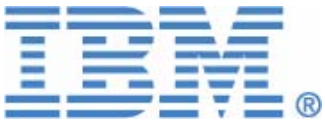
Rational. Statemate

Code Generation Reference Manual

Rational Statemate
Code Generation Reference Manual



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 BM® Rational® Statemate® 4.6 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Code Generation Basics

IBM Rational Statemate is a systems design automation tool for the development of reactive systems. In analyzing a design concept, a systems designer uses the Rational Statemate graphics editor to build and validate a graphical model of the system being developed, together with its user interface. The designer then analyzes the model on a workstation to verify its behavior using both static and dynamic analysis of the model's design concepts. Having validated the concept in this way,

Rational Statemate is then used to generate a C-based or Ada-based prototype of the design, based on the model, which can then be run on an appropriate host. The generation of prototype code in C or Ada is the subject of this document. There are three software code generator options:

- C (K&R or ANSII standard)
- Ada
- Embedded C

Standard C code has long been the preferred language for system designers and software developers. It was the original language generated by Rational Statemate. The C code generated by Rational Statemate is compatible with a variety of modern C/C++ compilers, including GCC, Visual C/C++, and Borland C/C++. Consult the release notes for your version of Rational Statemate for the latest compatibility list. Part 2 of this manual goes into extensive detail on the process of generating C code for a Native Host Environment (as opposed to the Rapid Prototyping environment discussed below).

Ada is another language supported by Rational Statemate. Generating Ada code has few differences from the process used to generate C code. Therefore, Ada specific information is discussed in **Ada Code Generation** of this manual. Consult the release notes for your version of Rational Statemate for the latest compiler compatibility list.

The Embedded C option is a central part of the Embedded Rapid Prototyping capability. This allows code generation to be taken to the next logical step: compilation and linking for use in an Embedded Prototyping Development System used to test the model designed in Rational Statemate in a prototype system environment. Rapid Prototyping allows a more function use case testing environment for moving the validation of the system design closer toward the end product. This guide also focuses on the details of Rapid Prototyping Code Generation.

Development Model

Rational Statemate facilitates a design process that begins with the construction of a graphical model of a design concept. This design concept is expressed as a set of charts, including *statecharts*, *activity charts*, and *module charts*. The designer creates this graphical representation of the desired product based on a written specification.

Chart	Design Focus	Shows
Statecharts	Behavior	How each function performs its job. The logic, ordering, and stimulus/response of each function.
Activity Charts	Function	How the functionality of the system is decomposed. The interfaces between functional units.
Module Charts	Structure	How the system is partitioned structurally. The interfaces between structural units.

Rational Statemate takes these charts and integrates them into a comprehensive formal model of a system that not only communicates the design intent clearly and precisely, but serves as a solid foundation for meaningful analysis and simulation. The designer creates these charts using Rational Statemate's language-sensitive and intuitive graphics editors. The editors all work in essentially the same way, though each is optimized for the type of chart being created.

The Properties editor facilitates the precise definition of the type and structure of all data and control elements. It also allows creation of user defined types, including records, unions, queues, and arrays. The properties can also be used to add information such as comments, descriptions, and attributes to all of the elements in the model.

Once the model is built, it can be verified through simulation. A successful simulation suggests a good working model. The Check Model tool performs a more exhaustive verification to ensure that the model is complete and consistent.

Executable Model

As a designer creates a model, Rational Statemate builds a formal mathematical representation of the model that can be dynamically analyzed on a computer at any time. In conjunction with debugging and analysis tools, the model can be refined even further. Dynamic Testing can be used to eliminate many logical problems that might otherwise not be found until the system is built and in the field.

The end result is a system design embodied in a formally defined working model of the system's functionality. This model can then be compiled using C or Ada source code generated by Rational Statemate, including the model's graphical interface panels. This compiled code can be run independently of Rational Statemate on another code-compatible computer.

Alternately, the Rapid Prototyping C code generator can be used to create code suitable for compilation/linking/downloading into a embedded prototyping development system. This would then allow the testing of the model within a prototype use-case environment.

Code Generation, and subsequent compilation, is the focus of this manual. The Rational Statemate code generators are consistent in their interface and basic functionality. This interface and functionality are the focus of the balance of this document.

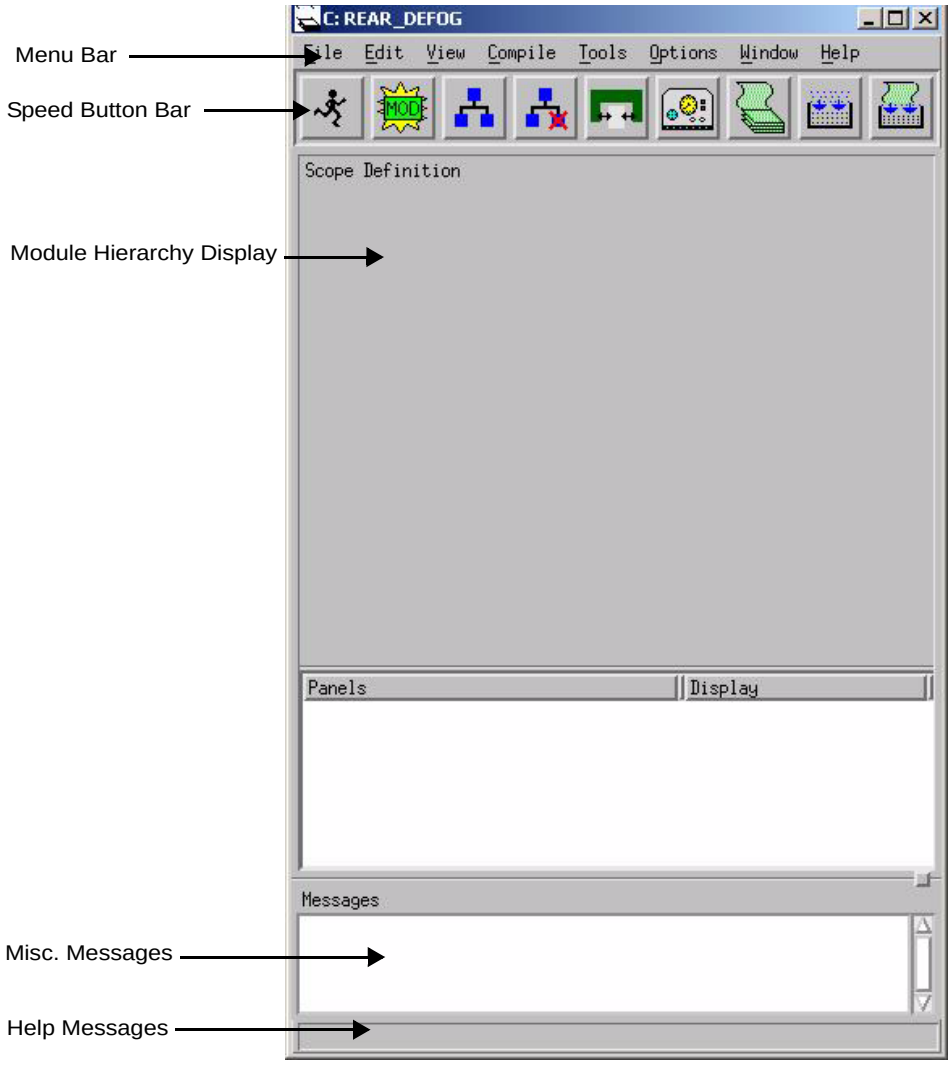
Generating Native Code

Generating code for a native environment is the simplest code generation task. The more complex task of generating code for an embedded development system environment is inherently more complex and is presented later in this manual.

The more complex a simulation model becomes, the more difficult it can be to "step through" the animated model. The ability to generate and compile code that will execute all or part of the simulation makes the over-all design process more efficient. Individual portions of the larger model can be refined to a point of design stability and stored as compiled code modules. These modules can be linked into a larger and more complex model as though they were "black boxes," thereby limiting the "under development" elements of the larger model that must be tested and refined.

