Real Time Threads Interface[1]

David Finkelstein, Norman C. Hutchinson, Dwight J. Makaroff,
Roland Mechler and Gerald W. Neufeld
Department of Computer Science
University of British Columbia
Vancouver, B.C., V6T 1Z4
Canada

Abstract

The Real Time Threads package (abbreviated RT Threads) provides a user-level, preemptive kernel running inside a single address space (e.g., within a UNIX process). RT Threads implements thread management, synchronization, and communication functions, including communication between RT Threads environments (i.e., with different address spaces, possibly on different machines and different architectures). Threads are scheduled using a real-time, multi-priority, preemptive scheduling algorithm. Each thread is scheduled on the basis of its modifiable scheduling attributes: starting time, priority and deadline. No thread is scheduled before its starting time. Schedulable threads (i.e., threads whose starting time has passed) are scheduled on a highest priority first basis. Schedulable threads of equal priority use an earliest deadline first (EDF) scheduling policy. An RT Threads environment is cooperative in the sense that memory is shared among all threads, and each thread runs to completion unless preempted on the basis of priorities and deadlines. Alternate scheduling policies, such as time slicing, can be implemented at the application level using the scheduling mechanisms provided by RT Threads. This report describes the interface to the RT Threads package.

1. This work was supported by grants from the Natural Sciences and Engineering Research Council of Canada and the Canadian Institute for Telecommunications Research.

Table Of Contents

1 Introduction

The Real Time Threads package (abbreviated RT Threads) provides a user-level, preemptive kernel running inside a UNIX process. All threads in a particular RT Threads environment share the same address space and thereby memory is shared among all threads. RT Threads implements thread management, synchronization, and communication functions, including communication between RT Threads environments (i.e., with different address spaces, possibly on different machines and different architectures). Each RT Threads environment is designed to be independent, except for the facility of message passing.

Threads in RT Threads are scheduled using a real-time, multi-priority, preemptive scheduling algorithm. A thread will not execute if there are any ready threads of higher priority. Within the same priority level, processes are scheduled according to their starting times and deadlines. Threads are not required to have deadlines. Threads with equal priority and equal deadlines are scheduled on a FCFS basis and run to completion unless they are blocked. Threads of equal priority but with differing deadlines are scheduled on an earliest deadline first (EDF) basis. Preemption is performed, but only threads of higher priority, or equal priority and earlier deadline, may preempt the currently running thread. For more details on the real-time scheduling algorithms used, see [2]. An individual thread may have data uniquely associated with it for purposes unique to that incarnation of the particular thread.

An underlying philosophy of the RT Threads environment is that of a cooperative environment. No thread should interfere with system resources that have been allocated to other threads, and the system has no exception handling for errors caused by improper resource usage by individual threads.

RT Threads routines which block the execution of individual threads will not block the execution of other schedulable threads. However, non RT Threads system calls and external library routines may block the entire RT Threads environment. Thus any such calls should be made only with extreme caution, and wherever possible equivalent RT Threads routines should be used.

2 Thread Management

An RT Threads application begins execution at the first statement of the routine mainp()[1]. *Mainp()* is passed the environment arguments *argc* and *argv[]*. *Mainp()* is **not** a thread and therefore most kernel calls which would be appropriate when called by threads as defined in this interface are not available from *mainp()*. The normal function of the *mainp()* routine is to create the first few user-defined threads of the environment.

2.1 Thread Scheduling Mechanism

Threads must be scheduled according to these real-time considerations: starting time, deadline and priority.

1. mainp() in RT Threads is analogous to main() for regular C applications.

The structures used are as follows:

```
typedef struct {
     long seconds;
     long microseconds;
} RttTimeValue;
```

The *RttTimeValue* structure represents real time values. When a time structure is to be used, the system call *RttGetTimeOfDay* (described later in this section) can be used to provide a base time value for scheduling purposes. Arithmetic operations can be performed to derive the correct values to place in an *RttTimeValue* data structure. The exact resolution of these time values is system-dependent. Not all systems may be able to provide microsecond resolution. When scheduling threads, any time value will be rounded up to the next highest clock interrupt time (10 ms is a typical value for UNIX syatems). Threads may thus be made ready up to that amount of time later than requested.

```
typedef struct {
     RttTimeValue startingtime;
     u_int        priority;
     RttTimeValue deadline;
} RttSchAttr;
```

An *RttSchAttr* structure provides all information necessary for the thread scheduling algorithm.

