid\_t pid, struct timespec \*deadline, sigevent\_t \*evp);**

Allows a periodic process to specify a deadline time and register itself or another process to receive a signal when it misses that deadline. A process is considered to have missed its deadline if it doesn't call **rt\_wait\_for\_next\_period** before its deadline time. The process must be periodic, i.e., it must have called **rt\_make\_periodic** prior to calling **rt\_signal\_deadline**.



```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

```
int pthread_mutexattr_setprotocol (pthread_mutexattr_t *attr,  
int protocol);
```

```
int pthread_mutexattr_getprotocol (pthread_mutexattr_t  
*attr, int *protocol);
```

```
int pthread_mutexattr_setgid (pthread_mutexattr_t *attr, int gid);
```

```
int pthread_mutexattr_getgid (pthread_mutexattr_t *attr, int *gid);
```

The **pthread\_mutexattr\_init** call initializes an attribute structure for a mutex. This must be done before that attribute structure is used in a **pthread\_mutex\_init (3)** call at which point the actual attributes of a mutex are fixed. **attr** is a pointer to a previously declared mutex attribute structure.

**pthread\_mutexattr\_init** disables all dynamic priority protocols such as priority inheritance and priority ceiling, and allows access only to threads who are members of the same process group as the thread that initialized the mutex.

**pthread\_mutexattr\_destroy** is included for completeness. It currently performs no useful function and simply returns 0.

**Real-Time Extensions  
to POSIX (IEEE 1003.1d)**

## Error Values

The variable **errno** can be used to determine the cause of a function failure. A function can set the variable to one of a number of values if it returns an indicator of failure.

<b>E2BIG</b>	Number of bytes used by argument and environment list exceeds the system imposed limit
<b>EACCES</b>	Permission denied
<b>EAGAIN</b>	Resource unavailable
<b>EBADF</b>	Bad file descriptor
<b>EBADMSG</b>	Bad message
<b>EBUSY</b>	Resource busy
<b>ECHILD</b>	No child process
<b>EDEADLK</b>	Resource deadlock avoided
<b>EDOM</b>	Domain error
<b>EEXIST</b>	File exists
<b>EFAULT</b>	Bad address
<b>EFBIG</b>	File too large
<b>EINPROGRESS</b>	Operation in progress
<b>EINTR</b>	Interrupted function call
<b>EINVAL</b>	Invalid argument
<b>EIO</b>	I/O error
<b>EISDIR</b>	Is a directory
<b>EMFILE</b>	Too many open files
<b>EMLINK</b>	Too many links

<b>EMSGSIZE</b>	Inappropriate message buffer length
<b>ENAMETOOLONG</b>	Filename too long
<b>ENFILE</b>	Too many open files in the system
<b>ENODEV</b>	No such device
<b>ENOENT</b>	No such file nor directory
<b>ENOEXEC</b>	Exec format error
<b>ENOLCK</b>	No locks available
<b>ENOMEM</b>	Not enough memory
<b>ENOSPC</b>	Not enough space on device
<b>ENOSYS</b>	Function not available on current implementation
<b>ENOTDIR</b>	Not a directory
<b>ENOTEMPTY</b>	Directory not empty
<b>ENXIO</b>	No such device or address
<b>EPERM</b>	Operation not permitted
<b>EPIPE</b>	Broken pipe
<b>ERANGE</b>	Result too large
<b>EROFS</b>	Read-only file system
<b>ESPIPE</b>	Invalid seek
<b>ESRCH</b>	No such process
<b>ETIMEDOUT</b>	Operation timed out
<b>EXDEV</b>	Improper link

## Primitive Functions

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

### Process creation, execution, and termination:

```
#include <sys/types.h>
pid_t fork ( void );
```

*returns:*

0	to child process upon success
<i>process ID</i>	to parent process upon success
<i>error no.</i>	on failure

*errno:*

**EAGAIN**  
**ENOMEM**

```
int execl (const char *path, const char *arg, ...);
int execv (const char *path, char *const argv[]);
int execl (const char *path, const char *arg, ...);
int execve (const char *path, char *const argv[], char *const envp[]);
int execlp (const char *file, const char *arg, ...);
int execvp (const char *file, char *const argv[]);
```

*errno:*

**E2BIG**  
**EACCES**  
**ENAMETOOLONG**  
**ENOENT**  
**ENOTDIR**  
**ENOEXEC**  
**ENOMEM**

```
int pthread_atfork(void (*prepare)(void), void (*parent) (void),
void (*child)(void));
```

*errno:*

**ENOMEM**

```
#include <sys/types.h>
#include <sys/wait.h>
```

```
pid_t wait( int *stat_loc);  
pid_t waitpid( pid_t pid, int *stat_loc, int options);
```

*returns:*

*ID of found process*      on success  
*error no.*                    on failure

*errno:*

```
EINTR  
ECHILD  
EINVAL
```

```
void _exit( int status);
```

## Signals and Timers

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

Default actions:

- 1) Abnormal termination
- 2) Ignore
- 3) Stop process
- 4) Continue process

<signal.h> defines these signal actions:

<b>SIGABRT (1)</b>	Abnormal termination
<b>SIGALRM (1)</b>	Timeout signal
<b>SIGFPE (1)</b>	Erroneous arithmetic operation
<b>SIGHUP (1)</b>	Hangup detected or death of controlling process
<b>SIGILL (1)</b>	Invalid hardware instruction
<b>SIGINT (1)</b>	Interactive attention
<b>SIGKILL (1)</b>	Termination signal
<b>SIGPIPE (1)</b>	Write on a pipe with no readers
<b>SIGQUIT (1)</b>	Interactive termination signal
<b>SIGSEGV (1)</b>	Detection of an invalid memory reference
<b>SIGTERM (1)</b>	Termination signal
<b>SIGUSR1 (1)</b>	Application-defined signal
<b>SIGUSR2 (1)</b>	Application-defined signal
<b>SIGCHLD (2)</b>	Child process terminated or stopped
<b>SIGCONT (4)</b>	Continue if stopped
<b>SIGSTOP (3)</b>	Stop signal

<b>SIGTSTP (3)</b>	Interactive stop signal
<b>SIGTTIN (3)</b>	Read from control terminal by member of background process group
<b>SIGTTOU (3)</b>	Write from control terminal by member of background process group

```
#include <signal.h>
int kill (pid_t pid, int sig);
```

*errno:*

**EINVAL**  
**EPERM**  
**ESRCH**

