Rational Statemate

Porting Embedded Rapid Prototyper Run-Time Libraries

White Paper

Before using the information in this manual, be sure to read the “Notices” section of the Help or the PDF file available from **Help > List of Books**.

This edition applies to IBM® Rational® Statemate ®4.6 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Porting Embedded Rapid Prototyper Run-Time Libraries

Embedded Rapid Prototyper (ERP) is shipped with runtime support for the following operating systems:

- Solaris
- HPUX
- Windows
- VxWorks

This white paper describes how to integrate ERP run-time libraries into new embedded environments. Topics include:

- **Structure of the Code** (includes the interface between the code and the environment)
- **Target Dependent Areas in the RTL**

This white paper is intended for users who wish to port the code or simply to gain a better understanding of how the code works on the Rational Statemate platform itself. You should be familiar with Rational Statemate terminology and concepts, as well as the terminology of the code generator tool, such as compilation profiles, scope, etc.

Structure of the Code

The following is a list of the C real time libraries:

- Intrinsics
- Debugger
- Real time timer/ Scheduler
- Simulated time timer/ Scheduler
- Graphical back animation client
- Remote panel/trace client
- Server communication utilities

Every library contains OS-dependent parts, which are basically grouped into several files. In general, the mechanisms that depend on the OS are:

- Tasks and task synchronization implementation
- Timer implementation
- TCP/IP connection
- Main routine implementation
- CTRL-C handling.

The code generated by ERP uses these services from “adapters” that are linked with the code. In order to use the generated code on other Real Time Operating Systems (RTOS), it is necessary to modify these adapters for the new RTOS.

The Architecture of the Generated Code

The ERP generates fully functional code, based on the state charts and activity charts in the Rational Statemate model. The generated modules are partitioned according to a user-specified compilation profile. This allows code to be generated from a sub-section of the complete Rational Statemate model.

Each generated module reflects the state, timing, and scheduling logic of the model that is included in the compilation profile. This allows a suitable set of components to be built that reflect the system logic (behavior). All the logic is written in K&R or ANSI C, with calls to services from an underlying runtime module.

The generated code uses runtime modules for timing, scheduling and communication with analysis tools like GBA, Panels, Trace and user-defined drivers (I/O mapping). Requests are generated to the timing module for time-outs and scheduled actions, to the scheduler module to control user written tasks that are connected to basic activities or auxiliary tasks that serve to supply interface with I/O card drivers, GBA, Trace and Panel servers. In addition, the data elements can be double buffered, so data assignments are synchronized, to prevent racing conditions among the ‘concurrent’ behavioral components. Note that the envelopes of the user written tasks perform direct assignment, so the assignments inside of these tasks are not synchronized.

All the runtime modules are actually a set of compiled libraries. The libraries are supplied in source code form so they can be modified and reused in other environments. The runtime modules actually provide an interface between the generated behavioral logic and the underlying operating system.

Porting the generated code to a particular Operating System (OS) environment involves tailoring the runtime libraries to use the specific services provided by the target OS/Real-time kernel. In case where no underlying system exists, the run time library should provide an alternative functionality.

Note that tailoring the runtime libraries is a one-time effort. Once done, the generated components can be compiled and linked without being modified at all.

Tasks View of the Code

Concurrency within the languages of Rational Statemate is represented explicitly between orthogonal states (AND states), and implicitly between separate (concurrent) activities. Sometimes it is natural to implement them as different threads (tasks), but it is also possible to implement them as a single threaded program.

All the modules of the generated code are executed sequentially, as a single thread. We refer to this thread as the “main task." The code is executed cyclically, with each iteration evaluating the next step of processing. In terms of simulation, executing the code is equivalent to executing a “go-step” repeatedly, while changing the environment asynchronously. The main difference is that the clock is incremented in real time, so time-outs will happen according to the time taken to execute the code.

Note, if the simulated asynchronous time model was selected in the compilation profile, there is no difference between the generated code and the simulation, except in some specific situations (like racing).

Multi-threading is used to implement basic activities as independent processes without complying with the one-cycle-at-a-time method.

It also allows additional environment processes to be written outside the system model. These processes typically will read inputs, drive outputs or simulate the environment. Therefore, a multi-threading capability is needed only if you wish to add threads that run “concurrently” with the generated code.

Another component in the task view of the code is the asynchronous timer. The main task issues timer requests to be notified about time-outs and scheduled actions. The timer module asynchronously notifies the main task when time-out events occur.

In addition to the above there are tasks that support the communication between the generated code and analysis tools. These are graphical back animation client task, remote panel client task, trace task, input card task.

These tasks can be created and started only if corresponding elements of compilation profile are defined. See Rational Statemate documentation for more details.