- Starting time is the first attribute considered in the scheduling discipline. If starting time is in the future, the thread is not ready and will not be scheduled; otherwise the thread is ready and can be scheduled, depending on the attributes and status of other threads in the system. When time advances so that the starting time is in the past, the thread is awakened.
- Priority is the next attribute considered in the scheduling algorithm and can take on any integer value from 0 to 30 (inclusive), where 0 is the highest priority and 30 is the lowest. It is **highly recommended** that threads use only priorities in the range 10 to 30 inclusive so as not to interfere with higher priority threads used by the underlying system (i.e., threads used to implement cross address-space communication).
- The deadline is the target finishing time for the task. If the task has not completed by the deadline, it continues to run. This implements soft real-time semantics. This value is used to determine the ordering of tasks within a priority level.

If a thread is to start immediately, the starting time should be set to zero (or any time in the past). The following constants provide simple defaults for scheduling attributes that also allow non real-time and suspended tasks to be scheduled.

```
TimeValue    RTTNODEADLINE;
TimeValue    RTTINFINITE;
TimeValue    RTTZEROTIME;
```

Specifying `RTTNODEADLINE` for the deadline results in a thread being scheduled after all threads of equal priority which have a deadline. `RTTINFINITE` is provided as a convenience for specifying a starting time far enough in the future that the thread will not be scheduled to run. `RTTZEROTIME` is a convenience for making a thread ready immediately.

For applications that do not require a fine granularity of priority, the following priority constants are available:

```
RTTHIGH = 10, RTTNORM = 20, RTTLOW = 30
```

The following call allows a thread to find a base time to use for further computations of time values:

(1) *int RttGetTimeOfDay(RttTimeValue *time)* Get the current time in seconds and microseconds.

2.2 Thread Creation

Threads are created using the following call:

(1) *int RttCreate(RttThreadId *thread, void (*addr) (), int stksize, char *name, void *arg, RttSchAttr schedAttr, int level)*

On success, *RttCreate()* returns (by reference) the thread identifier (hereafter called an RttThreadId) of the newly created ready thread. This thread can begin execution when the conditions of its scheduling attributes permit. *RttCreate()* returns `RTTOK` on success and `RTTFAILED` on failure.

- The first parameter, *thread*, is a pointer to an RttThreadId, and is used to return the newly created thread's identifier by reference.
- The second parameter, *addr*, is a pointer to the routine which acts as the entry point for the created thread.

- The *stksize* parameter is the size, in bytes, of the new thread's stack. The stack requirements vary according to the number and size of local variables and parameters, as well as the depth of subroutine calls.

- The fourth parameter, *name*, points to a text string which acts as a user supplied thread identifier. This string must be null terminated (maximum length 32 bytes including the null-terminator). Any number of threads may be identified by the same name. Memory containing the name may be released by the caller on return from *RttCreate()*. *Name* is to be used *strictly for debugging purposes*.

- The fifth parameter, *arg*, is an argument (parameter) for the created thread. This argument is passed transparently to the thread and may be of any (4 byte or smaller) type, although it should be cast to a *void* * on call. The entry routine for the new thread receives this argument as a parameter, or may instead not declare any parameters if no creation-time arguments are required.

- The sixth argument, *schedAttr*, indicates the scheduling attributes of the created thread. The possible thread priorities are as described earlier. If the starting time indicated is in the future, the thread will not be made ready until that time.

- The final argument indicates whether the created thread is a system (`RTTSYS`) or an application (`RTTUSR`) thread. The only difference between the two is that the RT Threads ***process*** (i.e., threads environment) will exit when there are no more `RTTUSR` level threads. Therefore, perpetual server threads should be created with level `RTTSYS` (unless they service requests from other RT Threads environments, in which case they will likely want to use `RTTUSR`).

2.3 Destruction of Threads

Threads terminate in one of three ways. The first possibility is for the thread to call the *RttExit()* routine which causes the thread to leave the system at that point. The second possibility is for the thread to ‘‘fall off the end of’’ (i.e. return from) the entry subroutine, which is equivalent to calling RttExit() as the last statement of the subroutine. The final possibility is for a thread to be killed by some other thread through a call to *RttKill()*. *RttExit()* requires no parameters and returns no result. *RttKill()* requires the RttThreadId as the argument and returns `RTTOK` on success or the value `RTTNOSUCHTHREAD` if the thread to be killed cannot be found. The headers are as follows:

(1) *int RttKill (RttThreadId thread)*

(2) *void RttExit()*

2.4 Rescheduling Threads

The scheduling attributes can be obtained and modified using the following system calls: *RttGetThreadSchedAttr()* and *RttSetThreadSchedAttr()*. A thread may access its own scheduling attributes or those of another thread using these calls.