```
int sigemptyset (sigset_t *set);
int sigfillset (sigset_t *set);
int sigaddset (sigset_t *set);
int sigdelset (sigset_t *set, int signo);
int sigismember (const sigset_t *set, int signo);
```

*returns:*

0	on success
<i>error no.</i>	on failure
1	returned by <b>sigismember</b> if signal is in set

*errno:*

**EINVAL**

```
int signalaction (int sig, const struct sigaction *act, struct
sigaction *oact);
```

*errno:*

**EINVAL**  
**ENOTSUP**

```
int pthread_sigmask (int how, const sigset_t *set, sigset_t *oset);
int sigprocmask (int how, const sigset_t *set, sigset_t *oset);
int sigpending (sigset_t *set);
```

*errno:*

**EINVAL**

```
int sigθsuspend (const sigset_t *sigmask);
```



*errno:*

**EINTR**

```
int sigwait (const sigset_t *set, int *sig);
int sigwaitinfo (const sigset_t *set, siginfo_t *info);
int sigtimedwait (const sigset_t *set, siginfo_t *info,
                  const struct timespec *timeout);
int sigqueue (pid_t pid, int signo, const union sigval value);
```

*returns:*

**sigwait:**

0

**sigwaitinfo, sigtimedwait:**

signal number

**sigqueue:**

0

on success

*error no.*

on failure

*errno:*

**EINVAL**

**ENOSYS**

**EINTR**

**EAGAIN**

**EPERM**

**ESRCH**

```
int pthread_kill (pthread_t *thread, int sig);
```

*errno:*

**ESRCH**

**EINVAL**

```
unsigned int alarm (unsigned int seconds);
```

```
unsigned int sleep (unsigned int seconds);
```

*returns:*

Always successful.

0

if time has expired

*amount of time left*

if interrupted.

```
int pause (void);
```

*errno:*

**EINTR**

## Process Identification

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

```
pid_t getpid (void);  
pid_t getppid (void);  
uid_t getuid (void);  
uid_t geteuid (void);  
gid_t getgid (void);  
gid_t getegid (void);  
pid_t getgrp (void);
```

*returns:*

Always successful: returns respective ID of calling process.

```
int setuid (uid_t uid);  
int setgid (gid_t gid);  
int setsid (void);
```

*errno:*

**EINVAL**  
**EPERM**

```
int getgroups(int gidsetsize, gid_t grouplist[]);
```

*returns:*

<i>No. of group ids</i>	on success
<i>error no.</i>	on failure

*errno:*

**EINVAL**

```
char *getlogin (void);  
int getlogin_r (char *name, size_t namesize);
```

*returns:*

**getlogin:**

pointer to the string containing user's login name  
**NULL** if not found.

**getlogin\_r:**

0	on success
<i>error no.</i>	on failure

*errno:*

**ERANGE**

**int uname (struct utsname \*name);**

**#include <stdlib.h>**

**char \*getenv (const char \*name);**

*returns:*

Pointer to the string list of the environment if successful, an error number if not.

**char \*ctermid (char \*s);**

*returns:*

Pointer to string that represents the pathname if successful, empty string if not.

**char \*ttyname (int fildes);**

**int ttyname\_r (int fildes, char \*name, size\_t namesize);**

**int isatty (int fildes);**

*returns:*

**ttyname:**

Returns a pointer to a string containing the pathname of the terminal if successful and **NULL** if not.

**ttyname\_r:**

Stores pathname in **\*name** if successful and returns an error number if not.

**isatty:**

Returns 1 if successful and 0 if not.

*errno:*

**EBADF**

**ENOTTY**

**ERANGE**

## Files and Directories

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

### Directories:

```
DIR *opendir (const char *dirname);  
struct dirent *readdir (DIR *dirp);  
int readdir_r (DIR *dirp, struct dirent *entry,  
               struct dirent **result);  
void rewinddir (DIR *dirp);  
int closedir (DIR *dirp);
```

*returns:*

**opendir** and **readdir** return a pointer if successful, or **NULL** if not.  
**readdir\_r** and **closedir** return 0 if successful, or an error number if not.

*errno:*

```
EACCESS  
ENAMETOOLONG  
ENOENT  
ENOTDIR  
EMFILE  
ENFILE  
EBADF
```

```
int chdir (const char *path);
```

*errno:*

```
EACCES  
ENAMETOOLONG  
ENOTDIR  
ENOENT
```

```
char *getcwd (char *buf, size_t size);
```

*returns:*

<b>buf</b>	on success
<b>NULL</b> pointer	on failure

*errno:*

```
EINVAL
```

## **ERANGE EACCES**

### **File, Directory Creation, Deletion and Manipulation:**

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

```
int open (const char *path, int oflag, ...);
```

*returns:*

Lowest nonnegative integer representing an unused file descriptor is successful, an error number if not.

*errno:*

```
EACCES  
EEXIST  
EINTR  
EINVAL  
EISDIR  
EMFILE  
ENAMETOOLONG  
ENFILE  
ENOENT  
ENOSPC  
ENOTDIR  
ENXIO  
EROFS
```

```
mode_t unmask (mode_t cmask);
```

*returns:*

Always successful

```
int link (const char *existing, const char *new);  
int mkdir (const char * path, mode_t mode);  
int mkfifo (const char *path, mode_t mode);  
int unlink (const char *path);  
int rmdir (const char *path);  
int rename (const char *old, const char *new);  
int stat (const char *path, struct stat *buf);  
int fstat (int fildes, struct stat *buf);  
int access (const char *path, int amode);
```

```
int chmod (const char *path, mode_t mode);
int fchmod (int fildes, mode_t mode);
int chown (const char *path, uid_t owner, gid_t group);
```

*errno:*

```
EACCES
EBUSY
EEXIST
EINVAL
EMLINK
ENAMETOOLONG
ENOENT
ENOSPC
ENOTDIR
EPERM
EROFS
EXDEV
```

```
#include <sys/types.h>
#include <utime.h>
```

```
int utime(const char *path, const struct utimbuf *times);
```

*errno:*

```
EACCES
ENAMETOOLONG
ENOENT
ENOTDIR
EPERM
EROFS
```

```
#include <unistd.h>
```

```
int ftruncate (int fildes, off_t length);
```

*errno:*

```
EBADF
EINVAL
EROFS
```

```
#include <unistd.h>
```

```
long pathconf (const char *path, int name);
long fpathconf (int fildes, int name);
```

*returns:*

Current variable value for the file or directory if succesful, an error number if not.

*errno:*

**EACCES**

**EBADF**

**EINVAL**

**ENAMETOOLONG**

**ENOENT**

**ENOTDIR**

## Input and Output Primitives

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