Main Task – Partition and Flow of Control

This section describes how the different generated modules are put together into a single thread, and what is the control flow of the main task. The whole execution starts with an initialization phase, where all components are initialized: the timer, the threads scheduler (if needed) and other tasks are created. In addition the user_init procedure is called and the debugger tables are initialized (if needed).

After initialization phase, the main-task starts processing in a cyclic manner, where every cycle corresponds to a single “go-step." In every cycle, all the concurrent state-machines are traversed, process their inputs and generate outputs, issue timing requests and take the necessary state transitions.

As mentioned before, the main program is actually the body of the main task that activates all the state machines. The pr_initialize is the initialization procedure, and pr_make_step completes a single-step of the whole system, including tasks execution.

Double Buffering

Rational Statemate semantics assumes that the step execution is based on the set of values at the beginning of the step. It means all assignments are performed using special update list, where the updated values are saved until the update function will be called. The update function executes all the deferred assignments into the actual data objects, based on the update list. As a by-product, the function can determine whether the system is still processing data or it has reached a stationary condition. If the update list is empty, it means that the behavioral module executed an idle step. The final decision on the system’s stability is based on the result of the update function execution, task execution and combinational assignment execution (if exist in model).

Note that the time-out events are buffered into another update list, which is processed by the update function too. Such separation allows not to block timer interrupts while the step in the behavioral module and tasks is executed. Moreover, the tasks perform direct assignments of their output parameters not using update list, because the step in the behavioral module finishes before task execution and there is no possibility of racing between model and task assignments.

Evaluating the Callback List

If you set callbacks for some variables or define some output to the panel, they will be checked at this point. If the variable bound to the callback or panel is changed, the callback procedure is called. Note that if no callbacks are defined and no output/inout panel binding exist, the callback call is not generated.

Entering the WAIT State

If the system executed the idle step, it is in a stationary condition. Note that the negation of events might yield an active trigger after the idle step. If such negation events are found in the model, a warning message will be issued in the .info file.

At this point, the main task will release the CPU by calling to a system service that will block it from running, until some external stimulus occurs. The external stimulus can be either an event/data change, or a time-out.

In cases where the wait primitive is “blind,” i.e., it is not based on a condition such as a semaphore or event flag, the test whether to enter a wait state or not should be handled carefully, since once the main task blocks itself only external input will wake it. This is the case in Unix, where the *pause* primitive is “blind” in the sense that it will block the whole process unconditionally. Therefore, the *sched_pause* procedure, that actually blocks the task, must test and block mutually exclusive to other asynchronous tasks (such as the timer ISR), to prevent a deadlock.

If the wait call is based on an event flag or a semaphore, then the above scenario will not lead to a deadlock, since the wakeup call will release the waiting flag (semaphore) and the wait primitive will simply flow through. This is usually the case in RTOS.

Target Dependent Areas in the RTL

Scheduler

The task mechanism serves to create, start, suspend, resume and stop the tasks. These tasks may be either user-defined tasks or service tasks intended to support GBA, Panel, Trace and/or I/O mapping features of generated code. The tasks implementation and synchronization between tasks is differing for user-defined tasks and other tasks. The tasks are implemented using native mechanism of OS. So every place, where these OS-dependent functions/data structures are used, should be changed/revised according to target OS task description. The task synchronization is performed using semaphore mechanism. Of course, if the target OS does not contain any semaphore implementation, an alternative feature should be used for that.

The scheduler library provides the multithreading package, which contains the task manipulation scheduling services. The task control block (task_entry) has different structure for every OS, so it should be changed for the porting purpose. Moreover, the task services should be implemented in different way for target OS. These parts of library should be rewritten.

- Task delay
- Timeout/scheduled action blocking/unblocking functions
- Waiting for event
- Semaphore mechanism

Timer

The timer implementation is the most complicated part of the run-time libraries. Currently there are three files that are OS dependent:

- Low_timer.c
- Vxtimer.c
- Dostimer.c

The timer implementation is based on the task mechanism. Timer task waits for timer requests. Once a request was received, it becomes pending request, that within a given-time out will expire unless overwritten by a new request. The major point is that while the task waits for expiration of the request, it has to listen to new request. The assumption is that the asynchronous timer should handle only a single pending request, since the support for multiple requests is implemented in the generic level of the timer package.

The timer uses the underlying OS services that in general supply current time and system timer access. So these parts of the library should be replaced with and interface to the targeted OS services.

In addition, the timer supplies the means of synchronization. The synchronization services are not directly related to time. They are included because they can block/unblock the asynchronous timer calls. The major role of the synchronization services is to synchronize the main task with the environment and other tasks.

Note that the UNIX- oriented part of the timer uses the system signal SIGALARM in order to receive the timeout event. The Windows/VxWorks –oriented timer is built as a task that executes every time interval and evaluates timeout event itself.