Changing a thread's attributes using *RttSetSchedAttr()* can have an immediate effect on scheduling. For example, a thread can be put to sleep by setting its starting time to some time in the future. The other attributes which can be changed are priority and deadline.

(1) *int RttSetThreadSchedAttr(RttThreadId thread, SchAttr schedAttr)* Change the scheduling attributes of a thread. If there should be a thread context switch resulting from this change, the switch is done immediately.

(2) *int RttGetThreadSchedAttr(RttThreadId thread, SchAttr *schedAttr)* Obtain the current values of the thread's scheduling attributes.

RttSetThreadSchedAttr() sets all of a thread's attributes. If only some of the attributes are to be changed, a prior call to *RttGetThreadSchedAttr()* is advisable to determine the current values of those attributes which are not to be changed.

2.5 Thread Identification

RT Threads provides two routines to identify threads. These are *RttMyThreadId()* and *RttThreadExists()*. *RttMyThreadId()* requires no parameters and returns the RttThreadId of the calling thread. *RttThreadExists()* takes an RttThreadId as its only argument and returns 1 if a thread with the given identifier exists, and 0 otherwise.

(1) *RttThreadId RttMyThreadId()*

(2) *int RttThreadExists(RttThreadId thread)*

2.6 Exiting an RT Threads Environment

When all user level threads (those with level `RTTUSR`) are done, an RT Threads environment will exit (i.e., the corresponding process will exit). The application can register routines which are to be called just prior to exiting. Registered exit routines are called in the order in which they were registered.

(1) *int RttRegisterExitRoutine(void (*routine)())* registers *routine* (which takes no parameters) to be called when RT Threads exits.

An application may also exit using *exit()*, but the exit routines will not be called.

3 Thread Synchronization

3.1 Semaphores

Counting semaphores are provided as the primary means of thread synchronization. Two attributes are specified by the user when allocating a new semaphore, *value* and *mode*. *Value* indicates the number of times *RttP()* (see below) can be called before a calling thread will block, assuming no *RttV()* calls are made. *Mode* indicates the desired semantics for the order in which semaphore blocked threads are made ready. `RTTFCFS` specifies first come first served semantics, in which threads are made ready in the order in which they were blocked, regardless of priority. `RTTPRIORITY` specifies highest priority first semantics, in which threads of higher priority will be made ready before threads of lower priority, with threads of equal priority being made ready in earliest deadline first order. Threads of equal priority and equal deadline are made ready in first come first served order in `RTTPRIORITY` mode.

Since memory is shared within an RT Threads environment, semaphores can easily be shared among threads within a single environment. In keeping with the cooperative nature of an RT Threads environment, it is left up to the user to ensure semaphores are no longer in use when a semaphore is freed. To help detect such a situation, an attempt to free a semaphore on which other threads are blocked will result in failure. In the event that a thread is killed while blocked on a semaphore, the state of the semaphore is automatically modified (its value is incremented).

RT Threads provides five semaphore operations *RttAllocSem(), RttFreeSem(), RttP(), RttV()* and *RttSemValue().* Semaphore variables are of type *RttSem.*

The headers for these routines are as follows:

(1) *int RttAllocSem(RttSem *sem, int value, int mode)* allocates a new semaphore. *Sem* is a pointer to type *RttSem* and returns the new semaphore by reference. *Value* is the numeric value to which the new semaphore is to be initialized. *Mode* allows one of two possible choices of semantics to be specified:

`RTTFCFS` - first come first served semantics for unblocking threads

`RTTPRIORITY` - highest priority first semantics for unblocking threads

(2) *int RttFreeSem(RttSem sem)* returns a previously allocated semaphore to the

system. Its single argument *sem* is the identifier of the semaphore to be freed. *RttFreeSem* will return `RTTFAILED` if any threads are blocked on the semaphore.

(3) *int RttP(RttSem sem)* implements the conventional semaphore wait primitive.

(4) *int RttV(RttSem sem)* implements the conventional semaphore signal primitive.

(5) *int RttSemValue(RttSem sem, int *value)* returns the current semaphore value via reference parameter *value*.

RttFreeSem(), RttP(), *RttV() and RttSemValue()* each take the parameter *sem* to identify the semaphore. Each routine returns `RTTOK` on success and `RTTFAILED` if *sem* does not identify an active semaphore.

The package also includes the routine *RttNewSem(RttSem *sem, int value)* for backward compatibility. Its use is equivalent to calling *RttAllocSem()* with mode `RTTFCFS`.