```
int pipe (int fildes[2]);  
int dup (int fildes);  
int dup2 (int fildes, int fildes[2]);  
int close (int fildes);
```

*errno:*

```
EBADF  
EMFILE  
ENFILE  
EINTR
```

**Reading to, writing from, and controlling the state of a file:**

```
ssize_t read (int fildes, void *buf, size_t nbyte);  
ssize_t write (int fildes, const void *buf, size_t nbyte);
```

*returns:*

Number of bytes written or read if successful, an error number if not.

*errno:*

```
EAGAIN  
EBADF  
EFBIG  
EINTR  
EIO  
ENOSPC  
EPIPE
```

```
int fcntl (int fildes, int cmd, ...);
```

*returns:*

A value determined by the **cmd** parameter if successful, an error number if not.

*errno:*

```
EACCES  
EAGAIN  
EBADF  
EINTR
```



**EINVAL**  
**EMFILE**  
**ENOLCK**  
**EDEADLK**

**off\_t lseek (int fildes, off\_t offset, int whence);**

*returns:*

<i>offset location</i>	on success
<i>off_t - 1</i>	on failure

*errno:*

**EBADF**  
**EINVAL**  
**ESPIPE**

**#include <unistd.h>**

**int fsync (int fildes);**  
**int fdatasync (int fildes);**

*errno:*

**EAGAIN**  
**EBADF**  
**EINVAL**  
**ENOSYS**

**Asynchronous I/O:**

**#include <aio.h>**

**int aio\_read (struct aiocb \*aiocbp);**  
**int aio\_write (struct aiocb \*aiocbp);**  
**int lio\_listio (int mode, struct aiocb \*const list[],**  
**int nent, struct sigevent \*sig);**  
**int aio\_error (const struct \*aiocbp);**  
**int aio\_return (struct aiocb \*aiocbp);**  
**int aio\_suspend (const struct aiocb \*const list[],**  
**int nent, const struct timespec \*timeout);**  
**int aio\_fsync (int op, struct aiocb \*aiocbp);**

*returns:*

0 if successful, status value or undefined if not complete, an error number if error occurs.

*errno:*

**EAGAIN**

**EINVAL**

**EINTR**

**EIO**

**EBADF**

**EINVAL**

**ENOSYS**

## Terminal Control

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

**#include <termios.h>**

### Baud rate functions:

```
speed_t cfgetospeed (const struct termios *termios_p);  
int cfsetospeed (struct termios *termios_p, speed_t speed);  
speed_t cfgetispeed (const struct termios *termios_p);  
int cfsetispeed (struct termios *termios_p, speed_t speed);
```

*returns:*

If a **set** function, the speed value is returned; if a **get** function, 0; an error number if not successful.

### Terminal interface functions:

```
int tcgetattr (int fildes, struct termios *termios_p);  
int tcsetattr (int fildes, int optional_actions,  
               const struct termios *termios_p);  
int tcsendbreak (int fildes, int duration);  
int tcdrain (int fildes);  
int tcflush (int fildes);  
int tcflow (int fildes, int action);
```

*errno:*

```
EBADF  
EINTR  
EINVAL  
ENOTTY
```

```
pid_t tcgetpgrp (int fildes);
```

*returns:*

A value greater than 1 that does not match existing process group ID if successful, an error number if not.

*errno:*

```
EBADF  
ENOSYS  
ENOTTY
```

```
int tcsetpgrp (int fildes, pid_t pgrp_id);
```

*errno:*

**EBADF**

**EINVAL**

**ENOSYS**

**ENOTTY**

**EPERM**

## Language-Specific Services

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

```
#include <stdio.h>
```

```
int fileno (FILE *stream);  
FILE *fdopen (int fildes, const char *type);  
void flockfile (FILE *file);  
int ftrylockfile (FILE *file);  
void funlockfile (FILE *file);  
int getc_unlocked (FILE *stream);  
int getchar_unlocked (void);  
int putc_unlocked (int c, FILE *stream);  
int putchar_unlocked (int c);
```

```
#include <setjmp.h>
```

```
int sigsetjmp (sigjmp_buf env, int savemask);  
void siglongjmp (sigjmp_buf env, int val);
```

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

```
char *strtok_r (char *s, const char *sep, char **lasts);  
char *asctime_r (const struct tm *tm, char *buf);  
char *ctime_r (const time_t *clock, char *buf);  
struct tm *gmtime_r (const time_t *clock, struct tm *result);  
struct tm *localtime_r (const time_t *clock, struct tm *result);
```

```
#include <stdlib.h>
```

```
int rand_r (unsigned int *seed);
```

*returns:*

0	for int returns on success
<i>pointer to return type</i>	for other returns on success
<i>error no. or NULL</i>	on failure

There are no **errno** values specified for these functions.

# Synchronization

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

## Semaphores:

```
#include <semaphore.h>
```

```
int sem_init (sem_t *sem, int pshared, unsigned int value);  
int sem_destroy (sem_t *sem);  
int sem_close (sem_t *sem);  
int sem_unlink (const char *name);  
int sem_wait (sem_t *sem);  
int sem_trywait (sem_t *sem);  
int sem_post (sem_t *sem);
```

*errno:*

```
EAGAIN  
EDEADLK  
EINTR  
EINVAL  
ENAMETOOLONG  
ENOENT  
ENOSPC  
ENOSYS  
EPERM
```

```
sem_t *sem_open (const char *name, int oflag, ...);
```

*returns:*

The address of the semaphore if successful, an error number if not.

*errno:*

```
EACCES  
EEXIST  
EINTR  
EINVAL  
EMFILE  
ENAMETOOLONG  
ENFILE  
ENOENT  
ENOSPC  
ENOSYS
```

```
int sem_getvalue (sem_t *sem, int *sval);
```

*returns:*

0 and sets **sval** to value of the semaphore referenced if successful, an error number if not.

### **Mutexes:**

```
#include <pthread.h>
```

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);  
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);  
int pthread_mutexattr_getpshared  
    (const pthread_mutexattr_t *attr, int *pshared);  
int pthread_mutexattr_setpshared  
    (pthread_mutexattr_t *attr, int pshared);  
int pthread_mutex_init (pthread_mutex_t *mutex,  
    const pthread_mutexattr_t *attr);  
int pthread_mutex_destroy (pthread_mutex_t *mutex);  
int pthread_mutex_lock (pthread_mutex_t *mutex);  
int pthread_mutex_trylock (pthread_mutex_t *mutex);  
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

*returns:*

0 if successful (**pthread\_mutexattr\_getpshared** stores the shared attribute in **pshared**), an error number if not.

*errno:*

