RTLinux Version Two

Victor Yodaiken and Michael Barabanov VJY Associates LLC
yodaiken@rtlinux.com

1 Introduction

RTLinux is the *hard realtime* variant of Linux that makes it possible to control robots, data acquisition systems, manufacturing plants, and other time-sensitive instruments and machines. Version 1 of RTLinux was designed to run on low end x86 based computers and provided a spartan API and programming environment. Version 2 RTLinux is a complete rewrite, designed support symmetric multiprocessing, to run on a larger range of systems, and with extensions for ease of use. In this paper, we will discuss the new system and its API with particular attention to the problems of increasing ease of use and adherence to standards, without performance compromise.

RTLinux provides the capability of running special realtime tasks and interrupt handlers on the same machine as standard Linux. These tasks and handlers execute when they need to execute no matter what Linux is doing. The worst case time between the moment a hardware interrupt is detected by the processor and the moment an interrupt handler starts to execute is under 15 microseconds on RTLinux running on a generic x86. A RTLinux periodic task runs within 25 microseconds of its scheduled time on the same hardware. These times are hardware limited, and as hardware improves RTLinux will also improve. Standard Linux has excellent *average* performance and can even provide millisecond level scheduling precision for tasks using the POSIX *soft realtime* capabilities. Standard Linux is not, however, designed to provide submillisecond precision and reliable timing guarantees.

RTLinux Version Two is structured as a small core component and a set of optional components.

- The core component permits installation of very low latency interrupt handlers that cannot be delayed or preempted by Linux itself and some low level synchronization and interrupt control routines. This core component has been extended to support SMP and at the same time it has been simplified by removing some functionality that can be provided outside the core.
- The majority of RTLinux functionality is in a collection of loadable kernel modules that provide optional services and levels of abstraction. These modules include.
    1. `rtl_sched` a priority scheduler that supports both a "lite POSIX" interface described below and the original V1 RTLinux API.
    2. `rtl_time` which controls the processor clocks and exports an abstract interface for connecting handlers to clocks.
    3. `rtl_posixio` supports POSIX style read/write/open interface to device drivers.
    4. `rtl_fifo` connects RT tasks and interrupt handlers to Linux processes through a device layer so that Linux processes can read/write to RT components.

5. `semaphore` is a contributed package by Jerry Epplin which gives RT tasks blocking semaphores. POSIX mutex support is planned to be available in the next minor version update of RTLinux.
6. `mbuff` is a contributed package written by Tomasz Motylewski for providing shared memory between RT components and Linux processes.

The key RTLinux design objective is that the system should be transparent, modular, and extensible. Transparency means that there are no unopenable black boxes and the cost of any operation should be determinable. Modularity means that it is possible to omit functionality and the expense of that functionality if it is not needed. The base RTLinux system supports high speed interrupt handling and no more. And extensibility means that programmers should be able to add modules and tailor the system to their requirements. As the obvious example, the RTLinux simple priority scheduler can be easily replaced by schedulers more suited to the needs of some specific application.

When developing RTLinux we have tried to maximize the advantage we get from having Linux and its powerful capabilities available. In fact **RTLinux development rule number 1** is:

> If a service or operation is inherently non-realtime, it should be provided in Linux and not in the RT components.

The remainder of this note is in XXX sections. In section 2 we explain why POSIX-style API is provided by RTLinux V2 and how we have tailored our implementation to meet the goals of compliance and performance. Section 3 covers the low level core API and the timer module. Section 4 introduces the API of the default scheduler and describes how alternate schedulers can be implemented. Section 5 describes the POSIX I/O structure in RTLinux and two critical drivers.

2 POSIX

2.1 General considerations

The POSIX 1003.1 realtime standard is both impossible and necessary for hard realtime. The standard is impossible because a straightforward implementation of, for example, a POSIX filesystem, imposes open ended computing and resource commitments that a hard realtime system cannot provide. The standard is necessary because it provides the only widely accepted and used API for realtime and because it facilitates connection and software migration between realtime and non-realtime POSIX systems. Fortunately, the POSIX concept of an *application environment profile*[**?**] provides a basis for reconciling POSIX and hard realtime.