4 Communication

An inter-thread communication facility is provided allowing message passing between threads both within an RT Threads environment, and between threads in different environments. A single interface for both types of communication is made possible because RttThreadIds are universally unique.

4.1 Intra-Address Space Communication

RT Threads provides blocking send/receive/reply style inter-thread communication primitives. A sending thread is blocked until the message has been received and a reply has been made. A receiving thread is blocked until some other thread sends it a message. *RttReply()* is a non-blocking operation. There is also a non-blocking routine *RttMsgWaits()* that allows a thread to test whether it has a message waiting for it to receive. The subroutine headers and error codes are as follows:

Possible error codes:

`RTTNOSUCHTHREAD` - no such destination thread

`RTTNOTBLOCKED` - specified thread is not blocked awaiting a reply

(1) *int RttSend(RttThreadId to, void *sData, u_int slen, void *rData, u_int *rlen)* has five parameters. The first one, *to* is the thread identifier of the intended recipient. The remainder are pairs of address and length values, the first pair

for the message to be sent and the second pair for the message to be received when *RttSend()* returns. *RttSend()* returns a status value, which is `RTTOK`, `RTTFAILED`, or `RTTNOSUCHTHREAD`. **Important note**: *rlen* is an input/output parameter whose dereferenced value must be set upon procedure call to the maximum expected length for the reply message. This length in bytes must also be allocated for *rData* prior to the call.

(2) *int RttReceive(RttThreadId *from, void *data, u_int *len)* requires three parameters. The argument *from* is a reference parameter which returns the RttThreadId of the sending thread. The argument *data* is a pointer to a buffer into which the message should be received. The memory for this buffer must be allocated by the application. The argument *len* is an input/output parameter whose dereferenced value must be set upon procedure call to the maximum expected length for the received message. Upon return, *len* will be set to the actual length of the received message. If the input value of *len* is less than the length of the received message, the message will be truncated.

(3) *int RttReply(RttThreadId sndr, void *data, u_int len)* requires three parameters. The argument *sndr* is the thread identifier of the reply destination (i.e. the original sender). The remaining parameters are *data*, which is a pointer to the reply message, and *len*, which is the length of the reply message. *RttReply()* returns `RTTOK` or `RTTFAILED`, or `RTTNOTBLOCKED` (when the destination thread was not blocked on a send).

(4) *int RttMsgWaits()* requires no parameters and returns 1 if there are messages waiting to be received by the calling thread, or 0 if no such messages currently exist.

The interface routines described in this section may also be used across address spaces after appropriate initializations are made to make thread identifiers visible outside an address space. These mechanisms are described in the following section.

4.2 Inter Address-Space Communication

The routines described in this section can be used to allow communicating threads to be in different address spaces (including on different machines). Thread identifiers can be passed across address spaces as return values of procedure calls or in messages sent by the applications. Alternatively, a name service can be established to register and lookup the RttThreadIds for remote threads.

Whenever a threads environment wishes to establish a port for communication in any manner with other address spaces, the call *RttNetInit()* is essential. It can only occur in *mainp()* and **must come**

before any calls to *RttCreate()*

(1) *int RttNetInit(unsigned int ipAddr, unsigned int portNo)*

RttNetInit() uses parameters *ipAddr* and *portNo* to set the IP address and port number to be used by the threads environment for communication with other threads environments (address spaces). A 32-bit unsigned integer representation is used for the IP address. If 0 is specified for *ipAddr*, RT Threads will map the IP address to be used (if the machine has more than one IP address, the chosen IP address may differ from the desired one). The chosen port number must be unique among threads environments on a given machine. If a value of 0 is specified for *portNo*, the system will choose a port number. The values of the chosen IP address and port number can be accessed using the routines *RttMyIP() and RttMyPort()*, whose headers are as follows

*(1) int RttMyIP(unsigned int *ipAddr)*

*(2) int RttMyPort(unsigned int *portNo)*

4.3 Name Service Functionality

Cross-address space communication requires a means to determine the RttThreadIds of threads in other RT Threads environments. This is typically done using a name service. RT Threads provides the facilities for implementing such a name service, by allowing the user to create a *name server* thread using the *RttCreateNS()* call. There must be only one name server thread per RT Threads environment. The RttThreadId for the name server thread can be determined from any other RT Threads environment using the *RttGetNS()* call, which requires only that the IP address and port number of the target environment be known. The headers for these routines are shown below:

(1) *int RttCreateNS(RttThreadId *thread, void (*addr) (), int stksize, char *name, void *arg, RttSchAttr schedAttr, int level)* takes the same parameters as *RttCreate()* and creates a name server thread.