```
EAGAIN  
EBUSY  
EDEADLK  
EINVAL  
ENOMEM  
ENOSYS  
EPERM
```

### **Condition variables:**

```
#include <pthread.h>
```

```
int pthread_condattr_init (pthread_condattr_t *attr);  
int pthread_condattr_destroy (pthread_condattr_t *attr);  
int pthread_condattr_getpshared  
    (const pthread_condattr_t *attr, int *pshared);
```

```
int pthread_condattr_setpshared
    (const pthread_condattr_t *attr, int pshared);
int pthread_cond_init (pthread_cond_t *cond,
    const pthread_condattr_t *attr);
int pthread_cond_destroy (pthread_cond_t *cond);
int pthread_cond_signal (pthread_cond_t *cond);
int pthread_cond_broadcast (pthread_cond_t *cond);
```

*returns:*

0 if successful (**pthread\_condattr\_getpshared** stores the shared attribute in **pshared**), an error number if not.

*errno:*

```
EAGAIN
EBUSY
EDEADLK
EINVAL
ENOMEM
ENOSYS
EPERM
```

```
int pthread_cond_wait (pthread_cond_t *cond,
    pthread_mutex_t *mutex);
int pthread_cond_timedwait (pthread_cond_t *cond,
    pthread_mutex_t *mutex,
    const struct timespec *abstime);
```

*errno:*

```
EINVAL
ETIMEDOUT
```



## Memory Management

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

### Memory locking:

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

```
int mlockall (int flags);
int munlockall (void);
int mlock (const void *addr, size_t len);
int munlock (const void *addr, size_t len);
```

*errno:*

```
EAGAIN
EINVAL
ENOMEM
ENOSYS
EPERM
```

### Memory mapping and protection:

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

```
void *mmap (void *addr, size_t len, int prot, int flags,
            int fildes, off_t off);
int munmap (void *addr, size_t len);
int mprotect (const void *addr, size_t len, int prot);
```

*returns:*

**munmap** and **mprotect** return 0, **mmap** returns the address if successful, an error number if not.

*errno:*

```
EACCESS
EAGAIN
EBADF
EINVAL
ENODEV
ENOMEM
ENOSYS
ENOTSUP
ENXIO
```

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

```
int msync (void *addr, size_t len, int flags);
```

*errno:*

**EINVAL**

**EFAULT**

## Process Scheduling

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

### Policy and scheduling parameters:

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

```
int sched_setparam (pid_t pid,  
                   const struct sched_param *param);  
int sched_setscheduler (pid_t pid, int policy,  
                       const struct sched_param *param);  
int sched_yield (void);
```

*errno:*

**EINVAL**  
**ENOSYS**  
**EPERM**  
**ESRCH**

```
int sched_getparam (pid_t pid, struct sched_param *param);  
int sched_getscheduler (pid_t pid);  
int sched_get_priority_max (int policy);  
int sched_get_priority_min (int policy);  
int sched_rr_get_interval (pid_t pid, struct timespec *interval);
```

*returns:*

<i>requested value</i>	on success
<i>error no.</i>	on failure

*errno:*

**ENOSYS**  
**EPERM**  
**ESRCH**

### Thread scheduling:

```
#include <pthread.h>
```

```
int pthread_attr_setscope (pthread_attr_t *attr,  
                          int contentionscope);  
int pthread_attr_getscope (const pthread_attr_t *attr,  
                          int contentionscope);  
int pthread_attr_setinheritsched (pthread_attr_t *attr,  
                                  int inheritsched);
```

```

int pthread_attr_getinheritsched
    (const pthread_attr_t *attr, int inheritsched);
int pthread_attr_setschedpolicy (pthread_attr_t *attr,
    int policy);
int pthread_attr_getschedpolicy
    (const pthread_attr_t *attr, int policy);
int pthread_attr_setschedparam (pthread_attr_t *attr,
    const struct sched_param *param);
int pthread_attr_getschedparam
    (const pthread_attr_t *attr,
    struct sched_param *param);
int pthread_setschedparam (pthread_t thread, int policy,
    const struct sched_param *param);
int pthread_getschedparam (pthread_t thread, int *policy,
    struct sched_param *param);

```

*errno:*

```

ENOSYS
EINVAL
ENOTSUP
EPERM
ESRCH

```

#### Scheduling mutex initialization:

```

#include <pthread.h>

int pthread_mutexattr_setprotocol
    (pthread_mutexattr_t *attr, int protocol);
int pthread_mutexattr_getprotocol
    (const pthread_mutexattr_t *attr, int *protocol);
int pthread_mutexattr_setprioceiling
    (pthread_mutexattr_t *attr, int prioceiling);
int pthread_mutexattr_getprioceiling
    (const pthread_mutexattr_t *attr, int *prioceiling);
int pthread_mutex_setprioceiling (pthread_mutex_t *mutex,
    int prioceiling, int *old_ceiling);
int pthread_mutex_getprioceiling
    (const pthread_mutex_t *mutex, int *prioceiling);

```

*errno:*

```

ENOSYS
ENOTSUP
EINVAL
EPERM
ESRCH

```

## Timers

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

### Structures:

```
struct timespec {
    time_t (tv_sec) //seconds
    long (tv_nsec) //nanoseconds
}

struct timespec it_interval; //timer period
struct timespec it_value; //timer expiration
```

### Clock functions:

```
#include <time.h>

int clock_settime (clockid_t clock_id,
                  const struct timespec *tp)
int clock_gettime (clockid_t clock_id,
                  struct timespec *tp)
int clock_getres (clockid_t, clock_id,
                  struct timespec *res)
```

*errno:*

**EINVAL**  
**ENOSYS**  
**EPERM**

### Timer functions:

```
int timer_create (clockid_t clock_id,
                 struct sigevent *evp, timer_t *timerid)
int timer_delete (timer_t timerid)
```

*errno:*

**EAGAIN**  
**EINVAL**  
**ENOSYS**

```
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *value,
```

```
        struct itimerspec *ovalue)
int timer_gettime(timer_t timerid,
        struct itimerspec *value)
int timer_getoverrun (timer_t timerid)
```

*errno:*

**EINVAL**  
**ENOSYS**

**Sleep function:**

```
int nanosleep (const struct timespec *rqtp,  
        struct timespec *rmtp)
```

*errno:*

**EINTR**  
**EINVAL**  
**ENOSYS**

## Message Passing

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

```
mqd_t mq_open (const char *name, int oflag, ...);
```

*returns:*

A message queue descriptor if successful, (**mqd\_t - 1**) if not.

*errno:*

```
EACCES  
EEXIST  
EINTR  
EINVAL  
EMFILE  
ENAMETOOLONG  
ENFILE  
ENOENT  
ENOSPC  
ENOSYS
```

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

```
int mq_close (mqd_t mqdes);  
int mq_unlink (const char *name);  
int mq_send (mqd_t mqdes, const char *msg_ptr,  
             size_t msg_len, unsigned int msg_prio);  
int mq_notify (mqd_t mqdes,  
              const struct sigevent *notification);  
int mq_setattr (mqd_t mqdes, const struct mq_attr *mqstat,  
               struct mq_attr *omqstat);  
int mq_getattr (mqd_t mqdes, struct mq_attr *mqstat);
```

*errno:*

```
EAGAIN  
EBADF  
EBUSY  
EINTR  
EMSGSIZE  
ENOSYS
```

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

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
```

**size\_t msg\_len, unsigned int \*msg\_prio);**

*returns:*

The length of the selected message in bytes, an error number if not.