Version 2.0 RTLinux moves towards the POSIX RealTime specification by following the POSIX ”single process/minimal realtime system” model with some compatible extensions for SMP[1]. The POSIX Draft standard identifies, in section 6, a ”Minimal Realtime System Profile” (PSE51) intended for hard realtime systems like RTLinux. The ”rationale” given is that ”the POSIX.1c Threads model (with all options enabled, but without a file system) best reflected current industry practice in certain embedded realtime areas. Instead of full file system support, basic device I/O (read, write, open, close, control) is considered sufficient for kernels of this size. Systems of this size frequently do not include process isolation hardware or software; therefore, multiple processes (as opposed to threads) may not be supported.” (Page 36).

[1]Specifications in this note are drawn from P1003.13 Draft 9 of the POSIX ”Draft Standard for Information Technology – Standardized Application Profile – POSIX Realtime Applications Support (AEP).” (IEEE Std P1003.13-1999x) and from the Single UNIX Specification available from http://www.opengroup.com.

2.2 Motivation

The switch to a POSIX style API was prompted by limitations of the original API, by user demand, and by our intention to simplify mixed mode hard realtime/non-realtime programming in a POSIX environment. As one benefit, we believe that the new API will make it relatively easy to emulate the RTL environment from within a standard POSIX threads environment so RTLinux code can be debugged in user mode as much as possible.

In implementing the POSIX API, we have made an effort to distinguish between critical components and compatibility components based on an analysis of the wide range of aplications including many implemented under the original, very simple, RTLinux API. Critical components are components of the POSIX API that must be present for tightly designed hard realtime applications. Compatibility components are components that facilitate porting applications, but that are not otherwise useful. As an example, POSIX real-time signals are compatibility components: they are extremely useful for porting an application that uses them, but applications written from scratch do not need them. Our guiding principle was that users of RTLinux should not be required to pay a performance, programming, or memory penalty for compatibility components that were part of POSIX, but not absolutely necessary for realtime applications. Some users will need full POSIX conformance in order to port applications developed on other operating systems or due to non-technical constraints, and we want to make this as convenient as possible. On the other hand, native RTLinux applications, using a simpler API should not be required to use or even load all the system support needed for full POSIX compliance. POSIX compatibility cannot be allowed to degrade performance and that POSIX components that are inherently performance challenged must be optional. That is, if you must use clumsy POSIX constructs, RTLinux will support you, but you are not required to do so.

2.3 Scheduling

The V2 scheduling module treats a a scheduler and a collection of realtime tasks as a single POSIX process, with tasks corresponding to POSIX threads. On an SMP system, a cluster, we may have multiple schedulers running in parallel and each one looks like a POSIX process. These ”processes” can share address space and even scheduler code on an SMP machine, but they are logically distinct. The scheduling module does not provide any support for moving threads from one ”process” to another and offers very limited support for controlling threads in one process from another process. Additional cross-processor control can be easily obtained, if needed, by using the primitives described below. Efficient cross-processor control is a difficult problem and we were reluctant to introduce hidden synchronzation points or operations that would cause significant cache disruption — e.g by migrating tasks between processors.

2.4 Posixio

RTLinux Version 2.0 also introduces a `posixio` module which provides a standard interface for drivers with POSIX style read/write/open I/O operations. The obvious difficulty for supporting the POSIX file API is that `open` in a POSIX filesystem is inherently non-realtime. Opening a file may require an unbounded traversal of the namespace, following symbolic links, resolving directories, and crossing mount points. Fortunately, the authors of the POSIX standard permit a very limited version of `open` in the minimal system. Section 6.3 ”Rationale” in the POSIX AEP document states:

> ”The open function is needed to do basic device I/O, also to provide device initialization.” ”Although this requires some form of name resolution, a full pathname is specifically not required. Directories are also not required.” (6.3.1.3 Files and directories)

The only pathnames supported in RTLinux are of the form "/dev/name". The only mode supported is read/write.