(2) *int RttGetNS(unsigned int ipAddr, int portNo, RttThreadId *nServThreadId)* returns, by reference, the ThreadId of the name server for an RT Threads environment specified by IP address *ipAddr* and port number *portNo*.

An RT Threads environment which wishes to make the RttThreadIds of any of its threads known to other environments will typically implement a name server thread which maintains a name-to-thread mapping and services requests using cross-address space communication, returning RttThreadIds in the reply. The following example shows a possible algorithm for the name server thread:

```
nameserver()
```

```
repeat
   receive message
   case message type of
      register: add a thread/name pair to the name server table
      deregister: remove a thread/name pair from the name server table
      lookup: given a thread name, reply with the corresponding RttThreadId
   end case
end repeat
```

The routine

> *(1) bool_t xdr_RttThreadId(XDR *xdrs, RttThreadId *objp)*

is provided so that RttThreadIds being sent in a message between address spaces can be encoded and decoded using XDR external data representation.

5 Memory Allocation

Memory can be allocated for the address space and returned to the system by the following calls:

(1) *void *RttMalloc(int size)* allocates and returns a pointer to at least *size* bytes of memory.

(2) *void *RttCalloc(int numElems, int elemSize)* allocates space for and returns a pointer to an array of *numElem* elements of size *elemSize*, initializing the space to zeros.

(3) *void *RttRealloc(void *ptr, int size)* changes the size of the memory block pointed to by *ptr*, returning a pointer to the (possibly moved) block. The contents will be unchanged up to the lesser of the new and old sizes.

(4) *void RttFree(void *mem)* frees memory allocated with any of the above routines.

If a thread which has allocated memory using one of these routines exits or is killed before that memory is freed, the memory will not be freed until another thread does so.

6 Thread-Specific Data

Interface routines are provided that allow a specific piece of data to be associated with a given thread. The parameter data may be a pointer to the memory block used for such an association.

> *(1) int RttSetData(RttThreadId thread, unsigned long data)*

> *(2) int RttGetData(RttThreadId thread, unsigned long *data)*

Both of these routines return `RTTOK` on success, and `RTTNOSUCHTHREAD` if the specified thread does not exist.

7 External I/O

RT Threads also includes interface routines for many of the UNIX I/O calls. Calls which are blocking will only block the thread making the call and not the entire UNIX process. The parameter lists are identical and the semantics are the same as the corresponding UNIX calls. See the UNIX man pages for details of a particular call. Table 1 shows the RT Threads calls which map directly to UNIX calls.

Table 1: Interface to UNIX I/O System Calls

RTT Routine	UNIX Call
`RttOpen`	`open`
`RttPipe`	`pipe`
`RttSocket`	`socket`
`RttClose`	`close`
`RttBind`	`bind`
`RttListen`	`listen`
`RttAccept`	`accept`
`RttConnect`	`connect`
`RttRead`	`read`
`RttWrite`	`write`
`RttRecvfrom`	`recvfrom`
`RttSendto`	`sendto`
`RttRecvmsg`	`recvmsg`
`RttSendMsg`	`sendmsg`
`RttGetsockopt`	`getsockopt`
`RttSetsockopt`	`setsockopt`
`RttGetsockname`	`getsockname`

Some other routines are provided which do not map directly to UNIX calls, but are provided for convenience. They are described as follows:

(1) *RttNonBlkRead(int fd, char *buf, int numbytes)* reads as much data as is available, up to *numbytes* bytes, without blocking.

(2) *RttNonBlkWrite(int fd, char *buf, int numbytes)* writes as much data as can be written, up to *numbytes* bytes, without blocking.

(3) *RttReadN(int fd, char *buf, int numbytes)* blocks until *numbytes* bytes have been read.

(4) *RttWriteN(int fd, char *buf, int numbytes)* blocks until *numbytes* bytes have been written.

(5) *RttSeekNRead(int fd, char *buf, int numbytes, int seekpos, int seekmode)* seeks to *seekpos* and reads *numbytes* bytes. *Seekmode* is the same as the UNIX *lseek()* parameter *whence*, which takes a value of `SEEK_SET`, `SEEK_CURR` or `SEEK_END` (see the UNIX man pages for *lseek()*).

(6) *RttSeekNWrite(int fd, char *buf, int numbytes, int seekpos, int seekmode)* seeks to *seekpos* and writes *numbytes* bytes. *Seekmode* is as described above for *RttSeekNRead()*.