It is also conceivable that an entire design can be compiled into an executable file that can be run on a system other than the development system. This allows sharing the model with others interested in the end result of the design effort.

Code generation within Rational Statemate follows a rather consistent process. This process is described in the following flow chart. It is important to note that most of the process occurs within the Code Generation Profile Editor window. However, it is also necessary to access the Workarea Browser in order to select the specific Activity Charts and Panels that are to be brought into the scope of the generated code.


Menu Bar
Speed Button Bar
Module Hierarchy Display
Misc. Messages
Help Messages
C: REAR_DEFOG
File Edit View Compile Tools Options Window Help
MOD
Scope Definition
Panels
Display
Messages

Concepts and an Example

This section describes the major Code Generator concepts that you need to know and provides an example that demonstrates how to generate code.

Compilation Profile Concepts

The behavioral model consists of many Statecharts and Activity-charts that you may want to segment into smaller components. With the Code Generator's Profile Editor, you can define the scope of the compilation profile that you want to compile into C or Ada. You can also customize the generated code by specifying several translation alternatives.

Profile Editor

The **Profile Editor** allows you to specify which charts to generate code from as well as select the parameters that control their style. Use the Profile Editor to define a compilation profile by:

- Defining code modules
- Assigning behavior (Statecharts and Activity-charts) to the modules
- Selecting preferences and settings
- Customizing the profile by adding testbenches and panels
- Generating host code (K&R C, ANSI C) (Ada is also supported; refer to **Ada Code Generation**).

Use a profile to save the scope and code generation options, then store it in a workarea where it can be retrieved, edited, and used over and over again for subsequent code generation runs. The profile can also be saved to the databank where it becomes part of a formal release or configuration item.

Module Structure

A **module** is a collection of Statecharts and Activity-charts that comprise a component.

- In C, a module signifies a single source file with its local data and functions.
- In Ada, a module signifies a library package.

Scope Definition

The **Scope Definition**, which is in the Profile Editor's main window, shows the Module structure of the profile in a tree format or as a list. Both views show the charts assigned to each module and how they were assigned.

Connection to the Workarea

The process of assigning behavior to the profile structure consists of three stages:

- **What** charts do you want. (Select the charts from the Workarea.)
- **Where** do you want them. (Select the module you want to assign the behavior to.)
- **How** do you want to assign them. (Select the method of assigning the behavior.)

Descendants

Descendants refer to all the subactivities that are lower than the current chart, down to the last state and primitive activity.

Testbenches

Testbenches (called Watchdogs in earlier versions of Rational Statemate) are separate Statecharts created outside the specification of the system being developed.

Testbenches trap a specific behavior to test a design's inputs and outputs. It's a "snapshot of a scenario." Testbenches also serve as debuggers, and they are visible to all signals in the design without having to draw discrete flows.

Note

Testbenches cannot test generics.

Concurrency

How does Rational Statemate translate *concurrent* activities into a *sequential* language? Even though one procedure in the generated code may call another, if both are executed in the same cycle, they are concurrent.

Sometimes it seems natural to implement concurrent activities as different threads (tasks), but it is also possible to implement them as a single threaded program. Writing a single or multi-threaded embedded application is a design decision that does not affect performance or modularity. Since the underlying architecture is sequential, a multi-threaded program is actually a set of sequential pieces managed by a sequential handler.

So why is multi-threading needed at all? A multi-threading capability is needed only if a designer wants to add threads that run "concurrently" with the generated modules that execute as a single thread, denoted as the "main task."

Graphical Back Animation (GBA)

The Rational Statemate Simulator highlights the charts as they are executed. Once you generate code, you lose that graphical feedback. Generating code with the **GBA** (Graphical Back Animation) option provides graphical highlighting similar to Simulation, but from generated code.

Inserting Handwritten Code

Code Generator's modular code architecture enables you to integrate handwritten code (also known as "user-written code") with Rational Statemate-generated code in two ways:

- The profile's scope can include stubs for handwritten code. A stub is an empty module where you can insert user-written code into Rational Statemate-generated code.
- The new method of supplementing code offers the following advantages:
  - Enables you to include code directly into your design.
  - Eliminates the need for special calls and services to integrate handwritten code.
  - Stores the code in the model's database so it is common to both simulation and code generation.
  - Automatically includes the user-written code whenever you run simulation or code generation.

For information on these methods, refer to **Required User-written Code.**

Note

Rational Statemate supports the stub method for compatibility reasons, but it is recommended that you use the new method for supplementing code.

Creating a Sample Profile

This section shows how to create a sample profile and generate code for it.

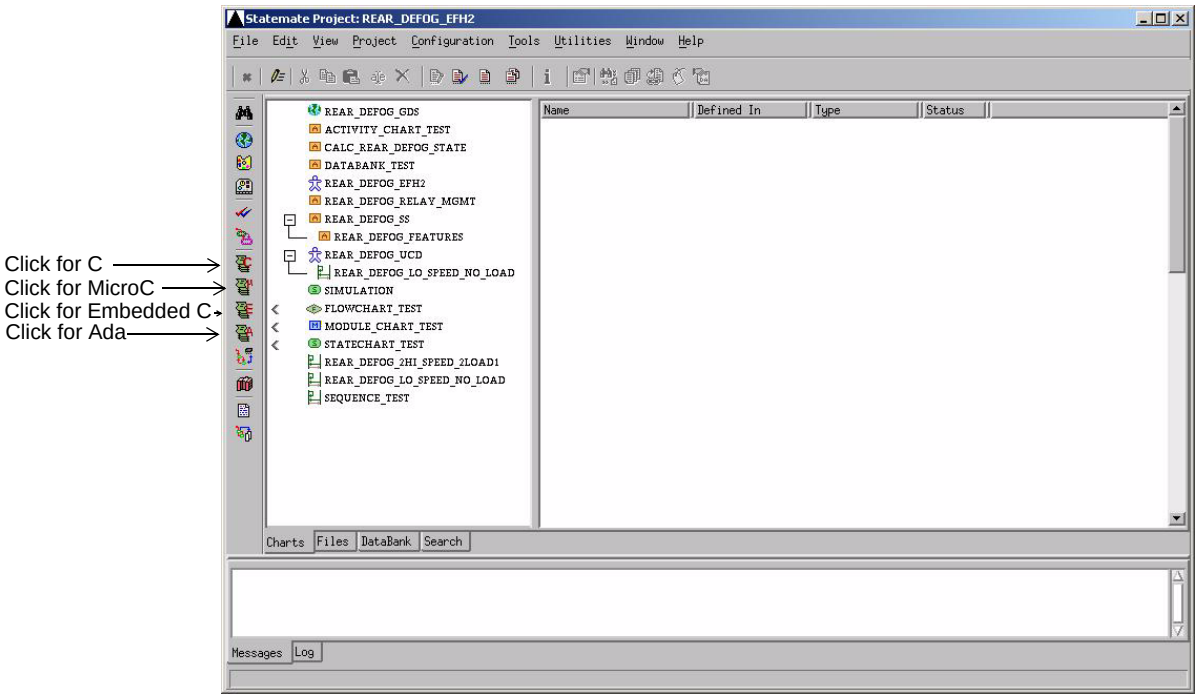
Invoking the Profile Editor

Use the following steps to access the Profile Editor and create a new profile.

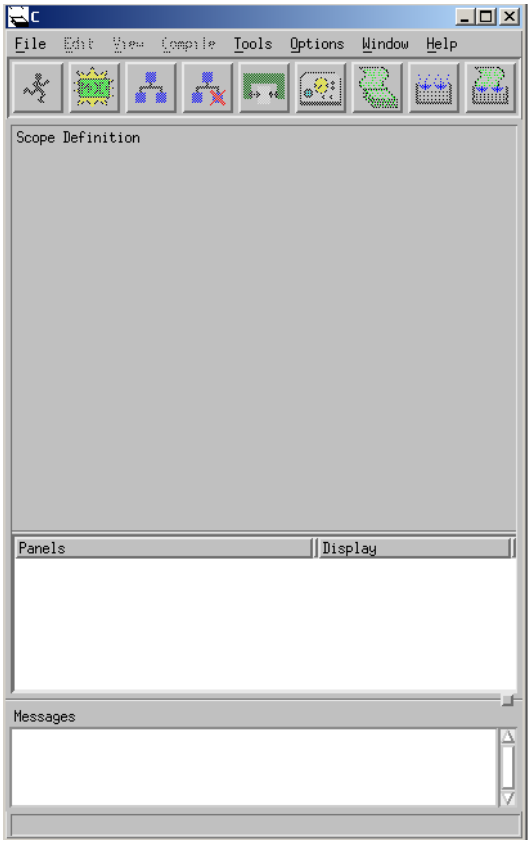
1. Select the **C** code generator from the main window.