2.5 Signals

One area of the POSIX standard we have not yet implemented is the requirement for asynchronous signals.

> [M]ost POSIX.1 process primitives to not apply.
>
> *but*
>
> "Signal services are a basic mechanism with POSIX based systems and are required for error and event handling" (6.3.2.1 Realtime signals)

Our current intention is to make signals an optional component. The utility of general asynchronous signalling mechanism in a hard realtime environment is not at all clear. The purpose of such a mechanism is to interrupt the control flow of a thread and force it to an error or event handling routine. Our belief is that the need for such a facility is an indication of a design error in the application. Realtime tasks should do some simple operation. They must be deterministic and predictable. Thus, event handling via signals seems pointless. Events can be handled by event handlers — hardware interrupt handlers or software event functions. If an event can abort a long running operation, the event handler should suspend the task and possibly call the scheduler. What about errors, such as floating point errrors? RTLinux design Rule Number One is that nonrealtime services should not be provided in the realtime component. If a realtime task uses FP, it should explicitly check for errors rather than allowing for a complex worst case FP interrupt.

Thus, we consider realtime signals to be a compatibility component not a core component. On the other hand, RTLinux does offer much of the signal capability in other forms. There are routines in the API to suspend tasks, to awaken tasks, and to install and remove interrupt and other event handlers.

3 Fundamental operations and timers

The RTLinux *core* provides operations to install, free, and direct, realtime interrupts and "soft" interrupt facility for communicating with the Linux non-realtime kernel. The prototypes for the core functions are in `rtl/include/rtl_core.h` . These are low level operations that are seen by device drivers and the scheduler, but not by tasks or interrupt handlers.

The code for a simple interrupt handler module might include the following code to set up interrupt handling.

```
int init_module(void)
{
        rtl_irqstate_t f;
        rtl_no_interrupts(f); /* disable interrupts on this CPU */
        if (!rtl_request_irq(MY_DEVICE_IRQ_NUMBER,my_handler) {
                do some hardware initialization
                rtl_hard_enable_irq(MY_DEVICE_IRQ_NUMBER);
        }
        rtl_restore_interrupts(f);
}
```

The API for controlling the interrupt hardware is given below. We make a distinction between *global* interrupts that come in from shared devices and *local* interrupts that are either local to the processor (such as an on-chip timer) or handled by a local interrupt controller (such as the IPIs on Intel multiprocessors). Local and global irq numbers may overlap. Where local/global is not specified, we assume global.

- Mode control.
  - `extern void rtl_make_rt_system_active(void);`
  - `extern unsigned int rtl_rt_system_is_idle(void);`
  - `extern void rtl_make_rt_system_idle(void);`

  These are used by schedulers to indicate that a realtime task is active, so that Linux interrupts should not be allowed to invoke Linux interrupt handlers or to indicate that no task is active and calling Linux handlers is ok.

- Special SMP operations.
  - `extern void rtl_reschedule(unsigned int cpu_id);` Generate an interrupt to a processor, causing the processor to run the realtime scheduler. This is an SMP only routine.
  - `extern unsigned int rtl_getcpuid(void);` Return the current processor id.

- Communication with Linux interrupt handlers.
  - `extern int rtl_get_soft_irq( void (*handler)(int, void *, struct pt_regs *), const char * devname);` Allocate and install a handler for a soft interrupt so that realtime tasks may send "interrupts" to Linux handlers.
  - `extern int rtl_free_soft_irq(int irq);`
  - `extern void rtl_global_pend_irq(unsigned int irq)` and
    `extern void rtl_local_pend_vec(int intvec,int cpu_id);`
    Send Linux a soft interrupt.