8 External Libraries

Some external library routines (e.g., X Windows Xlib routines) may not be threads safe. In order to use any routines which may not be threads safe, it is advisable to place the call in a critical section which cannot be interrupted by RT Threads scheduling or I/O. The following routines should be used to bracket any such call so it will not be interrupted:

(1) void RttBeginCritical()

(2) void RttEndCritical()

9 Example

The following example creates two threads which compete to enter a critical section protected with a semaphore. A few points are in order.

First, the header file ‘‘rtthreads.h'' contains the function prototypes, type definitions, etc. necessary to use the RT Threads routines.This file must be included. Then the following library must be linked in to every application that uses RT Threads: **libRtt.a**.

Second, UNIX normally buffers I/O statements such as ‘‘printf'' until an end-of-line character is output. In order to see each printf statement as it is executed, we turn off I/O buffering with the ‘‘setbuf(stdout, 0)" call. **Note:** ideally, the application should implement its own version of “printf” using the I/O routines described in section 7, although experience has shown that the standard C library printf can usually be used without ill effect. However, standard library input routines such as “scanf” will block all threads.

Comments at the beginning of the example program show the recommend command sequence for compiling RT Threads programs.

```
/* app.c
 * to compile:
 *      gcc -g -I($headerfiledir) -c app.c -o app.o
 *      gcc -g app.o ($libdir)/libRtt.a ($libs) -o app
 *      Solaris additionally requires -lsocket and -lnsl libraries.
*/
#include <stdio.h>
#include "rtthreads.h"    /* templates for the functions in RT Threads */

RttThreadId threada, threadb;

RTTTHREAD a(void *arg)
{
     RttSem sem;

     sem = *(RttSem *)arg;
     for(;;)
     {
           printf("1: Wait\n");
           RttP(sem);
           RttGetThreadSchedAttr(RttMyThreadId(),&myattr);
           RttGetTimeOfDay(&now);
           myattr.startingtime.seconds = now.seconds + 0;
           myattr.startingtime.microseconds = now.microseconds + 3000;
           RttSetThreadSchedAttr(RttMyThreadId(),myattr);
           printf("1: Crit\n");
           RttV(sem);
           printf("1: Signal\n");
     }
}

RTTTHREAD b(void *arg)
{
     RttSem sem;

     sem = *(RttSem *)arg;

     for(;;)
     {
           printf("2: Wait\n");
           RttP(sem);
           RttGetThreadSchedAttr(RttMyThreadId(),&myattr);
           RttGetTimeOfDay(&now);
           myattr.startingtime.seconds = now.seconds + 0;
           myattr.startingtime.microseconds = now.microseconds + 3000;
           RttSetThreadSchedAttr(RttMyThreadId(),myattr);
           printf("2: Crit\n");
           RttV(sem);
           printf("2: Signal\n");
     }
}
```

```
mainp()
{
     RttSem *sem;
     RttSchAttr sattrs;

     sem = (RttSem *)RttMalloc(sizeof(RttSem);
     setbuf(stdout,0);/*don't buffer stdout;show printf()'s immediately */
     RttAllocSem(sem, 1, RTTFCFS);
     sattrs.startingtime = RTTZEROTIME;
     sattrs.priority = RTTNORM;
     sattrs.deadline = RTTNODEADLINE;
     threada = RttCreate(a, 16000, "a", (void *)sem, sattrs, RTTUSR);
     threadb = RttCreate(b, 16000,"b", (void *)sem, sattrs, RTTUSR);
     printf("threads created\n");
}

-------------------------- Sample Output: -----------------------------
threads created
1: Wait
1: Crit
1: Signal
2: Wait
2: Crit
2: Signal
2: Wait
1: Wait
1: Crit
1: Signal
1: Wait
1: Crit
```

Appendix A Portability

A.1 Architectures Supported

RT Threads can currently be compiled and executed on the architectures shown in Table 2.

Table 2: Architectures Supported

Architecture	Operating Systems
IBM RS/6000	AIX 3.2.5 and AIX 4.1.1
Sun Microsystems SPARC	SunOS 4.x.x and SunOS 5.2.x (Solaris)
Intel 386	Linux

Most of our experience has been with running RT Threads on AIX and SunOS 3.2.5. The package has not been thoroughly tested on the other systems listed.

An RT Threads interface to the AIX 4.1.1 threads library (**libpthreads.a**) has also been implemented, and will be available in the near future. The AIX threads library uses the standard POSIX threads interface, thus applications using RT Threads will be portable to any system which provides POSIX threads.

Longer term plans include support for HP-UX, OS/2 and Windows NT, as well as possible ports to the Intel 486 and Pentium bare architectures.