Note

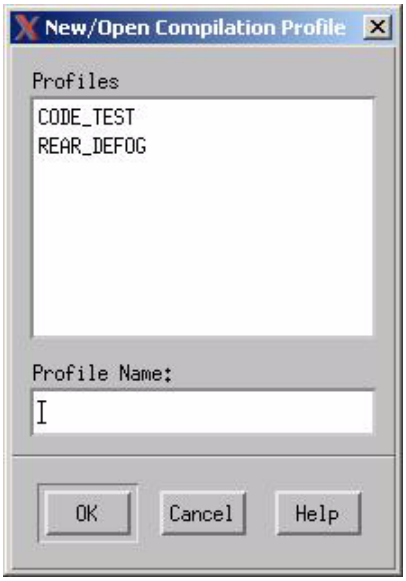
If you are developing **ADA**, **Embedded C, or MicroC** code, you need to select the appropriate icon.



The appropriate Profile Editor appears as shown in the following figure.



2. Select **File > New Profile.** The **New/Open Compilation Profile** dialog appears.



3. Name the new profile in the **Profile Name** text box and select **OK**. For example, in previous figure, the profile is given the name: **REAR_DEFOG**.

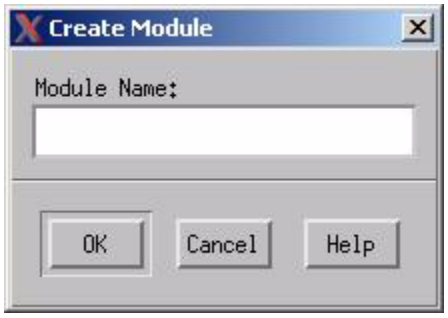
The **Profile Editor** enables all the menu selections and displays the profile name in the title bar.

Defining Code Modules

Code module can be structured as desired to meet the needs of the model being simulated. Use the following steps to define the structure of modules that reflects the way you want the code organized. Note that each module may contain one chart, several charts, or a portion of a chart.

1. Click **Create Module**  or select **Edit > Create Module**.

   The **Create Module** window opens.



2. Enter the **name** of the new module and click **OK**.

Note

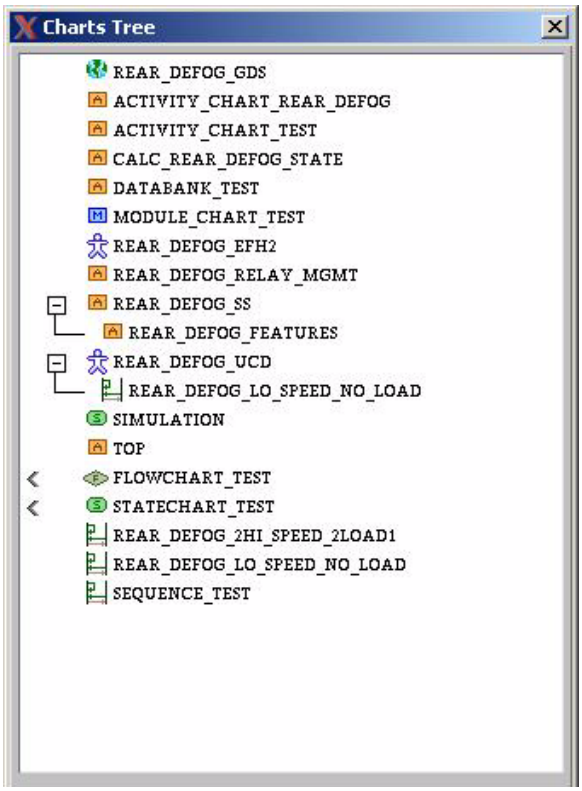
For the example presented here, the name the module is **REAR_DEFOG_MOD**.

Assigning Behavior to the Module

Use the following steps to select charts from the workarea and assign them to the module you want.

1. Click **Add Chart to Module**  or select **Edit > Add With Descendants**.

   The Chart Tree windows opens.



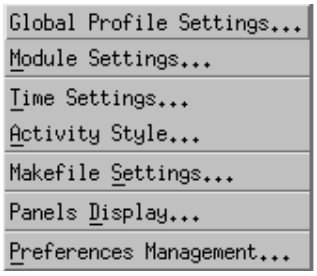
2. Select the chart(s) you want to assign to the module. For example, select the Activity-chart **REAR_DEFOG_SS**.

   Note: To select charts with their descendants in a hierarchy, select only the parent. The Code Generator adds your selection to the profile with its descendants.

3. Select the **File** > **Save** menu item to save the profile in your workarea.

Selecting Code Parameters

Use the **Options** menu to specify how you want the code generated.



Option	Description
Global Profile Settings	Allows you to change settings for **all modules** in the current profile. Use this feature to select the following: • Language (For the C Code Generator, you can select K&R C or ANSI C.) • Modularity Style • Double-Buffer Optimization • Generation of main • With Debugger • Graphical Back Animation • Infinite Loop Limit • Packages/Headers for External Subroutines
Module Settings	For the **selected module**, you can either create a separate file for each Statechart or set parameters.
Time Settings	Control time expressions settings (real, synchronous, or asynchronous) and in what units.
Activity Style	Specifies software or hardware. If you select software, activities can be started and stopped. If you select hardware, activities are always active.
Makefile Settings	Allows you to set flags for compilation and include libraries.
Panels Display	Allows you to have panel display on other workstations connected to the network.
Preferences Management	Allows you to select **general preferences**.

Generating Code

Before generating code, you may wish to run Check Profile to verify that the profile complies with the scoping rules. For example, Check Profile makes sure that the settings are legal and do not conflict with each other. The Code Generator also checks the profile when you generate code.

Complete the following steps to check the profile and generate code.

1. Select **Compile > Check Profile**.

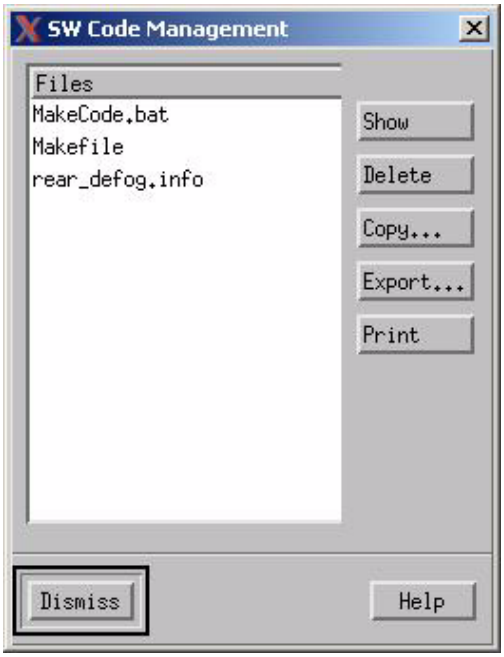
   Note: You must correct any errors before generating code, but you can continue to generate code with warnings and information messages.

2. Select **Compile > Generate Code**.

   Note: The default location for this operation is a sub-directory named according to your current profile name, i.e., it is inside the “prt” sub-directory of your current workarea.

3. Select **Compile > File Management**.

   The **SW Code Management** dialog appears



File Management offers the following options:

- **Show**—Displays a listing of the generated code.
- **Delete**—Deletes the selected file.
- **Copy**—Copies the selected file after you re-name it. (Works the same way as **Save** as in the **File** menu.)
- **Export**—Saves the file to another workarea or directory.
- **Print**—Prints the generated code.