- Requesting and freeing hardware interrupts.
  - `extern int rtl_request_global_irq(unsigned int irq, unsigned int (*handler)(unsigned int, struct pt_regs *));` aka `rtl_request_irq(x, y)`
  - `extern int rtl_free_global_irq(unsigned int irq);` aka
  - `rtl_free_irq(x)`
  - `int rtl_request_local_vec(int v, unsigned int (*handler)(struct pt_regs *r), unsigned int cpu);`
  - `int rtl_free_local_vec(int v, unsigned int cpu);`

  If local/global is not specified, we assume global. Vectors and irq numbers are just identifiers and can overlap but still refer to different events. For example, vector 0 and irq 0 are different events.

- Hardware enable and disable on a per interrupt basis.
  - `int rtl_hard_enable_local_vec(int v);`
  - `int rtl_hard_disable_local_vec(int v);`

  - `int rtl_hard_enable_global_irq(int i);` aka `int rtl_hard_enable_irq(int i);`
  - `int rtl_hard_disable_global_irq(int i);` aka `int rtl_hard_disable_irq(int i);`

In addition, RTLinux provides a set of primitive synchronization methods to disable and enable interrupts and to use spinlocks in a multiprocessor environment. Semaphores are not primitive in RTLinux for several reasons. One of the reasons is that there is a perfectly usable package originally written by Jerry Epplin that provides semaphores. A second reason is that semaphores are not essential for many realtime applications and the RTLinux design philosophy is that applications should not pay a price for features that they do not use. Finally, there are some intrinsic difficulties with the use of semaphores guarded critical regions that make semaphores a less than optimal design choice in many situations. RTLinux will add alternative mechanisms in future releases.

```
#include <rtl_sync.h>
```

The primitives are:

- `rtl_allow_interrupts()` Allows hardware interrupts on the current processor.
- `rtl_stop_interrupts()` Forbids hardware interrupts on the current processor.
- `rtl_no_interrupts(x)` Forbids hardware interrupts and saves the current interrupt state on the current processor.
- `rtl_save_interrupts(x)` Saves the hardware interrupt state on the current processor.
- `rtl_restore_interrupts(x)` Restores the hardware interrupt state on the current processor.
- `rtl_spin_lock(x)` and `rtl_spin_unlock(x)` operate on pointers to `spinlock_t`.
- `rtl_spin_lock_irqsave(x, flags)` Saves state in `flags`, disables interupts on the current processor, and sets a spinlock in `x` which must be a pointer to a `spinlock_t` .
- `rtl_spin_unlock_irqrestore(x, flags)` Does the obvious: making sure to clear the lock first!
- `rtl_critical(f)` For this to work, the file must contain a `spinlock_t my_spinlock;` and `#define RTL_SPINLOCK my_spinlock`. Then `rtl_critical(f))` resolves to `rtl_spin_irqsave(&my_spinlock,f)`
- `rtl_end_critical(f)`

3.1 Timer

The timer module exports a ”clock structure” as a low level interface for schedulers and other below pthreads modules. The timer also defines operations on these clocks in nanoseconds.

- `struct rtl_clock *rtl_getbestclock (unsigned int cpu);` This finds the ”best” system clock for scheduling RT tasks on this CPU and returns a token allowing operations on the clock.
- `int rtl_setclockhandler (clockid_t, clock_irq_handler_t);`
- `int rtl_unsetclockhandler (clockid_t);`

- `typedef void ( *clock_irq_handler_t)(struct pt_regs *r);`

The rtl_time.h file also defines some arithmetic operations on timespec structures.

```
timespec_gt(t1, t2)
timespec_ge(t1, t2)
timespec_le(t1, t2)
timespec_eq(t1, t2)
timespec_add(t1, t2)
long long timespec_to_ns (struct timespec *ts)
timespec_sub(t1, t2)
timespec_nz(t)
timespec_lt(t1, t2)
struct timespec timespec_from_ns (long long t)
```