*errno:*

**EAGAIN**

**EBADF**

**EMSGSIZE**

**EINTR**

**ENOSYS**



# Thread Management

Unless otherwise indicated, all functions return 0 on success and an error number on failure.

## Thread creation attributes:

```
#include <pthread.h>
```

```
int pthread_attr_init (pthread_attr_t *attr);
int pthread_attr_destroy (pthread_attr_t *attr);
int pthread_attr_setstacksize (pthread_attr_t *attr,
                               size_t stacksize);
int pthread_attr_getstacksize (const pthread_attr_t *attr,
                               size_t *stacksize);
int pthread_attr_setstackaddr (pthread_attr_t *attr,
                               void *stackaddr);
int pthread_attr_getstackaddr (const pthread_attr_t *attr,
                               void **stackaddr);
int pthread_attr_setdetachstate (pthread_attr_t *attr,
                                 int detachstate);
int pthread_attr_getdetachstate
    (const pthread_attr_t *attr, int *detachstate);
```

*errno:*

**EINVAL**  
**ENOMEM**  
**ENOSYS**

## Thread operations:

```
#include <pthread.h>
```

```
int pthread_create (pthread_t *thread,
                  const pthread_attr_t *attr,
                  void *(start_routine)(void *), void *arg);
int pthread_join (pthread_t thread, void **value_ptr);
int pthread_detach (pthread_t thread);
```

*errno:*

**EAGAIN**  
**EDEADLK**  
**EINVAL**  
**ESRCH**

**void pthread\_exit (void \*value\_ptr);**

*returns:*

This function does not return.

**pthread\_t pthread\_self (void);**

*returns:*

The ID of the calling thread.

**int pthread\_equal (pthread\_t t1, pthread\_t t2);**

*returns:*

A nonzero value if t1 equals t2, 0 if not.

**void \*pthread\_getspecific (pthread\_key\_t key);**

*returns:*

The thread-specific data value associated with the given key when successful.

### **Thread cancellation:**

**#include <pthread.h>**

**int pthread\_cancel (pthread\_t thread);**

**int pthread\_setcancelstate(int state, int \*oldstate);**

**int pthread\_setcanceltype(int type, int \*oldtype);**

*errno:*

**ESRCH**

**EINVAL**

**void pthread\_testcancel (void );**

**void pthread\_cleanup\_push (void (\*routine)(void \*),  
void \*arg);**

**void pthread\_cleanup\_pop (int execute);**

No return values.

### **Thread-specific data:**

**#include <pthread.h>**

```
int pthread_key_create (pthread_key_t *key,  
                        void (*destructor)(void *));  
int pthread_setspecific (pthread_key_t key,  
                        const void *value);  
int pthread_key_delete (pthread_key_t key);
```

*errno:*

**EAGAIN**  
**EINVAL**  
**ENOMEM**

## **Reference**

## Key Linux Commands

Below are some of the most useful Linux commands:

<b>cat <i>filename</i></b>	The <b>cat</b> command scrolls the contents of the file <b><i>filename</i></b> across the screen.
<b>cd <i>directoryname</i></b>	The <b>cd</b> command changes the directory you are in. There are a variety of parameters that you can put into <b><i>directoryname</i></b> .
<b>cd ..</b>	Moves you up one directory.
<b>cd ~</b>	Moves you to your home directory. You can also move to your home directory by putting nothing in the directory name parameter.
<b>cd <i>name</i></b>	Move you to the <b><i>name</i></b> directory. For more details on these commands, such as options and parameters, please read the <b>man</b> pages supplied in the Linux distribution.
<b>cp <i>oldfile newfile</i></b>	The <b>cp</b> command lets you copy <b><i>oldfile</i></b> to <b><i>newfile</i></b> .
<b>dir <i>directoryname</i></b>	The <b>dir</b> command displays the contents of the directory <b><i>directoryname</i></b> . If you leave <b><i>directoryname</i></b> blank, it will display the contents of the current directory.
<b>echo <i>string</i></b>	The <b>echo</b> command prints the string <b><i>string</i></b> to the display or can be redirected to a file, device, program, or your shell.
<b>find <i>directory- tostartsearch filename actionforlist</i></b>	The <b>find</b> command searches the directory <b><i>directorytostartsearch</i></b> , and all subdirectories, or the file <b><i>filename</i></b> , and <b><i>actionforlist</i></b> is what the command does with the list.
<b>grep <i>text file</i></b>	The <b>grep</b> command searches the file <b><i>file</i></b> for the text pattern <b><i>text</i></b> and prints to the screen all of the portions of the file in which <b><i>text</i></b> was found.

<b>insmod <i>module</i></b>	The <b>insmod</b> command inserts the module <b><i>module</i></b> into the kernel. You must be logged in as root or have super-user privileges to use this command.
<b>less <i>filename</i></b>	<b>less</b> is a program that displays the contents of the file <b><i>filename</i></b> to the screen like the <b>more</b> program. <b>less</b> allows you to move backwards in the file as opposed to <b>more</b> , which only allows you to move forward through the file.
<b>ls <i>directoryname</i></b>	The <b>ls</b> command lists the contents of the directory <b><i>directoryname</i></b> . You can change the format of the printed list via options that can be found in the <b>man</b> page for <b>ls</b> . If you leave <b><i>directoryname</i></b> blank, it will list the contents of the current directory.
<b>lsmod</b>	The <b>lsmod</b> command lists all of the modules that have been inserted into the system.
<b>make</b>	<b>make</b> is a utility that finds out which parts of a large program need to be recompiled and issues the commands needed to do the recompilation.
<b>man <i>subject</i></b>	The <b>man</b> command formats the online manual pages for the subject <b><i>subject</i></b> and displays that information to the screen. It is very useful because it gives very detailed information about commands and other things. It is advised that you read the <b>man</b> pages on any of the commands you look up in this appendix.
<b>mkdir <i>directoryname</i></b>	The command <b>mkdir</b> creates the directory <b><i>directoryname</i></b> in the current directory you are in, unless you give a full path name for <b><i>directoryname</i></b> , which will then create it there.
<b>more <i>filename</i></b>	<b>more</b> is a program that displays the contents of the file <b><i>filename</i></b> to the screen like the <b>less</b> program. <b>more</b> only allows you to move forward in the file as opposed to <b>less</b> , which allows you to move in both directions through the file.

