IBM® Rational® SDL Suite

Threaded OS Integrations

Copyright Notice

Copyright license

Trademarks

How to contact Customer Support

Contacting IBM Rational Software Support

If the self-help resources have not provided a resolution to your problem, you can contact IBM® Rational® Software Support for assistance in resolving product issues.

Note:

If you are a heritage Telelogic customer, you can go to http://support.telelogic.com/toolbar and download the IBM Rational Telelogic Software Support browser toolbar. This toolbar helps simplify the transition to the IBM Rational Telelogic product online resources. Also, a single reference site for all IBM Rational Telelogic support resources is located at http://www.ibm.com/software/rational/support/telelogic/

Prerequisites

To submit your problem to IBM Rational Software Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage from http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html

- To learn more about Passport Advantage, visit the Passport Advantage FAQs at http://www.ibm.com/software/lotus/passportadvantage/brochures_faqs_quickguides.html.
- For further assistance, contact your IBM representative.

To submit your problem online (from the IBM Web site) to IBM Rational Software Support, you must additionally:

- Be a registered user on the IBM Rational Software Support Web site. For details about registering, go to http://www-01.ibm.com/software/support/.
- Be listed as an authorized caller in the service request tool.

Submitting problems

To submit your problem to IBM Rational Software Support:

1. Determine the business impact of your problem. When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.

   Use the following table to determine the severity level.

Severity	**Description**
1	The problem has a ***critical*** business impact: You are unable to use the program, resulting in a critical impact on operations. This condition requires an immediate solution.
2	This problem has a ***significant*** business impact: The program is usable, but it is severely limited.
3	The problem has ***some*** business impact: The program is usable, but less significant features (not critical to operations) are unavailable.
4	The problem has ***minimal*** business impact: The problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

2. Describe your problem and gather background information, When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:

- What software versions were you running when the problem occurred?

To determine the exact product name and version, use the option applicable to you:

- Start the IBM Installation Manager and select **File > View Installed Packages**. Expand a package group and select a package to see the package name and version number.
- Start your product, and click **Help > About** to see the offering name and version number.

- What is your operating system and version number (including any service packs or patches)?
- Do you have logs, traces, and messages that are related to the problem symptoms?
- Can you recreate the problem? If so, what steps do you perform to recreate the problem?
- Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?
- Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.

3. Submit your problem to IBM Rational Software Support. You can submit your problem to IBM Rational Software Support in the following ways:

   - **Online**: Go to the IBM Rational Software Support Web site at https://www.ibm.com/software/rational/support/ and in the Rational support task navigator, click Open Service Request. Select the electronic problem reporting tool, and open a Problem Management Record (PMR), describing the problem accurately in your own words.

     For more information about opening a service request, go to http://www.ibm.com/software/support/help.html

     You can also open an online service request using the IBM Support Assistant. For more information, go to http://www-01.ibm.com/software/support/isa/faq.html.

   - **By phone**: For the phone number to call in your country or region, go to the IBM directory of worldwide contacts at http://www.ibm.com/planetwide/ and click the name of your country or geographic region.

– **Through your IBM Representative**: If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at http://www.ibm.com/planetwide/.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Rational Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Rational Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Rational Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.

Table of Contents

Threaded OS Integrations

Threaded OS Integrations

Overview

The threaded integrations with Real-time operating systems described in this document has been developed by IBM Rational in the IBM Rational Tau product. Our intension, however, is that the integrations should be possible to use both in IBM Rational SDL Suite (Cadvanced) and IBM Rational Tau (AgileC and C Code Generator) products.

The difference between the threaded integrations described in this document and the previous existing threaded integrations before SDL Suite version 4.5 is not that large. The integration principles are almost exactly the same. The major difference is that previous integrations are expressed using macros, while this integration uses a functional interface. Another difference is that all the previous integrations provided by IBM Rational are mixed into one .h file, while in this integration, each RTOS integration is implemented using one .h and one .c file. Both these changes are made to increase the readability and to simplify debugging. The change in file structure makes it also easier to implement new integrations and for a customer to modify an integration.