4. To view generated code, select a file and then click **Show**.

   The selected file appears in the xless editor similar to the example in the following figure.

5. Select **Quit** and then **Dismiss**.

6. Select **one** of the following*:*

   - **File > Close** to close this profile and leave the Profile Editor open.
   - **File > Exit** to close this profile and close the Profile Editor.

The Profile Editor automatically closes any related windows that you left opened and displays a dialog asking if you want to save any unsaved changes to the profile.

File Edit Format View Help

```
  int                   RD_TIMER;
  int                   VEHICLE_SPEED_IN;
  /* Vehicle Speed from Powertrain (KHP).*/
  event  tmENRD_RELAY_OFF;
  event  tmENRD_RELAY_ON;
  event  tmENREAR_DEFOG_TIMED;


static void schedule_timeouts()
{
  if (ENRD_RELAY_OFF)
    sc_tmo(&tmENRD_RELAY_OFF, OFF_TIME * SEC);
  if (ENRD_RELAY_ON)
    sc_tmo(&tmENRD_RELAY_ON, ON_TIME * SEC);
  if (ENREAR_DEFOG_TIMED)
    sc_tmo(&tmENREAR_DEFOG_TIMED, RD_TIMER * SEC);
} /* schedule_timeouts */

void exec_REAR_DEFOG_OUTPUT_TT ()
{
  if ( DEFOG_DRIVE_SIG )
  {
    tt_notify((void *)scope_id,(genptr)exec_REAR_DEFOG_OUTPUT_TT,FALSE,1);
    setba(DEFOG_DRIVE_OUT,1,0,0,s2ba("0b0"),1,0,0);
    return;
  }
  if ( ! DEFOG_DRIVE_SIG )
  {
    tt_notify((void *)scope_id,(genptr)exec_REAR_DEFOG_OUTPUT_TT,FALSE,2);
    setba(DEFOG_DRIVE_OUT,1,0,0,s2ba("0b1"),1,0,0);
    return;
  }
} /* exec_REAR_DEFOG_OUTPUT_TT */

void exit_CALC_REAR_DEFOG_ENABLED ()
{
  notify(scope_id,conCALC_REAR_DEFOG_ENABLED,FALSE);
  switch (CALC_REAR_DEFOG_ENABLED_isin) {
    case REAR_DEFOG_TIMED :
      notify(scope_id,conREAR_DEFOG_TIMED,FALSE);
      break;
    case REAR_DEFOG_NOT_TIMED :
      notify(scope_id,conREAR_DEFOG_NOT_TIMED,FALSE);
      break;
    case notaCALC_REAR_DEFOG_ENABLED :
      break;
  }
  CALC_REAR_DEFOG_ENABLED_isin  =  notaCALC_REAR_DEFOG_ENABLED;
```

Architecture of Generated C Code

This section describes the architecture of the generated C code including how the Code Generator structures the modules.

The Rational Statemate Code Generator generates fully functional code, based on the Activity-charts and Statecharts in the Rational Statemate model. The generated modules are partitioned according to a compilation profile, which allows you to generate code for a complete Rational Statemate model or just a part of one.

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).

The generated code uses runtime modules for timing and scheduling. Requests are generated to the timing module for timeouts and scheduled events, and to the scheduler module to control handwritten tasks that are connected to basic and external activities. In addition, the data elements are double buffered, so data assignments are synchronized to prevent racing conditions among the "concurrent" behavioral components.

Note

In some cases where there are no racing conditions, you may want to disable double buffering. For more information, refer to **Optimizing Double Buffers**.

Code Libraries

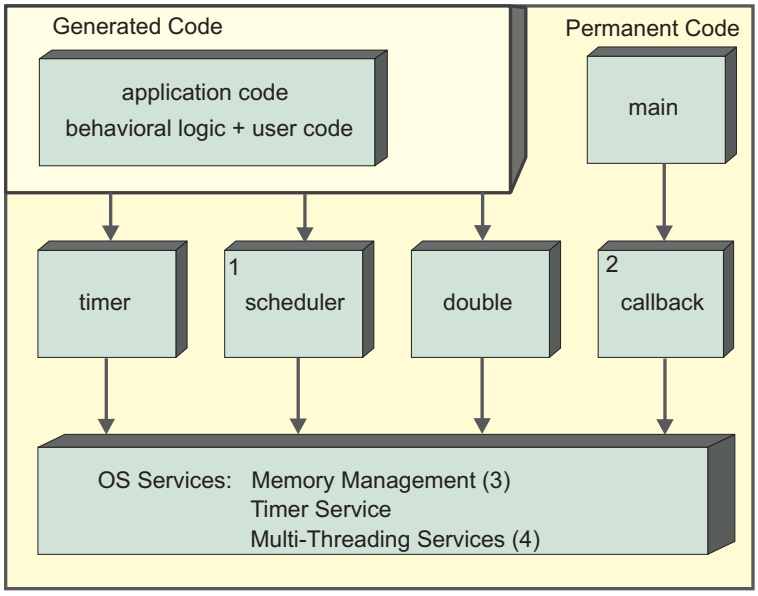
All of the runtime modules are actually a set of compiled libraries. These libraries can be reused for other projects because they are supplied in source code form which allows modifications based on project-specific requirements. The runtime modules actually provide an *interface* between the generated behavioral logic and the underlying Operating System (OS).

Porting the generated behavioral components to a particular environment primarily means tailoring the runtime library to use the specific services provided by the operating system. The target operating system can even be a Real-Time OS kernel. The runtime library can even be modified to provide an alternative functionality which does not normally exist in the target OS.

Note

Tailoring the runtime libraries is a usually one-time effort. Once completed, the generated components can be compiled and linked without being modified any further.

The detailed process of generating code for an embedded target system is discussed in detail later in this manual. However, there are many basic OS concepts that are more easily introduced using the more limited scope of an embedded operating system. The following figure shows the layered software components of the typical embedded application. The final executable image is normally built from some permanent pre-compiled modules (such as the RTOS kernel) and the generated/ compiled modules that are dependent on the application.



The key components include the following:

- **The Scheduler Component is Optional** - It is needed only if the user specifies that basic or external activities should be implemented as tasks or desires to link a graphic panel into the executable.
- **Callback Handler.** - This component will be used only if the user selects to attach callback routines to behavioral logic components.
- **Memory Management** - The runtime module's timer, double-buffering and callback handlers utilize dynamic memory allocations. Under certain assumptions it is possible to tailor them to use only static allocation, if a memory management package is not available or memory resources are limited.
- **Multi-Threading (Tasking) Support -** This support provides a mechanism for creating task threads and switching between them. This service is needed only if the user wishes to implement environment tasks or basic activities as tasks. This issue is discussed in greater detail in the *Software Code Generator Interface Manual*.

Tasks View of the Code

One of the major issues that confuse many users is how *concurrent* activities and states are actually translated into a sequential language. 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.

Writing an application as a single thread or multi-threaded is actually a design decision. Since the underlying architecture is sequential, a multi-threaded program is actually a set of sequential pieces managed by a sequential scheduler.

Module Execution

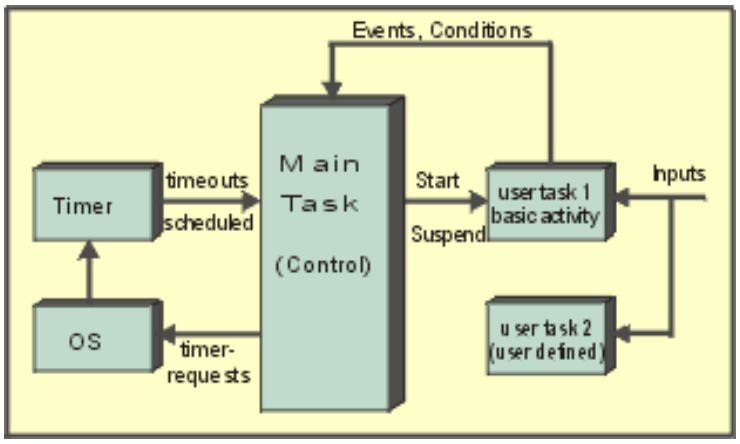
The modules of the generated code are sequential. They are executed cyclically with each iteration evaluating the next step of processing. In terms of simulation, executing the code is equivalent to executing a "go-step" repeatedly, while changing the environment asynchronously. The main difference is that the clock is incremented in real time, so timeouts will happen according to the time taken to execute the code.