Most operations are via the rtl_clock structure itself. This structure is described here to make the explanation concrete, but there is no guarantee that the time structure will remain unchanged.

```
struct rtl_clock {
  int (*init) (struct rtl_clock *);
  void (*uninit) (struct rtl_clock *);
  hrtime_t (*gethrtime)(struct rtl_clock *);
  int (*sethrtime)(struct rtl_clock *, hrtime_t t);
  int (*settimer)(struct rtl_clock *, hrtime_t interval);
  int (*settimermode)(struct rtl_clock *, int mode);
  clock_irq_handler_t handler;
  int mode;
  hrtime_t resolution;
  hrtime_t value; /* only makes sense for periodic clocks */
  struct rtl_clock_arch arch;
};
typedef struct rtl_clock * clockid_t;
enum { RTL_CLOCK_MODE_UNINITIALIZED = 1, RTL_CLOCK_MODE_ONESHOT,
RTL_CLOCK_MODE_PERIODIC};
```

4 Scheduler and pthreads

```
rtl/schedulers/rtl_sched.c
rtl/include/rtl_sched.h
```

The RTLinux schedulers define realtime tasks and provide basic operations on those tasks. In what follows we will write about “the RTLinux scheduler” for brevity, but please keep in mind that RTLinux does not mandate the use of any particular scheduler. It is quite simple to replace the default scheduler modules with an alternative. The API for Version 1 RTLinux is given in the appendix. Version2 provides POSIX Pthread and some non-POSIX pthread operations The non-POSIX calls have names `function_name_np` where the `_np` means — *non-POSIX* or *non-portable*.

In RTLinux, there are 3 important execution modes: user (u), kernel (k), and rt (r) in RTLinux User and kernel modes are the standard Linux user/kernel modes and when the Linux task is running, the processor must be in one of those modes. When a RT task or interrupt handler is running, the processor is in RT mode. In an SMP system, processors change modes independently. Most of the calls in the RTL API are ”RT safe” and may be called while in RT mode, but certain calls must only be run in Linux mode. These are generally calls that can introduce unbounded delays and/or need Linux kernel resources.

- `int pthread_create (pthread_t *p, pthread_attr_t *a, void *(*f)(void *), void *x)`
  Create a thread that will run code `f` passing it argument `x`. The thread starts execution immediately. Only works in kernel mode.

- `void pthread_exit(void *retval);` Exit. Only works in RT mode.

- `int pthread_delete_np (pthread_t thread);` Stops the thread and deallocates this thread's resources.

- `int pthread_setfp_np (pthread_t thread, int allow_fp_flag);` Allows or disallows this thread to use floating point.

- - `int pthread_wakeup_np (pthread_t thread);` Kernel or RT mode.
  - `int pthread_suspend_np (pthread_t thread);` RT mode only. Suspends the thread until it is woken up by a wakeup call.
  - `int pthread_wait_np(void);` RT mode only. Suspends this thread until its next period.

  POSIX does not provide a clean mechanism for suspending and waking up threads. Instead POSIX wants to programmers to use the signal mechanisms or the conditional variables. Both signals and condition variables/mutexes are rather complex and we do not want to make using them a requirement for RTLinux. So these analogs of the Version1 RTLinux API are primitives in RTLinux V2. These calls can be implemented on top of signals if you want to run in user mode.

- `int pthread_make_periodic_np(pthread_t p, hrtime_t start_time, hrtime_t period);` POSIX does not provide a simple method for adjusting periods. Instead, POSIX allows threads to create timers and use the timers to schedule a task periodically. The Version2 RTLinux scheduler controls the timers and currently does not permit threads to create or otherwise manipulate timers.

- - `int sched_get_priority_max(int policy);`
  - `int sched_get_priority_min(int policy);` Obvious. The default scheduler is pure priority driven.

- As noted above, default scheduler is pure priority driven and the set call is ignored. The POSIX RR or FIFO scheduling policies are compatibility components. In fact, it makes little sense to design a hard realtime system using either of these scheduling policies and the POSIX specification semantics is so loose as to make their utility unclear. We will happily add schedulers that implement these policies, but they seem less useful than the current pure priority/time driven scheduler or simpler policies we plan to provide later.

  - `int pthread_setschedparam(pthread_t thread, int policy,const struct sched_param *param)`
  - `int pthread_getschedparam(pthread_t thread, int *policy,const struct sched_param *param)`

  The priorities are in the param structures and this is standard POSIX.