<b>mount <i>directoryname</i></b>	The <b>mount</b> command attaches the filesystem to the directory <b><i>directoryname</i></b> . If <b><i>directory-name</i></b> is left blank, the command will list all of the currently mounted filesystems.
<b>mv <i>object1</i> [<i>object2</i> or <i>destinationlocation</i>]</b>	The <b>mv</b> command moves <b><i>object1</i></b> into <b><i>object2</i></b> or into <b><i>destinationlocation</i></b> . In other words, you can move a file into another file, or you can move a file into a directory.
<b>ps</b>	The <b>ps</b> command displays a snapshot of all the current processes.
<b>pwd</b>	The <b>pwd</b> command displays the path of the current directory you are in.
<b>rm <i>filename</i></b>	The <b>rm</b> command removes the file <b><i>filename</i></b> from the system. Be very careful with this command because there is no way of retrieving the file once it has been removed.
<b>rmdir <i>directoryname</i></b>	The <b>rmdir</b> command allows you to remove the empty directory <b><i>directoryname</i></b> . Remember, <b><i>directoryname</i></b> must be empty.
<b>rmmod <i>module</i></b>	The <b>rmmod</b> command removes the module <b><i>module</i></b> from the kernel. You must be logged in as root, or have super-user privileges, to use this command.
<b>su</b>	The <b>su</b> command allows you to have super-user privileges. It will ask you for a password. When it does, you must put in root's password. It is now as if you have logged in as root.
<b>umount <i>directoryname</i></b>	The <b>umount</b> command detaches the filesystem from the directory <b><i>directoryname</i></b> . Prints the clock frequency at which the system processor is running. The units are in MHz.

## Key TimeSys Linux Commands

These are some of the key commands for the Resource Kernel component of TimeSys Linux.

- clockfreq** Prints the clock frequency at which the system processor is running. The units are in MHz.
- rkattach** <resource set in hex>  
<process id>  
[process id] ... Attaches to the specified resource set the list of specified processes.
- RKcleanRS** Destroys all resource sets and their associated reserves in the system.
- rkdetach** <resource set in hex>  
[resource set] ... Destroy the specified resource set(s).
- rkdetach** <resource in hex>  
<processid>  
[process id]... Detach from the specified resource set the set list of specified processes.
- rkexec** --newrset  
--cpu [time in us]  
--period[period in us] -- deadline [deadline in us]  
--hard (or --[soft])  
--exec '<args>' Execute the specified process creating a new resource set (or specify an existing resource set to use, using a --rset option). The CPU reservation parameters (CPU time, period and deadline) can be specified.
- rklist** List the parameters of the current resource sets and their reservations in the system.



## Glossary of Terms and Concepts

The following definitions apply to terms used throughout this manual, and are derived from the “Handbook of Real-Time Linux.” A clear understanding of these terms is very useful for anyone working with real-time Linux.

<b>Action</b>	The smallest decomposition of a response; a segment of a response that cannot change system resource allocation. In TimeWiz, an action must be bound to a (physical) RESOURCE before it is analyzed. An action can also use zero, one, or more logical resources.
<b>Aperiodic event</b>	An event sequence whose arrival pattern is not periodic.
<b>Average-case response time</b>	The average response time of a response's jobs within a given interval. In TimeWiz, this is obtained through simulation. It is possible that there is a wide discrepancy between the average- and worst-case response times for a particular task. In many real-time systems (particularly for hard real-time tasks), the worst-case response time <i>must</i> be within a well-specified interval.
<b>Blocking</b>	The act of a lower-priority task delaying the execution of a higher-priority task; more commonly known as priority inversion. Such priority inversion takes more complex forms in distributed and shared memory implementations.
<b>Blocking time</b>	The delay effect (also called the “duration of priority inversion”) caused to events with higher-priority responses by events with lower-priority responses.
<b>Bursty arrivals</b>	An arrival pattern in which events may occur arbitrarily close to a previous event, but over an extended period of time the number of events is restricted by a specific event density; that is, there is a bound on the number of events per time interval. Bursty arrivals are

modeled in TimeWiz using their minimum interarrival time and their resource consumption in that interval.

**Critical section**

Period during which a real-time task is holding onto a shared resource.

**Data-sharing policy**

A policy specific to a (physical) resource that determines how logical resources bound to the (physical) resource can be accessed. Some schemes do not provide any protection against priority inversion, while others provide varying degrees of protection. TimeWiz supports multiple data-sharing policies including FIFO (no protection against priority inversion), PRIORITY INHERITANCE PROTOCOL, PRIORITY CEILING PROTOCOL, HIGHEST LOCKER PRIORITY PROTOCOL, and KERNELIZED MONITOR (non-preemptive execution) policies.

**Deadline-monotonic scheduling algorithm**

A fixed-priority algorithm in which the highest priority is assigned to the task with the earliest *relative* delay constraint (deadline) from each instance of its arrival. The priorities of the remaining tasks are assigned monotonically (or consistently) in order of their deadlines.

This algorithm and the earliest-deadline scheduling algorithm are not the same. In this algorithm, all instances of the same task have the same priority. In the earliest-deadline scheduling algorithm, each instance of the same task has a *different* priority, equal to the absolute deadline (time) by which it must be completed. The rate-monotonic scheduling algorithm and the deadline-monotonic algorithm are one and the same when the relative deadline requirement and periods are equal (which happens often).

**Deterministic system**

A system in which it is possible to determine exactly what is or will be executing on the processor during system execution. Deterministic systems result from the use of

certain scheduling policies for groups of processes.

**Dynamic-priority scheduling policy**

An allocation policy that uses priorities to decide how to assign a resource. Priorities change from instance to instance of the same task (and can also vary during the lifetime of the same instance of a task). The earliest-deadline scheduling algorithm is an example of a dynamic-priority scheduling policy.