The integrations currently available integrations are listed below.

Previous integrations:

- SUN Solaris (This is a POSIX pthread integration that probably can be used on most UNIX-like operation system.)
- Win32
- VxWorks
- OSE

The new threaded integrations:

- POSIX pthreads (tested on SUN Solaris and Linux but can probably be used on most UNIX-like operation system.)
- Win32
- VxWorks
- Nucleus Plus

Each RTOS integration consists of two files with the names rtapidef.h and rtapidef.c. The rtapidef.h file is included in the scttypes.h file, while the rtapidef.c file is included in the sctsdl.c file. As rtapidef.c is included in sctsdl.c there are no new files to compile and the make files are not effected, except for some compilation options discussed below.

When it comes to compilation none of the compilation switches used for the previous threaded integrations should be defined. To use the new integrations the following should be given:

- a switch selecting an application kernel, for example SCTAPPLCLENV
- the switch THREADED
- a compiler option to tell the compiler where to find the rtapidef.h and the rtapidef.c file.

Example: `cc -DSCTAPPLCLENV -DTHREADED -I/some/suitable/path`

In the remaining part of this document the new threaded integrations are described in detail. This documentation is provided for customers who want to understand and possibly modify the integrations or to make new integrations themselves. The documentation was initially written for the AgileC code generator in IBM Rational Tau and has been somewhat adopted to Cadvanced in IBM Rational SDL Suite.

Note: The new threaded integrations are not prepared for ALTERNATIVE_SIGNAL_SENDING as the old threaded integrations were. This limits the new threaded NucleusPlus integration interrupt routines usage. The way Nucleus Plus handle semaphores in conjunction with interrupts makes it impossible to send messages from an interrupt routine directly to an SDL process.

Threaded integrations

To implement a new integration or to understand an existing one it is recommended to use this manual together with the code for some existing integration(s). There are some major aspects that have to be handled to implement an integration with real-time operating system.

- It is necessary to implement a clock function.
- There is need for a number of mutexes or binary semaphores to protect some shared data.

- Some startup code, for creating threads with relevant properties and synchronizing them are needed.
- A thread must be able to suspend its execution when it has nothing to do. It must then be possible to wake it up again when a signal is sent to a part in the thread.

To explain the details in these integration aspects the POSIX integration will be used as an example. Apart from the code mentioned below the rtapidef.h should include the necessary system include files to be able to access the concepts needed.

Example 1: Includes in rtapidef.h for POSIX

```
#include <pthread.h>
#include <sched.h>
#include <semaphore.h>
#include <time.h>
#include <sys/time.h>
```

If the RTOS has any requirements on the main function, which might be the case, it is possible to rename the main function included in uml_kern.c by defining XMAIN_NAME to for example:

```
#define XMAIN_NAME agilec_main
```

Then the user has to implement a proper main function that calls the `agilec_main` function.

The clock function

To support the SDL concept of timers, a clock function is necessary. The generated code and the kernel assumes that there is a clock function called `xNow` that returns the current time. Time values are represented by values of type `SDL_Time` which is defined as:

```
typedef struct
  s, ns xint32;
} SDL_Time;
```

`xint32` is implemented as a 32-bit int. The components `s` and `ns` represent the number of seconds and nanoseconds passed from some time in the past depending on the implementation of the clock function.

There are two standard implementations of the clock function, one for UNIX like systems and one for Windows. In Windows the standard function `_ftime` is used to read the system clock, while on UNIX like systems the standard function `clock_gettime` is used.

To implement a clock function you should include code in your own `rtapidef.h` and `rtapidef.c` files according to the details below.

If timers are not used and the clock is not explicitly accessed in SDL or C, there is no need for a clock implementation. Just include the macro definition:

```
#define xInitSystime()
```

in `rtapidef.h`.