- Attributes. Attributes are optional calls as the scheduler will use default values if no attribute is provided to `pthread_create`. All calls to set attribute values are only valid in kernel mode.

  - `int pthread_attr_init(pthread_attr_t *attr)`

  - `int pthread_attr_getstacksize(pthread_attr_t *attr, size_t * stacksize)`
  - `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)`
  - `int pthread_attr_getcpu_np(pthread_attr_t *attr, int * cpu)`
  - `int pthread_attr_setcpu_np(pthread_attr_t *attr, int cpu)` These last two are non-POSIX extensions to support SMP. If you do not set this specifically, the task is created, by default, on the current processor.

- Clock operations. Threads have no clocks. Clocks belong to schedulers. We can add scheduling policies in which a scheduler juggles multiple clocks, but there is no obvous advantage that in allowing a thread to specify its hardware clock On the other hand, POSIX ”clocks” are not necessarily hardware clocks.
  - `extern int clock_gettime(clockid_t clock, struct timespec *tsptr);`
  - `extern int clock_settime(clockid_t clock, struct timespec *tsptr);`
  - `extern int clock_getres(clockid_t clock, struct timespec *tsptr);`
  - `extern hrtime_t gethrtime(void);`
  - `int rtl_setclockmode (clockid_t clock, int mode, hrtime_t period);`

Here are some basic POSIX calls that have not been implemented yet.

- `void pthread_yield(void)`
- `int timer_create(clockid_t clockid, void /*struct sigevent*/ *evp, timer_t *timerid);`
- `int timer_delete(timer_t timerid);`
- `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);`

Type: `pthread_t` is a pthread identifier. The current implementation defines `pthread_t` as a pointer to a thread structure, but this may change later. `pthread_t` is intended to be opaque to RTLinux application programmers.

5 Posixio and drivers

The `rtl_posixio` module provides a file system like interface to drivers. The basic operations for drivers are to register and unregister devices. The calls here are variations on the standard Linux calls, but they also provide a string identifier — the file system is essentially part of the device table.

- `extern int rtl_register_chrdev(unsigned int, const char *, struct rtl_file_operations *);`
- `extern int rtl_unregister_chrdev(unsigned int major, const char * name);`

Rtl_posixio also provides open close, read, write, ioctl and
tt mmap calls. As mentioned above, open will only open files named /dev/filenameNN where filename is the name provided to the register call by the driver and NN is an optional device minor number.

Perhaps the simplest example of a posixio driver is the physical memory device driver, /dev/mem. This driver allows real-time threads to access physical computer memory in the same way as Linux processes can do. A real-time process opens /dev/mem, and then performs an mmap(2) call on it. After that, the physical memory area at the requested offset can be accessed by using the address returned by mmap.

The implementation of this driver is currently trivial, because all physical memory is mapped to the kernel address space at a constant offset, and real-time threads are also executed in the kernel address space. However, using /dev/mem is preferable for two reasons. First, doing so will allow debugging real-time threads as Linux threads first. Second, future implementations of real-time Linux may provide optional memory protection for real-time tasks.

6 The fifo driver

The RTLinux fifo driver is a good illustration of the use of the POSIX I/O features and is important on its own. The fifo driver fields requests from both Linux processes and from RT tasks and handlers. The interface for Linux processes is the standard device interface of open,close, read and write. In fact, on startup the fifo driver (running in Linux kernel mode) asks Linux to register a character device with the standard table of operations so that Linux processes can treat fifos as ordinary character devices.

```
/* this table is a standard Linux table for
   character device operations
   with RTL fifo operations in the correct slots
*/
static struct file_operations rtf_fops =
{
        rtf_llseek,
        rtf_read,
        rtf_write,
        NULL,
        rtf_poll,
        NULL,
        NULL,
        rtf_open,
        NULL,
        rtf_release,
        NULL,
        NULL,
        NULL,
        NULL,
        NULL
};
```

The fifo driver also needs to provide an API to RT code. This API is configurable. If the configuration variable CONFIG_RTL_POSIX_IO is set, the fifo driver will register a RTLinux device with the Posixio module and then offer read/write/open to RT components. Read and Write are non-blocking.