Multi-Threading

So why is multi-threading needed at all? Multi-threading is used to allow the user to implement basic activities as independent processes, without having to comply with the “one cycle at a time” method. It also allows writing additional environment processes outside the system model, to process inputs, to drive outputs or for simulating the environment. Therefore, a multi-threading capability is needed only if the user wishes to add threads that run “concurrently” with the generated modules that execute as a single thread, denoted as the “main task.”

Asynchronous Timer

Another component in the process view of the code is the asynchronous timer. The main task issues timer requests to be notified about timeouts and scheduled actions. The timer module asynchronously notifies the main task when timeout events are occurring. An example is shown in the following figure.



- In some applications there will be no basic activities implemented as tasks. In those cases, the only processes that exist are the main task and the asynchronous timer. If basic-activity tasks exist, the main task issues tasking control calls such as start, suspend, etc.
- There are cases where the user implements environment tasks, but none of the basic-activities are implemented as a task. In these cases, the generated-code (the main task) does not use any tasking services. The code does not need a multi-threading adaptor unless the user connects a panel to the executable.

Using Simulated Time Model

Generated code uses the *real-time* model by default. In this model, timeouts and scheduled actions are treated very similarly to other inputs. The system clock keeps time and generates interrupts that are processed along with the other inputs.

When using this time model, it is possible for the code to miss a timeout or scheduled action due to heavy loading of the processor or an extremely small request for a timeout. In such a situation, the generated code may actually behave slightly different than a simulation of the same model.

An additional time model is provided called the *simulated-time* model. The purpose of this model is to force the generated code to behave in the same manner as the simulated model. It does this at the cost of the *real-time* nature of the generated code.

The simulated-time model may be either asynchronous or synchronous. In the asynchronous time model, time is consumed only for timeout statements and scheduled actions; otherwise, it runs in real-time. In the synchronous time model, transitions are made on a clock. Every transition consumes one clock period and every step consumes one clock cycle.

In order to meet all timeouts regardless of duration and CPU loading, the code would be required to run at an arbitrarily fast speed. Since this is not possible, code which is compiled using the *simulated-time* model, does not adhere to the system clock. Rather, it keeps its own artificial time, much the same as a simulator. The code executes model steps until it reaches a stable status. It then advances the internal clock to the necessary value to execute the next timeout or scheduled action.

```
-- The main loop, loops forever

int main(argc, argv)
       int argc;
       char **argv;
{
       while (TRUE) {
              -- Execute a step --
              -- Advance internal time keeper to next
              relevant time --
              -- Apply timeouts and scheduled actions. --
```

Implementing a Function to Get External Inputs

To retrieve external inputs, you can create separate tasks within the Rational Statemate model. This process is described in **Adding User-Written Code.**

Use the tasks to read inputs from the environment (possibly from the keyboard or an input file), and use the value setting functions to insert the changes into the Rational Statemate model. In order to simulate the passage of time, the `delay` function should be used between inputs.

The outputs can be captured using the event callback mechanism, or they can be polled using a separate task.

Extracting the Time

The function `sched_time (double)` returns the simulated time. It can be used by the handwritten code to decide when to stimulate the model or to generate reports.

Main Task: Partition and Flow Control for C

This section describes how 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 basic activity tasks are created. In addition the `user_init` procedure is called.

The `user_init` procedure resides in a file called `user_activities.c`. When you generate code, the Code Generator automatically creates the `user_activities.c` file and the *`user_init`* procedure. Prior to executing the model, you may initialize values in the `user_init` procedure.

After the 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.

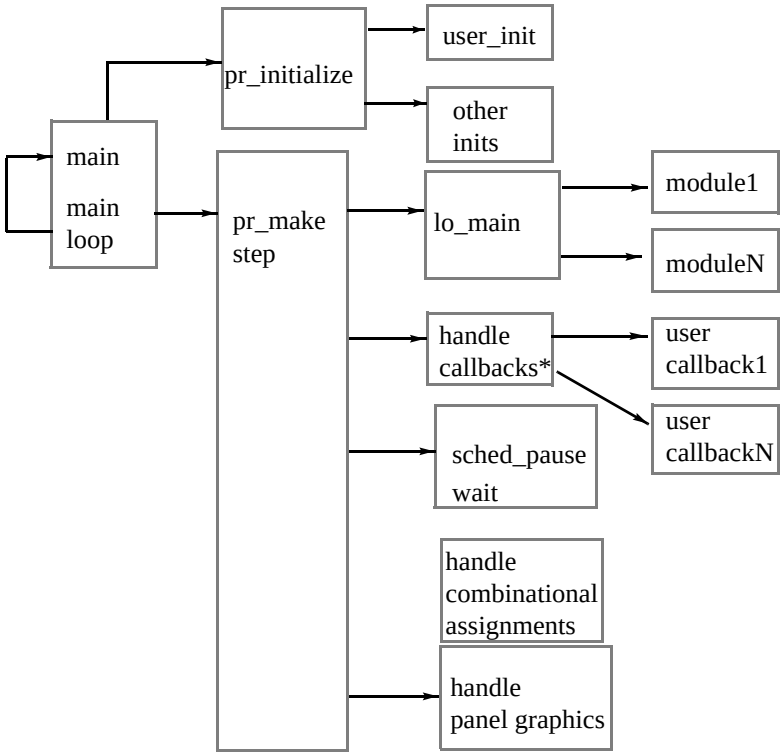
This is how the main program looks:

```
int main(argc, argv)
     int argc;
     char **argv;
{
   pr_initialize();
   while(TRUE) {
   if (pr_make_step()) /*if system is in stable status,
          pr_pause;it enters the pause mode, waiting
                 for external inputs */
   }
}
```

The C function `pr_make_step` returns TRUE when system is in a stable status.

The main program is written as a task, which calls all the state machines within the profile. `pr_initialize` is the initialization procedure, and `pr_make_step` completes a single-step of the whole system. Note the user-tasks, including basic activities are processing independently, as well as the asynchronous timer.

The following diagram shows the calling sequence within the main task:



The C procedure `pr_make_step` follows:

```
boolean  pr_make_step()
{
    boolean step_has_changes = FALSE;


    incr_stepN(); /* increment step counter */
    sched_disable(); /* disable async timer interrupts
                during execution of step */
    lo_main(); /* step execution */
    step_has_changes = update(); /* perform all deferred
                assignments */
    garbage_collect() ; /* clearing intra-step allocations */
    sched_enable() ; /* enable accepting of elapsed
                timeouts */
    scheduler(); /* yield control to other ready
                tasks, including panel driver */
    if (!step_has_changes) /* no changes:
           return TRUE;                    system is in a stable status */
```

```
    pge_start_graphics(); /* start critical section of panel
                  updates */
    call_cbks(FALSE); /* evaluate callbacks,
                  including panel outputs */
    pge_end_graphics(); /* end critical section of panel
                  updates */
    return FALSE; /* step finished;
                  system status is not stable */
}
```

The `pr_make_step` procedure activates all the functions that complete the execution of a step.

Activating the generated modules (the “state machines”)

`lo_main` is a generated procedure, that “glues” together all the specific modules as partitioned by the compilation-profile. It calls the top level procedures of these modules:

```
lo_main()
{
  <module1>_EXEC_all();
  <module2>_EXEC_all();
   ..........
  <moduleN>_EXEC_all();
}
```

Note

The `lo_main` is actually the scheduler of the generated components. It applies a fair non-prioritized round-robin scheduling policy, similar to the interpretive simulator. However, it is possible to introduce priority scheduling by modifying `lo_main`.

Updating double buffer assignments

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 system executed an idle step. The `step_has_changes` flag indicates whether the step has ongoing processing, or the previous execution cycle was actually an idle step.