TCP/IP Connection

The interface between the generated code (client) and several analysis tools (server) is implemented using TCP/IP protocol. There are three tasks, which are using this connection: GBA, Remote panel and trace tasks. The socket implementation differs for every target OS, so you need to change the functions, implementing the TCP/IP connection. In addition, semaphores are used for mutual-exclusion on the task buffer for these tasks. Their implementation may be changed for the target OS.

Main Routine Implementation

The main routine looks different for different OS. For example, the VxWorks main routine returns an integer and does not contain parameters. However, the Unix main routine contains two parameters and can be exited by the exit() call. So you must rewrite the <main> routines in order to port them to the targeted OS.

CTRL-C Handling

The CTRL-C handling mechanism differs in targeted OS. First of all, some OS need to reset it every time after the SIGINT (CTRL-C) happened. Besides, the set handler service is named in different way for targeted OS. Our assumption is that the generated code without debugger (PDB)option, does not handle CTRL-C since the OS takes care about it. But sometimes this is not correct. In this case, add CTRL-C handling mechanism to the <main> function.

Target Description File Setting

The new library **svrcom** has been added into .trg/.rtrg files. It should be linked together with GBA or/and RPGERTL library and should follow them in the corresponding paragraphs of these files. See the *Code Generator User Guide* for more details.

Files and Libraries

The following files contain target-dependent parts of run-time libraries.

INTRINSICS Library

- **OS_FLAGS.H** file defines the flags for target platform. The flags for the new target should be defined here.
- **OS_INCLUDE.H** file contains the mapping of the common-named macros on the target-dependent functions to their calls. To understand which routines execute the same function in different target OS, compare the same macro for different OS. For instance, **OS_wake_main_task** macro is mapped to the vx_wake_main_task for VxWorks and it does not exist under Windows and Unix. Write a similar function <new_target>_wake_main_task, place it into the <new_target>_timer.c file and declare the proper macro in the OS_INCLUDE.H file as shown:

```
#Ifdef <new_target>
#Define OS_wake_main_task <new_target>_wake_main_task
#Endif <new_target>
```

  This file contains target dependent macro definitions, declarations, and constants. Create the new #ifdef-#endif section for the new target platform.

- **OS_create_task** is the macro definition, which allows creating target-oriented task. It has the following parameter

P1	task name
P2	task body function name
P3	end of task callback function name
P4	task priority
T1	first task body function parameter
T2	second task body function parameter

- **OS_CTRLC_handler** sets the SIGINT handler function and contains a single parameter, which is the function name.
- **OS_stderr_redirect** redirects STDERR to a temporary file since some OS can give an error message in when opening of the. pdbrc file if it does not exist.
- **OS_stderr_return** returns the previous STDERR file descriptor.

The couple of macros, mentioned above, is used for run-time debugger (PDB) only.

- **OS_exit** sets the operator that should be called when the program is finished. Different OS may need different exit command (e.g., VxWorks needs the return () command instead of the exit () function call in Unix).
- **OS_getuid** macro is used in the TCP/IP protocol implementation. See the get_default_port_number function in the gba and the rpgertl libraries.
- **OS_main** is the main program name including parameters and the "{" character.
- **OS_pause** allows to replace the pause () function call by some other OS service call, existing in the target OS.
- **OS_pr_pause** sets the implementation of the Rational Statemate pause "adapter," which allows the exit from WAIT state and monitors the CTRLC or other external event.
- **OS_reset_CTRLC_handler** resets the SIGINT/CTRLC handler if it is needed; otherwise it is an empty definition.

  Some operating systems need to forbid the new SIGINT event appearing via its handling. In this case the macro **OS_disable_signal** is called.

  The next group of macros concerns the semaphore implementation. Their names explain their functions: **OS_sem_create, OS_sem_delete, OS_sem_give** and **OS_sem_take.** All of them have single parameter – the semaphore descriptor name. Its type is **OS_sem_id** and it is defined in the **OS_INCLUDE**.H file too.

  The task synchronization is done using the **OS_synchronize** macro call. It has one parameter – time delay.

  All active tasks should be deleted when the program is finishing. It should be done using the **OS_terminate_tasks** macro.

  It is necessary to call the timer handler every step if the timer is implemented using separate thread/task. In other cases it is not needed. So this call is implemented by the macro **OS_timer_call_handler**, which should be defined properly for the target OS.
- **OS_timer_handler_p** is either NULL or the function that is called every step in the main function in order to check whether the next time-out event occurs or not.
- **OS_timer_start_p** is either NULL or the function that starts the timer.
- **OS_timer_stop_p** is either NULL or the function that starts the timer.

  If the generated code is running under the PDB debugger, it is necessary to exit from the timer. This can be achieved using the **OS_timer_exit** call.