If a clock implementation is needed then include the following prototypes in `rtapidef.h`:

```
extern void xInitSystime(void);
extern SDL_Time xNow (void);
```

If no initialization function is needed then the `xInitSystime` function can be replaced by the macro.

```
#define xInitSystime()
```

In the file `rtapidef.c` the implementation of these functions should be provided. The implementations will depend a lot on the support in software and hardware for the underlying architecture.

Protection of shared data

It is necessary to protect the list of available signals, the list of available timers, and a few other things. For this four global mutexes or binary semaphores are needed. These variables should be defined `extern` in `rtapidef.h` and declared in `rtapidef.c`. The names of the variables should be the same as in the example given below.

Example 2: In rtapidef.h: ––

```
extern pthread_mutex_t xFreeSignalMutex;
extern pthread_mutex_t xFreeTimerMutex;
extern pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  extern pthread_mutex_t xMemoryMutex;
#endif
```

––

Example 3: In rtapidef.c:

```
pthread_mutex_t xFreeSignalMutex;
pthread_mutex_t xFreeTimerMutex;
pthread_mutex_t xCreateMutex;
#ifdef USER_CFG_USE_MEMORY_PACK
  pthread_mutex_t xMemoryMutex;
#endif
```

These four variables should be initialized during the startup of the application to an unlocked state. The function `xThreadInit` is a proper place for this initialization. Note that the names of the variables are chosen for AgileC in IBM Rational Tau, and do not fully reflect their usage in Cadvanced in IBM Rational SDL Suite.

Example 4: xThreadInit

```
void xThreadInit (void)
{
  (void)pthread_mutex_init(&xFreeSignalMutex, 0);
  (void)pthread_mutex_init(&xFreeTimerMutex, 0);
  (void)pthread_mutex_init(&xCreateMutex, 0);
  #ifdef USER_CFG_USE_MEMORY_PACK
    (void)pthread_mutex_init(&xMemoryMutex, 0);
  #endif
  ....
}
```

The lock and unlock operation must also be implemented for mutexes or binary semaphores. The following two functions should be implemented.

Example: In `rtapidef.h`:

```
extern void xThreadLock (pthread_mutex_t *);
extern void xThreadUnlock (pthread_mutex_t *);
```

In `rtapidef.c`:

```
void xThreadLock (pthread_mutex_t *M)
{
  (void)pthread_mutex_lock(M);
}

void xThreadUnlock (pthread_mutex_t *M)
{
  (void)pthread_mutex_unlock(M);
}
```

Startup phase - creating the threads

After some basic initialization the kernel will in the main function start the specified threads. For each thread the functions `xThreadInitOneThread` and `xThreadStartThread` will be called, where `xThreadInitOneThread` should perform some thread specific initialization and `xThreadStartThread` should start the thread. Each thread should run the function `xMainLoop` declared in the kernel. This is performed by using a wrapper function, `xThreadEntryFunc`, which is defined in the integration and is the function really started in the thread.

After all the threads have been started the function `xThreadGo` is called in the function `main`. Some more information on these functions are given below.

It is important that the started threads do not execute any SDL transitions before all threads are created. Therefore the `xThreadEntryFunc` will as first action wait on a semaphore. The `xThreadGo` function will when all threads are created release this semaphore.

Many functions has a pointer to type `xSystemData` as parameter. This contains the local information for the thread. Among other things it contains a field of type `xThreadVars`, which should be defined in the RTOS integration.