If `CONFIG_RTL_POSIX_IO` is not set, the fifo module will export primitive access routines.

```
int rtf_create(unsigned int minor, int size)
int rtf_destroy(unsigned int minor)
int rtf_put(unsigned int minor, void *buf, int count)
int rtf_get(unsigned int minor, void *buf, int count)
```

Each fifo can have a handler installed via the calls:

```
int rtf_create_handler(unsigned int minor, int (*handler) (unsigned int fifo))
```

This handler is executed in the Linux kernel context whenever a Linux process reads or writes the FIFO.

7 A serial port driver: rt_com

Another illustration of the posixio capabilities is the rt_com driver. This is a PC serial port driver, originally written by Jens Michaelsen and Jochen Küpper. It has been modified to provide POSIX IO interface for real-time threads.

For example, to access the first serial port, a real-time thread opens `/dev/ttyS0`, and proceeds to use `read(2)` and `write(2)` operations on the device. The support is planned for POSIX `termios` family of functions which will allow changing baud rates, byte sizes, and other communication parameters in a standard way.

8 A shared memory driver: mbuff

This driver, written by Tomasz Motylewski [2], provides means to share memory between kernel and user address spaces. It is ideally suited for providing shared memory between RT components and Linux processes.

The mbuff driver supports the same set of operations for both kernel components and Linux processes:

```
#include <mbuff.h>

void * mbuff_alloc(const char *name, int size);
void mbuff_free(const char *name, void * mbuf);
```

The first time mbuff_alloc is called with the given name, a shared memory block of the specified size is allocated. The reference count for this block is set to 1. On success, the pointer to the newly allocated block is returned. NULL is returned on failure. If the block with the specified name already exists, this function returns the pointer that can be used to access this block and increases the reference count.

mbuff_free deassociates mbuf from the specified buffer. The reference count is decreased by 1. When it reaches 0, the buffer is deallocated.

[2] `motyl@stan.chemie.unibas.ch`

A Version 1 RTLinux Scheduler API

Version1 RTLinux provided the following operations in the default scheduler.

- `rt_get_time` returns the time in “ticks”.
- `rt_task_delete` destroys a task and frees its resources.
- `rtl_task_init` sets up, but does not schedule a task.
- `rt_task_make_periodic` asks the periodic scheduler to the run task at a fixed period (given as a parameter).
- `rt_task_suspend` takes the task off the run queue.
- `rt_task_wait` yields the processor until the next time slice for this task.
- `rt_task_wakeup` wakes up a suspended task.
- `rt_use_fp` allows the task to use floating point operations.
- `rtl_set_periodic_mode` optimizes the system for running a collection of tasks that share a common fundamental period.
- `rtl_set_oneshot_mode` optimizes the system for cases where periodic mode is not appropriate.

B Sysconf

`long int sysconf(int name);` ”Conforming applications must act as if CHILD_MAX == 0” – which it will be in RTL. (I have no idea what most of these are. VY).