**Earliest-deadline scheduling**

A dynamic-priority assignment policy in which the highest priority is assigned to the task with the most imminent deadline.

**Event**

A change in state arising from a stimulus within the system or external to the system; or one spurred by the passage of time. An event is typically caused by an interrupt on an input port or a timer expiry. See also TRACE and TRIGGER.

**Execution time**

Amount of time that a response will consume on a CPU.

**Fixed-priority scheduling policy**

An allocation policy that uses priorities to decide how to assign a resource. The priority (normally) remains fixed from instance to instance of the same task. Rate-monotonic and deadline-monotonic scheduling policies are fixed-priority scheduling policies.

**Hardware-priority scheduling policy**

An allocation policy in which the priority of a request for the backplane is determined by a hardware register on each card that plugs into the backplane. Presumably, the hardware priority value reflects the importance of the device that is connected to the adapter.

**Highest-locker priority**

A DATA-SHARING POLICY in which an action using a logical resource is executed at the highest priority of all actions that use the logical resource (i.e. at the PRIORITY CEILING of the resource). This protocol provides a good level of control over priority inversion.

<b>Input jitter</b>	The deviation in the size of the interval between the arrival times of a periodic action.
<b>Kernelized monitor</b>	A DATA-SHARING POLICY in which an action using a logical resource is executed in non-preemptive fashion (i.e., at kernel priority). This protocol provides a good level of control over priority inversion except when one or more actions using a logical resource has a long execution time (relative to the timing constraints of other higher-priority tasks).
<b>Logical resource</b>	A system entity that is normally shared across multiple tasks. A logical resource must be bound to a physical resource like a processor, and is modeled in TimeWiz as an action with a mutual exclusion requirement. Also, see DATA-SHARING POLICY.
<b>Output jitter</b>	The deviation in the size of the interval between the completion times of a periodic action.
<b>Period</b>	The interarrival interval for a periodic event sequence. Also, see INPUT JITTER.
<b>Periodic event</b>	An event sequence with constant interarrival intervals. Described in terms of the period (the interarrival interval) and a phase value.
<b>Preemption</b>	The act of a higher-priority process taking control of the processor from a lower-priority task.
<b>Priority ceiling</b>	This is associated with each logical resource and corresponds to the priority of the highest-priority action that uses the logical resource.
<b>Priority ceiling protocol</b>	A data-sharing policy in which an action using a logical resource can start only if its priority is higher than the PRIORITY CEILINGS of all logical resources locked by other responses. This protocol provides a good level of control over priority inversion.

<b>Priority inheritance protocol</b>	A DATA-SHARING POLICY in which an action using a logical resource executes at the highest of its own priority or the highest priority of any action waiting to use this resource. This protocol provides an acceptable level of control over priority inversion.
<b>Priority inversion</b>	This is said to occur when a higher-priority action is forced to wait for the execution of a lower-priority action. This is typically caused by the use of logical resources, which must be accessed mutually exclusively by different actions. Uncontrolled priority inversion can lead to timing constraints being violated at relatively low levels of RESOURCE UTILIZATION. Also see BLOCKING and BLOCKING TIME.
<b>Rate-monotonic scheduling algorithm</b>	Algorithm in which highest priority is assigned to the task with the highest rate (in other words, with the shortest period) and the priorities of the remaining tasks are assigned monotonically (or consistently) in order of their rates.
<b>Rate-monotonic scheduling</b>	A special case of fixed-priority scheduling that uses the rate of a periodic task as the basis for assigning priorities to periodic tasks. Tasks with higher rates are assigned higher priorities.
<b>Real-time system</b>	<p>A system that controls an environment by receiving data, processing it, and taking action or returning results quickly enough to affect the functioning of the environment at that time.</p> <p>A system in which the definition of system correctness includes at least one requirement to respond to an event with a time limitation.</p>
<b>Resource</b>	A <i>physical</i> entity such as a processor, a back-plane bus, a network link, or a network router which can be used by one or more actions. A resource may have a resource allocation policy (such as rate-monotonic scheduling) and a data-sharing policy.

<b>Response</b>	A time-ordered sequence of events arising from the same stimulus. In TimeWiz, an event can trigger one or more actions to be executed.
<b>Responses</b>	Multiple time-ordered sequences of events, each arising from a distinct event. Event sequences that result in responses on the same resource often cause resource contention that must be managed through a resource allocation policy.
<b>Task</b>	A schedulable unit of processing composed of one or more actions. Synonymous with <i>process</i> .
<b>Tracer</b>	A stimulus. Synonymous with a single instance of an <code>EVENT</code> within TimeWiz, and is used to represent an end-to-end data flow sequence spanning <i>multiple</i> physical resources. An end-to-end timing constraint is normally associated with a tracer event. TimeWiz computes both worst-case and average-case response times to a tracer using analysis and simulation respectively. Also see <code>TRIGGER</code> .
<b>Trigger</b>	<p>A stimulus with an arrival pattern. Mostly synonymous with the term “<code>EVENT</code>” within TimeWiz but is used to name an event whose response consists of a chain of actions executing on, at most, a <i>single</i> resource.</p> <p>In TimeWiz, a trigger is bound to a (physical) resource when one or more actions in its corresponding response are bound to a (physical) resource. Also see <code>TRACER</code>.</p>
<b>Utilization</b>	The ratio of a response's usage to its period, usually expressed as a percentage. For a CPU resource, this is execution time divided by period.
<b>Worst-case response time</b>	The maximum possible response time of a response's jobs (instances). Also see <code>OUTPUT JITTER</code> .

## Some Key References on Real-Time Linux

ACM, *Operating Systems Review Special Issue on Real-Time Operating Systems*, Vol. 23, No. 3, July 1989.

Baker, T., "Stack-Based Scheduling of Realtime Processes," *Journal of Real-Time Systems*, Vol. 3, No. 1, pp. 67-100, March 1991.

Bollela, G., et al., *The Real-Time Specification for Java*, 2000.

Borger, M. W., and Rajkumar, R., "Implementing Priority Inheritance Algorithms in an Ada Runtime System," *Technical Report*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pa., February 1989.

Burns, A., "Scheduling Hard Real-Time Systems: A Review," *Software Engineering Journal*, pp. 116-128, May 1991.

Chen, M., and Lin, K.J., "Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-Time Systems," *Journal of Real-Time Systems*, Vol. 2, No. 4, pp. 325-346, November 1990.

Gafford, J. D., "Rate Monotonic Scheduling," *IEEE Micro*, June 1990.

Gagliardi, M., Rajkumar, R., and Sha, L., "Designing for Evolvability: Building Blocks for Evolvable Real-Time Systems," *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1996.