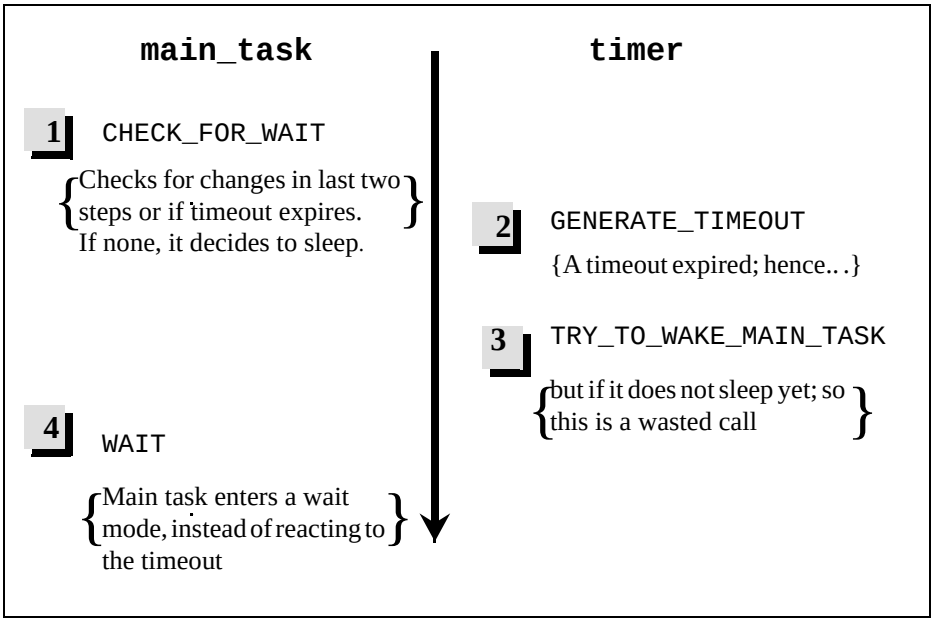
Evaluating the callback list

If you define callbacks, they are checked at this point. When an element gets a new value, its callback procedure is called. Note that if no hooks are set, the handle to the callback (`call_cbks_p`) handler remains null, and it is not called at all.

Entering the wait state

If the system executes an idle step, it is in a stationary condition. At this point, the main task releases the CPU by calling a system service that blocks it from running until some external stimulus occurs. The external stimulus can be either an event/data change, or a timeout.

The decision 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. Therefore, the `pr_pause` procedure that actually blocks the cannot be uninterrupted to prevent the following scenario:



This scenario leads to a deadlock condition. Since the timeout is ignored by the system, the main task has already “decided” to hibernate itself but has not yet done so and the “wake” call is lost. The **pr_pause** procedure will apply the test-and-wait in a mutually exclusive manner.

The `pr_pause` procedure will be discussed later, since it is dependent on the underlying operating system.

Structure of a Behavioral Module

In this example, the module is called *l*`ight.c`:

```
#include “types.h”
#include “<gds_name>.h”

#include “light.h”
#include “<compilation_profile_name>main.h”

#include “user_activities.h”
```

Headers of other modules:

- **types.h** — Basic type definitions for condition, events, data-items etc.
- **<gds_name>.h** — If you used Global Definition Sets in the design, the GDSs will have corresponding header files in the generated code. They are included by all modules in the code.
- **light.h** — Header file for the local module.
- **<compilation_profile_name>main.h** - Definition of all intermodules shared data. Note that all the data elements shared by more than one module are defined in the main module.
- **user_activities.h** - Prototypes for the user-written activities for which stubs were requested in the profile.

Interface Section

This section documents the inputs and outputs flowing into/out of the module. It is useful for reusability purposes to understand the interface of the module.

```
/* Inputs */
/* event CAR; */
/* event ADVANCE; */

/* Outputs */
/* condition YELLOW_C; */
```

Status Types

Every non-basic or-state has a status variable that indicates what substate is currently active. The status type is actually an enumerated type, defined at the beginning of the module.

```
typedef enum {nota_Chart_TMODES, st_LIGHT_MODES} tp_Chart_TMODES_states;
```

State Variable Definition

```
tp_Chart_TMODES_states  st_Chart_TMODES_isin = nota_Chart_TMODES;
```

Definitions of Data/Control Elements

In this section all the LOCAL data-items, events and conditions are defined. The word LOCAL means that the elements are not used outside the module scope. Note that activities are also allocated a status variable of type *activity*, which is an enumerated type that contains the possible activity statuses.

```
activity   acy_INPUT_TASK = nonactive;
condition GREEN_C = FALSE;
```

Definition of Fictive Events/Conditions

The fictive events are events not explicitly defined in the model. Thus, they can be thought of as "Statemate-generated events." They are essentially timeout events, enter/exit state events, and in-state conditions. Fictive events are generated only when necessary, i.e., only if the model uses `en(STATE)` and the `enst_STATE` event is generated.

```
event  enst_CHANGE1 = FALSE;
event  tmenst_CHANGE3 = FALSE;
```

Definition of Truth-Table Elements

The following lists the code generated out of a Truth-Table:

- The expression of the format: "if(<Boolean>==true)" is generated as "if(<Boolean>)."
- The expression of the format: "if(<Boolean>==false)" is generated as "if(!<Boolean>)."
- The expression of the format "if(true)" generates the expression under the "if" without an "if" statement.

Schedule Timeouts Procedure

This procedure executes every execution-cycle, and evaluates what timeouts should be triggered in the particular module. All timeout triggers are evaluated, and the necessary timeouts are SCHEDULED using the timing module service `sc_tmo`.

```
static void schedule_timeouts()
{
  if (enst_CHANGE3)
    sc_tmo(&tmenst_CHANGE3, 0.3 * SEC);
} /* schedule_timeouts */
```

Action Procedures

In some cases actions are translated into procedures (depending on the modularity style). In this case, a C procedure represents the Rational Statemate action DO_BLACK.

```
void exec_DO_BLACK()
{
  setc(&YELLOW_C, FALSE);
  .........
} /* exec_DO_BLACK */
```

State Enter/Exit Procedures

Depending on the modularity style, the enter/exit (including history enter) sequences are grouped into procedures. The example shows the default entering sequence (i.e. entering via a transition that goes to the edge of the state) for the NIGHT state:

- Change parent status variable to NIGHT.
- Generate the event en(NIGHT) represented as enst_NIGHT.

```
void entdef_st_NIGHT()
{
   st_LIGHT_MODES_isin = st_NIGHT;
   gen(&enst_NIGHT);
   ..........
} /* entdef_st_NIGHT */
```

State EXEC Procedures

```
void EXEC_st_LIGHT_MODES()
{
  switch(st_LIGHT_MODES_isin)
  {
    case  st2_NIGHT  :
        EXEC_st2_NIGHT();
        break;
    case  st2_STD_BY  :
        ............
  }
} /* EXEC_st_LIGHT_MODES */

void EXEC_st_Chart_TMODES()
{
  switch (st_Chart_TMODES_isin) {
    case  nota_Chart_TMODES  :
    case  st_LIGHT_MODES  :
      EXEC_st_LIGHT_MODES();
      break;
    default:
  }
} /* EXEC_st_Chart_TMODES */
```

The EXEC procedure is actually the heart of the behavioral logic as described in the statecharts. Every non-basic state has an EXEC procedure that activates all the state-logic within a single execution cycle. The EXEC procedure will take care of in state transition, static reactions, and activation of substate EXEC procedures.

The traversal is done hierarchically, starting at the very top state in the module, going down towards the basic states. In case of an “and” state, the orthogonal components are traversed sequentially one after the other but on the same semantic step utilizing the double-buffering mechanism.

Module Initialization Procedure

The module initialization procedure is called once the executable is started, before running through any execution cycle. It initializes all local data of the module. The procedure also establishes tasks that implement basic activities if there are such:

```
void light_init()
{
  tp_acy_INPUT_TASK=
    sched_create_task(user_code_for_input_task,
      0,stop_acy_INPUT_TASK);
}
```

The init procedure is one of the two procedures that the module exports, and it is called by the `lo_init` procedure.