Example: `xThreadVars` type in `rtapidef.h`

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
  pthread_t       ThreadId;
} xThreadVars;
```

where the two first fields will be discussed in the next section, and the ThreadId will be used during the startup phase to store the identity of the threads.

The code for the behavior described in this section should look something like the following example.

Example: In `rtapidef.h`:

```
extern sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  extern sem_t xMainThreadSem;
#endif

extern void xThreadInitOneThread (
   struct _xSystemData *);
extern void xThreadStartThread (
   struct _xSystemData *,
   unsigned int, unsigned int,
   unsigned int, unsigned int);
```

In `rtapidef.c`:

```
sem_t xInitSem;
#if !defined(USER_CFG_USE_xInEnv) && !defined(XENV)
  sem_t xMainThreadSem;
#endif

void xThreadInit (void)
{
  ....
  (void)sem_init(&xInitSem, 0, 0);
}

void xThreadInitOneThread(struct _xSystemData *xSysDP)
{
  (void)pthread_mutex_init(
    &xSysDP->ThreadVars.SignalQueueMutex, 0);
  (void)pthread_cond_init(
    &xSysDP->ThreadVars.SignalQueueCond, 0);
}


static void *xThreadEntryFunc (void *xSysDP)
{
  (void)sem_wait(&xInitSem);
  (void)sem_post(&xInitSem);
  xMainLoop((xSystemData *)xSysDP);
}


void xThreadStartThread(struct _xSystemData *xSysDP,
                        unsigned int StackSize,
                        unsigned int Prio,
                        unsigned int User1,
                        unsigned int User2)
{
  pthread_attr_t Attributes;
  ....
  (void)pthread_create(&xSysDP->ThreadVars.ThreadId,
                       &Attributes, xThreadEntryFunc,
                       (void *)xSysDP);
  ....
}

void xThreadGo(void)
{
  (void)sem_post(&xInitSem);
```

```
      #if defined(USER_CFG_USE_xInEnv)
        xInEnv();  /* AgileC */
      #elif defined(XENV)
        xInEnv(xNow());  /* Cadvanced */
      #else
        (void)sem_init(&xMainThreadSem, 0, 0);
        (void)sem_wait(&xMainThreadSem);
      #endif
    }
```

The `xInitSem` semaphore is used for synchronization of the threads. It is initialized in the beginning of `xThreadInit` to 0, that is to a blocking state. After that the `xThreadStartThread` once for each thread that is to be started. The function `pthread_create` will call the function given as third parameter (`xThreadEntryFunc`) with the void * parameter given as fourth parameter (the `xSysD` pointer) as parameter. `pthread_create` will also store the identity of the thread in the variable passed as first parameter. The second parameter is the properties of the thread. This will be discussed later in this section.

If any of the threads get a chance to execute before all the threads are created, these threads will hang on the `sem_wait` call in `xThreadEntryFunc`, until the main thread calls `xThreadGo` that will post the semaphore `xInitSem` once. One of the threads waiting on this semaphore will then be able to execute and will immediately post the semaphore again. This will continue until all threads are free to execute.

After that all threads are running and depending on the OS and the application properties, the main thread can perform different things. Our recommendation is to call the `xInEnv` function and let that function run in this thread. For more details see the discussion on `xInEnv`. Another alternative is to hang the main thread on a semaphore, as shown above using the semaphore `xMainThreadSem` (if `xInEnv` is not used). In this case you can post the `xMainThreadSem` semaphore anywhere to restart the execution of the main thread.

When the main thread returns from the function `xThreadStart`, the program will continue to execute in the main function and will perform a call to exit. The behavior of a threaded program when the main thread performs exit, is OS dependent. In POSIX pthreads all threads are stopped at such an action. That is the reason it is important to hang the main thread at the end of the `xThreadStart` function.

Now to the properties of the threads. In most RTOS, properties like stack size and priority can be set for individual threads. Together with the definition of the thread artifacts, four integer values can be specified.

- The first value is interpreted as the stack size.
- The second value is interpreted as the priority.
- The third and fourth values can be used for other properties, defined by the RTOS integration or defined by you.

The currently predefined integrations only makes use of the first and second values. These values are passed as parameters to the `xTreadStartThread` function.

How the properties are set up in detail depend on the RTOS. Please see the available integrations, in the function `xThreadStartThread`, for examples.

In `rtapidef.h` proper default values for the four `xThreadData` fields should be set up. These default values are used if no value is specified in the thread definition.

Example:

```
#define DEFAULT_STACKSIZE     1024
#define DEFAULT_PRIO             0
#define DEFAULT_USER1            0
#define DEFAULT_USER2            0
```

Suspending and waking up threads

When a thread finds out that it has nothing more to do, at least just for the moment, it should suspend itself to make the processor available for other threads. The thread should then wake up again either when a timer has expired and needs to be handled, or when some other thread (including `xInEnv`) sends a signal that should be treated by the suspended thread.

To implement these features one mutex or binary semaphore is used together with some sort of conditional variable. We need the possibility to perform a condition wait, with or without a timeout. We need also a way to signal to a thread to wake up again. These two entities are needed for each thread and is therefore included in the `xThreadVars` struct mentioned earlier:

```
typedef struct {
  pthread_mutex_t SignalQueueMutex;
  pthread_cond_t  SignalQueueCond;
```

```
     pthread_t        ThreadId;
   } xThreadVars;
```

The purpose of the `SignalQueueMutex` is to protect the signal queue where signals from the outside of the thread are inserted. The `SignalQueueCond` should facilitate the conditional wait.

The `SignalQueueMutex` should be initialized in `xThreadInitOneThread`. If `SignalQueueCond` needs to be initialized it could be performed at the same place.

Example 5 ––

```
void xThreadInitOneThread(struct _xSystemData *xSysDP)
{
 (void)pthread_mutex_init(&xSysDP->ThreadVars.SignalQueueMutex, 0);
 (void)pthread_cond_init(&xSysDP->ThreadVars.SignalQueueCond, 0);
}
```

––

The `SignalQueueMutex` is locked by using the function `xThreadLock`, discussed above. It is then unlocked in three different ways:

- `xThreadUnlock` (discussed above)
- `xThreadWaitUnlock`
- `xThreadSignalUnlock`

The `xThreadWaitUnlock` is called by the thread itself when it has come to the conclusion that it should suspend itself, while `xThreadSignalUnlock` is called by another thread that wants to wake up the current thread. Both functions are passed the `xSysD` pointer for the thread that the operation should be performed on.

Example: In `rtapidef.h`:

```
extern void xThreadWaitUnlock (struct _xSystemData *);
extern void xThreadSignalUnlock (struct _xSystemData *);
```

Example: In `rtapidef.c`:

```
void xThreadWaitUnlock (struct _xSystemData *xSysDP)
{
  #if defined(CFG_USED_TIMER) || defined(THREADED)
  #ifdef THREADED
      /* Cadvanced */
    if (xSysDP->xTimerQueue->Suc==xSysDP->xTimerQueue) {
  #else
      /* AgileC */
    if (! xSysDP->TimerQueue) {
```

```
    #endif
        (void)pthread_cond_wait(
          &xSysDP->ThreadVars.SignalQueueCond,
          &xSysDP->ThreadVars.SignalQueueMutex);
      } else {
        struct timespec timeout;
        #ifdef THREADED
            /* Cadvanced */
          timeout.tv_sec =
            ((xTimerNode)xSysDP->xTimerQueue->Suc)->
              TimerTime.s;
          timeout.tv_nsec =
            ((xTimerNode)xSysDP->xTimerQueue->Suc)->
              TimerTime.ns;
        #else
            /* AgileC */
          timeout.tv_sec = xSysDP->TimerQueue->Time.s;
          timeout.tv_nsec = xSysDP->TimerQueue->Time.ns;
        #endif
        (void)pthread_cond_timedwait(
          &xSysDP->ThreadVars.SignalQueueCond,
          &xSysDP->ThreadVars.SignalQueueMutex,
          &timeout);