Harbour, M. G., Klein, M. H., and Lehoczky, J. P., "Fixed Priority Scheduling of Periodic Tasks with Varying Execution Priority," *Proceedings of IEEE Real-Time Systems Symposium*, December 1991.

IEEE Standard P1003.4 (Real-time extensions to POSIX), IEEE, 345 East 47th St., New York, N.Y., 10017, 1991.

Jeffay, K., "Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems," *IEEE Real-Time Systems Symposium*, pp. 89-99, December 1992.

Joseph, M., and Pandya, "Finding Response Times in a Real-Time System," *The Computer Journal (British Computing Society)*, Vol. 29, No. 5, pp. 390-395, October 1986.

- Juwa, K., and Rajkumar, R., "A Middleware Service for Real-Time Push-Pull Communications," *Proceedings of IEEE Workshop on Dependable Real-Time E-Commerce Systems (DARE'98)*, June 1998.
- Lee, C., Lehoczky, J., Rajkumar, R., and Siewiorek, D., "On Quality of Service Optimization with Discrete QoS Options," *Proceedings of the IEEE Real-time Technology and Applications Symposium*, June 1999.
- Lee, C., Rajkumar, R., and Mercer, C. W., "Experiences with Processor Reservation and Dynamic QOS in Real-Time Mach," *Proceedings of Multimedia Japan*, March 1996.
- Lee, C., Yoshida, K., Mercer, C. W., and Rajkumar, R., "Predictable Communication Protocol Processing in Real-Time Mach," *Proceedings of the Real-Time Technology and Applications Symposium*, June 1996.
- Lehoczky, J. P., Sha, L., and Strosnider, J., "Enhancing Aperiodic Responsiveness in a Hard Real-Time Environment," *IEEE Real-Time System Symposium*, 1987.
- Lehoczky, J. P., Sha, L., and Ding, Y., "The Rate-Monotonic Scheduling Algorithm — Exact Characterization and Average Case Behavior," *Proceedings of IEEE Real-Time System Symposium*, 1989.
- Lehoczky, J. P., "Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines," *IEEE Real-Time Systems Symposium*, December 1990.
- Leung, J., and Whitehead, J., "On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks," *Performance Evaluation (2)*, 1982.
- Liu, C. L., and Layland, J. W., "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *JACM*, Vol. 20 (1), pp. 46-61, 1973.
- Mercer, C. W., and Rajkumar, R., "An Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management," *Proceedings of the Real-Time Technology and Applications Symposium*, pp. 134-139, May 1995.
- Mercer, C. W., Rajkumar, R., and Zelenka, J., "Temporal Protection in Real-Time Operating Systems," *Proceedings of the 11th IEEE Workshop on Real-Time Operating Systems and Software*, pp. 79-83, May 1994.



Mercer, C. W., Rajkumar, R., and Zelenka, J., "On Predictable Operating System Protocol Processing," *Technical Report CMU-CS-94-165*, School of Computer Science, Carnegie Mellon University, May 1994.

Molano, A., Juva, K., and Rajkumar, R., "Real-Time Filesystems: Guaranteeing Timing Constraints for Disk Accesses in RT-Mach," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

Molano, A., Juva, K., and Rajkumar, R., "Dynamic Disk Bandwidth Management and Metadata Pre-fetching in a Reserved Real-Time Filesystem," *10th Euromicro Workshop on Real-Time Systems*, June 1998.

Oikawa, S., and Rajkumar, R., "Linux/RK: A Portable Resource Kernel in Linux," *IEEE Real-Time Systems Symposium Work-In-Progress*, Madrid, December 1998.

Oikawa, S., and Rajkumar, R., "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior," *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Vancouver, June 1999.

Rajkumar, R., "Real-Time Synchronization Protocols for Shared Memory Multiprocessors," *The Tenth International Conference on Distributed Computing Systems*, 1990.

Rajkumar, R., "Synchronization in Real-Time Systems: A Priority Inheritance Approach," Kluwer Academic Publishers, ISBN 0-7923-9211-6, 1991.

Rajkumar, R., and Gagliardi, M., "High Availability in The Real-Time Publisher/Subscriber Inter-Process Communication Model," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

Rajkumar, R., Juva, K., Molano, A., and Oikawa, S., "Resource Kernels: A Resource-Centric Approach to Real-Time Systems," *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

Rajkumar, R., Lee, C., Lehoczky, J., and Siewiorek, D., "A Resource Allocation Model for QoS Management," *Proceedings of the IEEE Real-Time Systems Symposium*, December 1997.

Rajkumar, R., Sha, L., and Lehoczky, J. P., "An Experimental Investigation of Synchronization Protocols," *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*, May 1988.

Rajkumar, R., Sha, L., and Lehoczky, J. P., "Real-Time Synchronization Protocols for Multiprocessors," *Proceedings of the IEEE Real-Time Systems Symposium*, Huntsville, Ala., pp. 259-269, December 1988.

Sha, L., Lehoczky, J. P., and Rajkumar, R., "Solutions for Some Practical Problems in Prioritized Preemptive Scheduling," *IEEE Real-Time Systems Symposium*, 1986.

Sha, L., and Goodenough, J. B., "Real-Time Scheduling Theory and Ada," *IEEE Computer*, April 1990.

Sha, L., Rajkumar, R., and Sathaye, S., "Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems," *Proceedings of the IEEE*, January 1994.

Sha, L., Rajkumar, R., and Lehoczky, J. P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions On Computers*, September 1990.

Sprunt, H. M. B., Sha, L., and Lehoczky, J. P., "Aperiodic Task Scheduling for Hard Real-Time Systems," *The Journal of Real-Time Systems*, No. 1, pp. 27-60, 1989.

Stankovic, J. A., "Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems," *IEEE Computer*, Vol. 21, No. 10, October 1988.

## **Corporate Headquarters**

925 Liberty Avenue  
6th Floor  
Pittsburgh, PA 15222

888.432.TIME  
412.232.3250  
Fax: 412.232.0655

**[www.timesys.com](http://www.timesys.com)**

© 2002 TimeSys Corporation.  
All rights reserved.

*TimeTrace® and TimeWiz® are registered trademarks of TimeSys Corporation.*

*TimeSys Linux™, TimeSys Linux/Real Time™, TimeSys Linux/CPU™, TimeSys Linux/NET™, TimeStorm™, and TimeSys™ are trademarks of TimeSys Corporation.*

*Linux is a trademark of Linus Torvalds.*

*All other trademarks, registered trademarks, and product names are the property of their respective owners.*

0110