RG_MAX _SC_ARG_MAX
BC_BASE_MAX _SC_BC_BASE_MAX
BC_DIM_MAX _SC_BC_DIM_MAX
BC_SCALE_MAX _SC_BC_SCALE_MAX
BC_STRING_MAX _SC_BC_STRING_MAX
CHILD_MAX _SC_CHILD_MAX
CLK_TCK _SC_CLK_TCK
COLL_WEIGHTS_MAX _SC_COLL_WEIGHTS_MAX
EXPR_NEST_MAX _SC_EXPR_NEST_MAX
LINE_MAX _SC_LINE_MAX
NGROUPS_MAX _SC_NGROUPS_MAX
OPEN_MAX _SC_OPEN_MAX
PASS_MAX _SC_PASS_MAX (LEGACY)
_POSIX2_C_BIND _SC_2_C_BIND
_POSIX2_C_DEV _SC_2_C_DEV
_POSIX2_C_VERSION _SC_2_C_VERSION
_POSIX2_CHAR_TERM _SC_2_CHAR_TERM
SIX2_FORT_DEV _SC_2_FORT_DEV
_POSIX2_FORT_RUN _SC_2_FORT_RUN
_POSIX2_LOCALEDEF _SC_2_LOCALEDEF
_POSIX2_SW_DEV _SC_2_SW_DEV
_POSIX2_UPE _SC_2_UPE
_POSIX2_VERSION _SC_2_VERSION
_POSIX_JOB_CONTROL _SC_JOB_CONTROL
_POSIX_SAVED_IDS _SC_SAVED_IDS
_POSIX_VERSION _SC_VERSION
RE_DUP_MAX _SC_RE_DUP_MAX
STREAM_MAX _SC_STREAM_MAX
TZNAME_MAX _SC_TZNAME_MAX
_XOPEN_CRYPT _SC_XOPEN_CRYPT
_XOPEN_ENH_I18N _SC_XOPEN_ENH_I18N
_XOPEN_SHM _SC_XOPEN_SHM
_XOPEN_VERSION _SC_XOPEN_VERSION
_XOPEN_XCU_VERSION _SC_XOPEN_XCU_VERSION
_XOPEN_REALTIME _SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS _SC_XOPEN_REALTIME_THREADS
_XOPEN_LEGACY _SC_XOPEN_LEGACY
ATEXIT_MAX _SC_ATEXIT_MAX
IOV_MAX _SC_IOV_MAX
PAGESIZE _SC_PAGESIZE
PAGE_SIZE _SC_PAGE_SIZE
_XOPEN_UNIX _SC_XOPEN_UNIX
_XBS5_ILP32_OFF32 _SC_XBS5_ILP32_OFF32
_XBS5_ILP32_OFFBIG _SC_XBS5_ILP32_OFFBIG
_XBS5_LP64_OFF64 _SC_XBS5_LP64_OFF64
_XBS5_LPBIG_OFFBIG _SC_XBS5_LPBIG_OFFBIG
AIO_LISTIO_MAX _SC_AIO_LISTIO_MAX
AIO_MAX _SC_AIO_MAX
AIO_PRIO_DELTA_MAX _SC_AIO_PRIO_DELTA_MAX
DELAYTIMER_MAX _SC_DELAYTIMER_MAX
MQ_OPEN_MAX _SC_MQ_OPEN_MAX
MQ_PRIO_MAX _SC_MQ_PRIO_MAX
RTSIG_MAX _SC_RTSIG_MAX

SEM_NSEMS_MAX _SC_SEM_NSEMS_MAX
SEM_VALUE_MAX _SC_SEM_VALUE_MAX
SIGQUEUE_MAX _SC_SIGQUEUE_MAX
TIMER_MAX _SC_TIMER_MAX
_POSIX_ASYNCHRONOUS_IO _SC_ASYNCHRONOUS_IO
_POSIX_FSYNC _SC_FSYNC
_POSIX_MAPPED_FILES _SC_MAPPED_FILES
_POSIX_MEMLOCK _SC_MEMLOCK
_POSIX_MEMLOCK_RANGE _SC_MEMLOCK_RANGE

C Random comments on the POSIX Rationale

C.1 6.3.1.2 Process Environment

Posix requires ”sysconf” and ”uname” but we don’t offer these yet. See appendix B for a description of the *many many* sysconf parameters. Perhaps uname can simply read the Linux data. It seems harmless.

C.2 6.3.1.4 Input and output

”The functions `read`, `write`, and `close` are required to do basic i/O and device cleanup.”

See above.

C.3 6.3.1.5 Device and class specific functions

”POSIX.1 Device of Class-Specific functions are not required.”

C.4 6.3.1.6 Language specific

C must be supported. We do this!

C.5 6.3.1.7 System databases

”Implementations are not required to support more than one user and group id ... No POSIX.1 system database functions are required.”