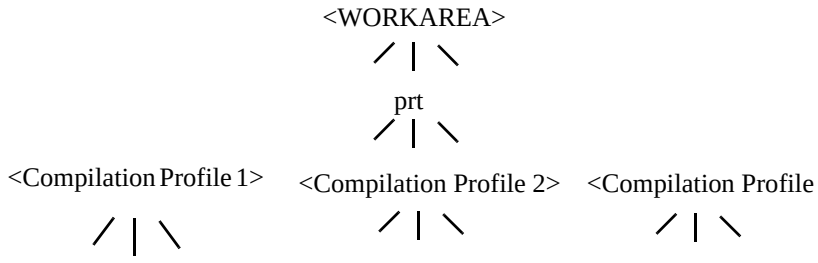
Module Execution Procedure

```
void light_EXEC_all()
{
  schedule_timeouts();
  EXEC_st_Chart_TMODES();
} /* light_EXEC_all */
```

This procedure activates a single execution cycle (step), once being called by `lo_main` in the main module. It activates the `schedule_timeouts` procedure to schedule potential timeouts, and most importantly, is activating the hierarchical traversal of the state EXEC procedures by activating the EXEC procedure of the top-level Statechart.

Structure Of The Generated Code

The Code Generator writes the generated source code into a designated directory in the workarea. The organization of the code directories in the workarea is as follows:



For every code generation, the code is placed in a subdirectory of the “prt” directory based on the compilation profile name (unless you specify another Output Directory.

In the figure above, `<WORKAREA>` stands for the root of the workarea directory, and `<Compilation Profile 1>`, `<Compilation Profile 2>`, and `<Compilation Profile 3>` represent directories corresponding to different profile names.

Structure of the Output Source Files

The generated files can be partitioned into six categories:

- **Control Modules** - These files carry the model’s logic and scheduling and are the most significant part of the model.
- **Modules for Subroutines Defined in Model** - For each subroutine, a separate file is generated.
- **User Supplemented Code** (templates) - These files contain hooks and frames used to interface the behavioral model with the environment or any other user-supplemented modules. As opposed to subroutines, this code is not stored in the model.
- **Interface Modules** - Interface code for panels and the Debugger.
- **Makefiles and Compilation Scripts** - These are scripts used to automate the process of building an application (compile & link) from the source files.
- **The Info File** - Contains cross reference information.

Control Files

There are two types of control files: behavioral modules and a top-level module.

Behavioral Modules

The behavioral modules are the heart of the code and implement the logic as described by the statecharts and mini-specs. The specification is partitioned into behavioral modules in the compilation profile. For each specified module, two files are generated based on the user-defined module name.

The following header file exports all the specification objects defined in the module (for use by other modules), and the module execution procedure.

```
<module_name>.h
```

The following module body defines all the local objects (events, conditions, data-items), and the procedures that implement the logic of the statecharts and mini-specs.

```
<module_name>.c
```

The Top Level Module

The top-level module identified as `<compilation_profile_ name>main.c` “wraps” all the behavioral modules into a single behavioral unit. It also defines all the global elements, i.e., those elements used by more than one module. It defines two procedures:

- `lo_init` - Initialization of all the participating modules
- `lo_main` - Execution of a single step of all modules.

The header file exports the global elements, the initialization and the execution procedures. The file name is:

```
<compilation_profile_name>main.h
```

Implementation of Subroutines

This section describes how the Code Generator implements subroutines in the model's database.

For information on how to supplement generated code with subroutines, refer to **Adding User-Written Code**.

For each subroutine that the model uses (i.e., either called or referenced in the callback or user-added code bindings), a separate file is generated. This file, named `<subroutine_name>_ sc.c` contains a code that implements the subroutine.

The Code Generator implements the subroutine in one of the following ways:

- Handwritten code (in C or Ada), which is stored in the model
- Translation of Rational Statemate Action Language, Procedural Statechart, or Truth Table (when a subroutine is implemented using one of these languages)

In the following cases, the Code Generator only produces a template for the subroutine:

- No implementation at all is given for the subroutine in the model.
- Some implementation exists, but it is disabled by setting the "Select Implementation" option in the Properties window to "None."
- Selected implementation does not match the Code Generator's target language. For example, if your model has a C-code implementation of a subroutine and you are generating Ada, the Code Generator will ignore it.

In addition to `*_sc.c` files, the Code Generator creates a file called `<profile_name>_envelopes.c`. This file contains "envelopes" for all the subroutines in the scope. The envelopes ensure that the Rational Statemate execution rules are properly mapped into those of C or Ada.

User Supplemented Files (User_activities Stubs File)

The stub method adds code to the generated C or Ada code by modifying the `user_activities` files. For more information on this method, refer to the *Software Code Generator Interface Manual*.

The Code Generator automatically creates the following two files:

```
user_activities.c (user_activities.c_temp)
user_activities.h (user_activities.h_temp)
```

These files include all the stubs generated for the basic activities according to the compilation profile. Once the user-activities stubs file exists in the output directory, it is not overwritten, and a file `user_activities.c_temp` is generated. The stub file includes a corresponding header file, which is also not overwritten.

Interface Modules

Debugger Symbol Table File

This file is generated only when the debug option is enabled. It includes symbolic information about the original model that is used by the debugger.

```
<profile_name>.dbg
```

Panel Interface Files

The following files are generated only if there are panels attached to your code. This is actually the code that glues the panel to the behavioral modules.

```
panel_transmitter.c
panel_displays.dat
```

Makefiles and Compilation Scripts

The following files compile and link the code on UNIX platforms where the “make” utility is available:

- Makefile
- User_Makefile

Info File

The info file (format below) contains information about the translation process, the relevant portion of the model, and the generated modules.

```
<profile_name>.info
```

This file contains the following information:

- **Compilation Profile Parameters**
- **Errors and Warnings**
- **Cross Reference Table** - The cross reference table contains all the elements in the code and the original elements they represent. This information is useful when supplementing the generated code. In cases where the same name is used for different model, this cross-reference table is the only way to identify which code-element maps to the spec-element.
- **Interface Report** - The interface report is a graphical diagram that shows the flow of information and control among the behavioral modules, and among the environment and the rest of the model

Compiling Generated C Code

This section describes the procedure and environment for compiling, linking, and porting compiled C code.

The Rational Statemate code generator supports the generation of both ANSI C and the traditional Kernighan & Ritchie style of C code. Select between these two styles of C when you define the Global Profile Settings in the compilation profile. Refer to **Selecting Code Parameters** for more information.

Library Location

The source files for the **Kernighan & Ritchie C libraries** are stored in `$STM_ROOT/etc/prt/c`.

The source files for **ANSI C libraries** are stored in `$STM_ROOT/etc/prt/ansic`.

The Code Generator allows a mix of different styles of C. For example, if the target language is K&R C, the Code Generator includes any subroutines implements in ANSI C, or vice versa. In these cases, in addition to `Makefile` for the target language, the Code Generator produces another file for compiling the subroutines that were implemented in a different style of C. This file is called `C_Makefile` or `ANSIC_Makefile` respectively.

Compilation Command

The default compiler statements for each of the supported Rational Statemate platforms are listed in the following table:

Sun SunOS	acc
Sun Solaris	acc
HP HPUX	cc -Aa -D_HPUX_SOURCE
Windows NT	cl

If the C compiler you are using differs from the default for your platform, then customize it by editing the `Makefile` and `User_Makefile`. These files are produced when the code is generated.

Supplementing the Rational Statemate Model with C Code

When supplementing the Rational Statemate model with handwritten C code, the additional compilation statements will be automatically added to the `User_Makefile`. This file is produced when the code is generated.

The following is an example of the `User_Makefile` for ANSI C generated code.

```
CC = acc
OBJECTS = user_activities_out.o
CFLAGS = -o -ansi -pedantic -Wstrict-prototypes
                -I$STM_ROOT/etc/prt/ansic
                -I$STM_ROOT/etc/prt/ansisched
all : out_lib.a
out_lib.a : $(objects)
                ar rvu out_lib.a $(objects)
                ranlib out_lib.a
```

Add all objects that require compiling to the elements list.