A.2 Portability Issues

The operating systems listed in Table 2 are all versions of UNIX. Ports to other versions of UNIX should be straightforward, but must take into account variations in system calls between different versions of UNIX. In particular, the use of *fcntl()* and especially *ioctl()* tends not to be portable.

Other operating systems may provide a service interface which differs from that provided by UNIX. Ports to other operating systems will thus require the substitution of other approriate system calls. The RT Threads system clock and external I/O routines are currently implemented using UNIX signals. Substituting the interrupt handling facilities of other operating systems may require considerable changes in the implementation of these features.

The context switching routines of RT Threads require some architecture specific assembly code. For ports to architectures other than those listed in Table 2, these routines will need to be written in the appropriate assembly language.

Appendix B Networking and RT Threads

The underlying network facilities used by RT Threads for cross address space communication are implemented using the Xpress Tranport Protocol (XTP Revision 4.0). The implementation of XTP used is a multithreaded user level protocol which is portable to a variety of network technologies such as Ethernet and ATM. IP is currently used for addressing. XTP also supports other addressing formats, although minor changes to the RT Threads and XTP implementations may be required to offer such support.

The cross address space send/receive/reply facilities of RTT Threads use reliable XTP data streams to implement a transaction-style service. Message lengths are limited only by end system memory resources.

RT Threads applications have direct access to all the facilities provided by the XTP Application Programming Interface as part of the package. A variety of network data transport services are available from XTP including both reliable and unreliable data streams and datagrams. Additional features include optional and orthogonal use of error control, flow control and rate control.

The use of the XTP interface routines is enabled by a call to *RttNetInit()* in *mainp()* of the application, as described in section 4.2 of this report. For details on using XTP, see the technical report, “XTP Application Programming Interface” [1].

Appendix C Utilities Library

This appendix describes routines which are not part of the RT Threads kernel, but provide useful functions and are included in a utilities library (**libRttUtils.a**).

C.1 Common Scheduling Routines

The scheduling routines described here are provided as a convenience for performing commonly used scheduling tasks, i.e., sleeping threads and suspending/resuming threads. Each of the sleep routines are provided in two forms, one which allows the running thread to put itself to sleep, and one which allows the running thread to put another thread to sleep. In the latter case, if the specified thread was already sleeping, its wake up time will be modified to reflect the new call. The sleep routines are as follows:

(1) *int RttSleep(unsigned int seconds)* puts the calling thread to sleep for the specified number of seconds.

(2) *int RttUSleep(unsigned int microseconds)* puts the calling thread to sleep for the specified number of microseconds.

(3) *int RttSleepFor(RttTimeValue sleepTime)* puts the calling thread to sleep for

the duration specified by an RttTimeValue.

(4) *int RttSleepThread(RttThreadId sleeper, int seconds)* puts the specified thread to sleep for the specified number of seconds.

(5) *int RttUSleepThread(RttThreadId sleeper, int microseconds)* puts the specified thread to sleep for the specified number of microseconds.

(6) *int RttSleepThreadFor(RttThreadId sleeper, RttTimeValue sleepTime)* puts the specified thread to sleep for he duration specified by an RttTimeValue.

Threads can be suspended and resumed using the following routines:

(7) *int RttSuspend(void)* suspends the calling thread.

(8) *int RttResume(RttThreadId thread)* resumes the specified thread.

Note: Sleeping threads can be awoken using *RttResume()* or *RttSetThreadSchedAttr()*, and suspended threads can be resumed using any of the applicable sleep routines or *RttSetThreadSchedAttr()*. This is because the routines are merely convenient wrappers around *RttSetThreadSchedAttr()*.

C.2 Mutex Routines.

The mutex routines provide a convenient way to protect critical sections of code from shared access among threads. Please note that these mutexes **do not** prevent context switches from occurring within a critical section.

The following header file must be included: `RttMutex.h`.

The calls are as follows:

(1) *int RttNewMutex(u_long *mutexvar)*. This call returns an integer for the critical section.

(2) *int RttReapMutex(u_long mutexvar)*. This call frees resources for a mutex.

(3) *int RttMutexLock(u_long mutexvar)* This call ensures exclusive access.

(4) *int RttMutexUnlock(u_long mutexvar)* This call releases the exclusive access claim.

All of the routines return `RTTOK` on success and `RTTFAILED` on failure.

C.3 Queuing Routines.

The utilities library provides the queueing functions *RttNewQueue, RttReapQueue, RttDequeue,* and *RttEnqueue.* Multiple threads can enqueue and dequeue items using the same queue.

The following header file must be included: `RttQueue.h.`

It defines the following status values: `RTTQOK`, `RTTQFAILED`, `RTTQFULL`, `RTTQEMPTY`.