      }
    #else
      (void)pthread_cond_wait(
        &xSysDP->ThreadVars.SignalQueueCond,
        &xSysDP->ThreadVars.SignalQueueMutex);
    #endif
    (void)pthread_mutex_unlock(
      &xSysDP->ThreadVars.SignalQueueMutex);
  }

  void xThreadSignalUnlock (struct _xSystemData *xSysDP)
  {
    (void)pthread_cond_signal(
      &xSysDP->ThreadVars.SignalQueueCond);
    (void)pthread_mutex_unlock(
      &xSysDP->ThreadVars.SignalQueueMutex);
  }
```

At the time when `xThreadWaitUnlock` or `xThreadSignalUnlock` is called the `SignalQueueMutex` will be locked and must therefore be unlocked at the end of both functions.

In `xThreadWaitUnlock` the thread wants to suspend itself. If timers are used and there is a timer active in the timer queue, it should wait until the timer expires or until some other thread tells it to wake up. In POSIX pthreads the function `pthread_cond_wait` performs exactly this. If timers are not used or there is no timer active, the thread should be suspended until someone else wakes it up. In POSIX pthreads this can be achieved with the function `pthread_cond_wait`.

In `xThreadSignalUnlock` the thread given by the parameter should be waken up. Here the `pthread` function `pthread_cond_signal` can be used.

The integrations described here are also used when the Cadvanced is used to generate threaded applications. This adds a few requirements in the implementation of a threaded integration. First a function that can stop a thread is needed.

In rtapidef.h:

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
extern void xThreadStopThread(struct _xSystemData *);
#endif
```

In rtapidef.c:

```
#if defined(THREADED) || defined(CFG_USED_DYNAMIC_THREADS)
void xThreadStopThread(struct _xSystemData *xSysDP)
{
  pthread_mutex_destroy(&xSysDP->ThreadVars.SignalQueueMutex);
  pthread_cond_destroy(&xSysDP->ThreadVars.SignalQueueCond);
  pthread_exit(0);
}
#endif
```

The `xThreadStopThread` function should clean up the thread specific semaphores and stop the thread. It is always the thread that should be stopped that will call this function to stop itself.

Another difference is the way timers are accessed for the two code generators. This effects the details in the `xThreadWaitUnlock` function. Please see this function above and especially the sections under #ifdef THREADED.

In the case of the Cadvanced the RTOS integrations are accessed through a macro layer. The macros in this layer is used in the Cadvanced kernel files and in the generated code.

Example 6: Defines in scttypes.h –––––––––––––––––––––––––––––––––

The following defines are relevant (from scttypes.h):

```
#define THREADED_GLOBAL_VARS
#define THREADED_GLOBAL_INIT \
     xThreadInit();
#define THREADED_THREAD_VARS \
     xThreadVars ThreadVars;
#define THREADED_THREAD_INIT(SYSD) \
     xThreadInitOneThread(SYSD);
```

```
#define THREADED_THREAD_BEGINNING(SYSD)
#define THREADED_AFTER_THREAD_START \
     xThreadGo();
#define THREADED_START_THREAD(F, SYSD, STACKSIZE, PRIO, USER1,
USER2) \
xThreadStartThread(SYSD, STACKSIZE, PRIO, USER1, USER2);
#define THREADED_STOP_THREAD(SYSD) \
     xThreadStopThread(SYSD);
#define THREADED_LOCK_INPUTPORT(SYSD) \
     xThreadLock(&SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_UNLOCK_INPUTPORT(SYSD) \
     xThreadUnlock(&SYSD->ThreadVars.SignalQueueMutex);
#define THREADED_WAIT_AND_UNLOCK_INPUTPORT(SYSD) \
     xThreadWaitUnlock(SYSD);
#define THREADED_SIGNAL_AND_UNLOCK_INPUTPORT(SYSD) \
     xThreadSignalUnlock(SYSD);
#define THREADED_LISTREAD_START  xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTWRITE_START xThreadLock(&xFreeSignalMutex);
#define THREADED_LISTACCESS_END  xThreadUnlock(&xFreeSignalMutex);
#define THREADED_EXPORT_START    xThreadLock(&xCreateMutex);
#define THREADED_EXPORT_END      xThreadUnlock(&xCreateMutex);
```