  If the behavioral module is in a stable state, it can be awakened. It is needed if some external event happened. The awakening mechanism is platform dependent, so the **OS_wake_main_task** macro is used in addition to the sched_resume function call. See the try_to_wake_main_task function in the intrinsics library.

SCHEDULER/TIMER Library

- **LOW_TIMER.C** contains timer enable/disable/pause functions and Unix timer implementation. The following are the Unix-destined routines and data:
  - Signal mask sets (the **alarm_set** and the **pause_set).**
  - The pointer to the SIGALARM signal handler routine (the **timer_handler**).
  - **start_time** variable to save the time shift for model time calculation.
  - **handler()** of the SIGALARM signal.
  - **OS_timer_set()** sets the new interval timer according the closest started time-out event.
  - **OS_timer_reset()** initializes the interval timer by zero value.
  - **OS_timer_init()** sets SIGALARM signal handler.
  - **OS_timer_pause()** suspends the main program until delivery of a SIGALARM signal.
  - **OS_timer_disable()** calls the function that blocks SIGALARM and, as a result, timeout events cannot happen.
  - **OS_timer_enable()** calls the function that unblocks SIGALARM.
  - **OS_timer_get_time()** returns current model time value.
- **VXTIMER.C** contains VxWorks timer implementation. All of its routines and data items are VxWorks-dependent.
- **DOSTIMER.C** contains Windows timer implementation. All of its routines and data items are Windows-dependent.

  These files (vxtimer.c and dostimer.c) should not be touched and may be used as a pattern.
- **SCHEDULER.H** contains the **task_entry** data structure, which depends on the target OS. This data structure contains the target dependent fields for every OS and the same savearea field but with the different length. Declare this field correctly and define (if it is needed) some additional OS-specific fields. The RTL part of this file is located under the **"#ifdef prt"** definition.
- **CONTEXT_SWITCH.C** implements context switching between different threads**.** This operation is completely OS-dependent, so it should be rewritten for each new target OS.

- **SCHEDULER.C** supports initialization and execution of the tasks. The OS–dependent parts are the **task_entry** data structure **initialization** and context switching implementation. In addition, there are some task manipulation functions. They use target OS services and should be changed when porting to a new target OS. The following routines should be changed for the porting purpose:
  - **Init_task_entry()** initiates the task_entry data structure.
  - **sched_init()** initiates the main task and a set of the pointers for the using by other libraries.
    - **stack_overflow()** checks the status of the task.
    - **scheduler()** controls context_switching between the tasks.
    - **create_task()** allocates the new task_entry data structure and initiates it
    - **terminate_tasks()** (currently it exists for VxWorks only).
    - **create_VxWorks_task()** (for VxWorks only).

Server Communication Utilities Library

The **svrcom** library contains the TCP/IP socket connection utilities. It is a set of functions to initialize, read, write and close sockets for remote communication between the remote panel/trace/gba client and servers. All OS-dependent parts are placed under the "#ifdef" operators and self documented.

The library consists of one file rcomm_util.c, which contains the following OS-dependent routines:

- **InitializeWinSock()** (for Windows only)
- **OS_read_socket()**
- **OS_write_socket()**
- **HandleSigPipe()** (for Unix only)
- **OS_connect()**
- **OS_getprotoname()**
- **OS_gethost_addr()**
- **OS_initialize_sock()**

Remote Panel Library

The library calls the OS-dependent routines, which reside in the **svrcom** library. In addition the **CONNECT.C** file contains one OS-dependent routine **wait_for_msg (),** which needs to be ported to the new RTOS.

GBA Library

The library calls the OS-dependent routines, which reside in the **svrcom** library. No need for porting to this specific library.

Building the Libraries

There are special scripts/makefiles for building libraries. They are placed into the $STM_ROOT/etc/prt/c and $STM_ROOT/etc/sched directories. Their names are of the form CREATE_ xxx. In order to build target oriented set of libraries, copy the proper scripts, change the platform flag (for example, “-DVxWorks”) and add additional flags if needed. The following scripts/makefiles exist:

Script	Makefiles
Unix library creating scripts	• Create_intrinsics • Create_dbg • Create_sched • Create_sched_sim • Create_gba • Create_rpgertl • Create_svrcom
Windows library creating batch files	• Create_intrinsics.bat • Create_dbg.bat • Create_sched.bat • Create_sched_sim.bat • Create_gba.bat • Create_rpgertl.bat • Create_svrcom.bat
VxWorks library creating makefiles	• Create_VxWorks_intrinsics • Create_VxWorks_dbg • Create_VxWorks_sched • Create_VxWorks_sched_sim • Create_VxWorks_gba • Create_VxWorks_rpgertl • Create_VxWorks_svrcom