A queue can be created in one of several modes defined as follows:

`RTTQDEFAULT`: Default style queue, simple add/remove. Routines return `RTTQUEUEOK.`

`RTTQREPORT`: Returns a status (see above) to *RttEnqueue()* so the producer can detect the status of the queue. If you don't care if your producer is faster than your consumer (or vice versa) you don't need `RTTQREPORT`.

`RTTQNBDEQ`: *RttDequeue* will not block if the queue is empty. Instead, it returns `RTTQEMPTY` immediately.

`RTTQNBENQ`: *RttEnqueue* will not block if the queue is full. Instead, it returns `RTTQFULL` immediately.

`RTTQOVERWRITEENQ`: With this mode, a call to *RttEnqueue()* with a full queue will overwrite the last entry in the queue with the item passed in to *RttEnqueue()* and then return `RTTQFULL`. `RTTQNBENQ` simply returns `RTTQFULL` and does not put the item in the queue.

Mode is set by doing a bitwise-or of the modes desired, as in:

```
RttNewQueue(&queue, queueSize, RTTQNBDEQ | RTTQNBENQ);
```

The calls are as follows:

(1) *int RttNewQueue(RttQueue *queue, int size, int mode)*; This creates a new queue and returns a pointer to that structure by reference.

(2) *int RttReapQueue(RttQueue queue, void(*cleaner)());* This destroys the queue structure. The routine *cleaner* is a user defined cleanup routine which takes a *void** as a parameter and is called for each remaining item in the queue.

(3) *int RttDequeue(RttQueue queue, void **item);* Take an item off of the queue,

with appropriate reporting if the mode is `RTTQREPORT`.

(4) *int RttEnqueue(RttQueue queue, void *item, int *underflowcount);* Place an item on the queue. If the queue mode is `RTTQREPORT` and the return status is `RTTQEMPTY,` *underflowcount* will be set to the number of times the consumer requested an item from the queue and found it to be empty.

C.4 Barriers.

The utilities library provides an additional synchronization facility with the barrier routines *RttNewBarrier, RttWaitOnBarrier*, *RttGrowBarrier*, *RttShrinkBarrier*, and *RttReapBarrier.*

An application using barrier facilities must have the following header file: `RttBarrier.h`.

The calls are as follows:

(1) *int RttNewBarrier(RttBarrier *barrier, int size);* This call returns a barrier identifier by reference parameter *barrier. Size* indicates the number of processes participating in the barrier, and must be non-negative.

(2) *int RttWaitOnBarrier(RttBarrier barrier, int status);* No thread can continue past this call until all threads participating in the barrier have reached their barrier point. This call returns when that situation exists. The *status* parameter allows each caller to pass in a status of either `RTTOK` or `RTTFAILED`. If all callers specify `RTTOK`, then *RttWaitOnBarrier()* will return `RTTOK`, otherwise it will return `RTTFAILED`.

(3) *int RttGrowBarrier(RttBarrier barrier);* This call increases the size of the barrier (i.e., the number of threads which participate in the barrier) by one.

(4) *int RttShrinkBarrier(RttBarrier barrier);* This call shrinks the size of the barrier (i.e., the number of threads which participate in the barrier) by one. A barrier can never have a size less than zero.

(5) *int RttReapBarrier(RttBarrier barrier);* This routine cleans up the data structures used and the threads associated with the barrier.

All of the routines return `RTTOK` on success and `RTTFAILED` on failure, or `RTTNOSUCHTHREAD` if the barrier does not exist..

Appendix D Obtaining RT Threads

The RT Threads package is available via anonymous ftp to `ftp.cs.ubc.ca`, where it can be found in the directory `pub/local/dsg/rtt` as the file `rtt.tar.gz`.

This distribution includes support for the architectures listed in Appendix A, Table 2, as well as the XTP facilities described in Appendix B. Also included in the distribution are `README` files with instructions on compilation, as well as the technical report "XTP Application Programming Interface" [1].

Acknowledgements

We thank Murray W. Goldberg whose Pthreads preemptive threads package formed the basis for RT Threads, and Ann Lo whose work on real time scheduling was incorporated into the package.

References

[1] Roland Mechler and Gerald W. Neufeld. XTP Application Programming Interface, UBC Technical Report 95-17, May 1995.

[2] Siu Ling Ann Lo, Norman C. Hutchinson, Samuel T. Chanson. Architectural Considerations in the design of Real-Time Kernels. In *IEEE Proceedings of the Real-Time Systems Symposium*, pp.138-147, December 1993.