Details of Compilation and Linking

This section describes the UNIX and PC compilation environments.

UNIX Compilation Environment

The prototype executable consists of three components:

- Code generated by the Code Generator that reflects the Rational Statemate model.
- Additional user-provided source files and libraries.
- Runtime library modules (refer to the following table).

File	Description
Code Generator Intrinsics:	
libintrinsic.a	for K&R C
libaintrinsic.a	for ANSI C
Scheduler Library (not always needed):	
libscheduler.a	for real time
libsim_scheduler.a	for simulated time
Debugger Module:	
libdbg.a	needed only when using debug facilities

PC Compilation Environment

Refer to the *Rational Statemate Administrator's Guide* for the supported Windows compilation environment.

Library	Description
libdbg.lib	Debugger library
libintrinsics.lib	Intrinsics library
libscheduler.lib	Scheduler library
libsim_scheduler.lib	Scheduler for simulated-time library
libpgertl.lib	Panels run-time library

Locating Rational Statemate Libraries

The libraries for your Rational Statemate platform are pre-compiled and located in `$STM_ROOT/lib`. If you wish to compile and link your prototype on a platform that is *not* your Rational Statemate platform, you will have to compile these libraries from the provided sources in `$STM_ROOT/etc/prt/c`.

Sources for building of scheduler and sim_scheduler libraries are located in the `$STM_ROOT/etc/sched`.

For ANSI C, sources are located in the `$STM_ROOT/etc/prt/ansic` and `$STM_ROOT/etc/ansisched`.

Using `make` to Link and Compile

The compile and link phase compiles the generated code and handwritten code into a library called `out_lib.a`, and links it with the runtime modules and the user-specified libraries into an executable prototype.

Every time you modify your specification and generate code, you have to follow this procedure. The mechanism that manages this process is the `make`.

The advantage of `make` on "flat" compilations is that it can manage incremental compilation. That is, compiling only what is necessary due to the latest changes.

The input to `make` are two dependency files: `Makefile` and `User_Makefile`. They contain lists of files and dependencies that determine what has to be re-compiled after every change in the source files.

The `Makefile` lists all the generated files that should remain intact. The `User_Makefile` compiles the `user_activities` template and additional files added by the user.

Makefile Settings

You can enter the compilation command, flags, and libraries you want linked to the prototype by selecting **Options** > **Makefile Settings**. Refer to **Selecting Code Parameters** for more information.

Adding Files to the Prototype

The following is an example of the `User_Makefile` on UNIX.

```
CC = acc

OBJECTS = user_activities_out.o

CFLAGS = -o -I$$STM_ROOT/etc/prt/c\
-I$$STM_ROOT/etc/sched -DPRT

all : out_lib.a

out_lib.a : $(objects)

    ar rvu out_lib.a $(objects)
    ranlib out_lib.a
```

Assume that you wish to add a file `myfile.c` and a header `myfile.h` to the prototype. The `User_Makefile` should look like:

```
CC = acc

OBJECTS = user_activities_out.o myfile.o

CFLAGS = -o -I$$STM_ROOT/etc/prt/c\
-I$$STM_ROOT/etc/sched -DPRT

    all : out_lib.a

out_lib.a : $(objects)

    ar rvu out_lib.a $(objects)
    ranlib out_lib.a

myfile.o : myfile.h
```

The following is an example of the `User_Makefile` on a PC.

```
CC = cl
OBJECTS = user_activities.obj

CFLAGS = /nologo /MTd /W3 /Zd /Os
/I “$(STM_ROOT)\etc\prt\c”
/I “$(STM_ROOT)\etc\sched”
/D “PRT”
/D “LIB4WIN_NT”

all :     tmp_out_lib.lib
tmp_out_lib.lib : $(OBJECTS)
       lib$(OBJECTS) /OUT:tmp_out_lib.lib

user_activities.obj : user_activities.h\
       garage_c_profmain.h\
       garage_door.h
```

Executable Image

The resulting executable files are created:

- Without the Debugger option selected in the profile:
  `<profile_name>`
- With the Debugger option selected in the profile:
  `<profile_name>_dbg`

Exporting an Executable Image

To export a Rational Statemate-generated executable so it will run in a different directory or even on a different computer, you just have to copy the contents of the output directory. At a minimum, you have to copy the following files:

- **Executable image** — `<profile_name>` or `<profile_name>_dbg` depending on whether the Debugger was requested in the profile.
- If the Debugger was requested in the profile, select the following files

<profile_name>_dbg	This file contains a symbol table needed to run the code with the Debugger. If this file is not copied, you can still run the executable, but Debugger facilities will not be available to you.
help.dat	Needed only if during execution you wish to use Debugger's online help.

- If the compilation profile contains a panel, select the following files

<panel_name>.pnl	This file is an ASCII representation of the panel.
stm_color_base	Contains information on colors to be used during the panel execution.
panels_displays.dat	Needed when the profile specifies that the panel is to be displayed on a non-default terminal (for example., not the one executing the code).

Note

To run the code on an operating system other than the one it was originally created on, you must compile the source code as well as all of the Rational Statemate libraries for that particular operating system.

Building the Runtime Modules on Foreign Platforms

The runtime modules consist of four libraries:

- **libscheduler.a** - timing and multi-threading
- **libsim_scheduler.a** - timing and multi-threading for simulated time mode
- **libintrinsics.a** - double buffers and callbacks support
- **libdbg.a** - the debugger.

The sources for the libraries are located in two different directories:

- The sources for the scheduler are in `$STM_ROOT/etc/sched.`
- The sources for the intrinsics and the debugger are in `$STM_ROOT/etc/prt/c.`

Supported Platforms

If you wish to build the libraries on any of the Rational Statemate supported platforms, there are scripts that will compile and create the libraries in the source directories.

In the scheduler directory there is a script that builds the scheduler library for a supported platform. Specify the platform in the script's argument. The following example creates a shell on the Solaris platform:

```
create_sched sol
```

The intrinsics library is created with the following script, which is located in `STM_ROOT/etc/prt/c.` The following example creates an intrinsics library on the Solaris platform:

```
create_intrinsics sol
```

The debugger library is created with the following script, which is located in `STM_ROOT/etc/prt/c`. The following example creates a debugger library on the Solaris platform:

```
create_dbg sol
```

Unsupported Platforms

The scheduler library, which supports tasking (multi-threading) and timing services, is platform dependent. If you do not use tasking in your `user_activities`, which is the common case, then you have to customize the software interrupts (signals) used for the timing services.

Implementation of the Timing Control

All the timing mechanisms are implemented in the file `timer.c`. The scheduler uses the UNIX signal mechanism to implement these software interrupts. If your system is UNIX compliant, then you do not have to modify the code. If your system does not support these calls, you have to replace the `set timer` calls with other software interrupt calls available on your system.

Implementation of Tasking Services

You have to implement the tasking system if you choose to:

- Implement primitive activities as tasks
- Use environment tasks

The heart of the multi-threading system is the context switching mechanism between threads. This mechanism is not fully supported by C and involves machine-level coding. The context switching is done by the `context_switch` routine, which you have to implement.

On some systems, the C **setjmp**, **longjmp** mechanisms perform context switching (on some systems **longjmp** refuses to jump to higher stack addresses), but the initial thread setting must be done in machine code. On some systems, there are vendor supported multi-threading libraries (for example, lightweight processes on SUNOS) that provide library services for these purposes.

Adding User-Written Code

This section describes supplementing Rational Statemate-generated code with handwritten code (also called “user-written” code) You may include handwritten code as part of your Rational Statemate model, and this code become part of the generated code, as well as part of simulation.

In cases where **most** of the code you are using is handwritten or from third parties, refer to **Adding STM Code Modules.**

The Code Generator enables you to extend the Rational Statemate model by supplementing the model with handwritten code. This means that you can implement those elements and aspects of the system’s behavior that have not been explicitly defined by the controlling Statecharts and mini-specs.

You may want to use this feature to accomplish the following:

- Describe a particular function programmatically.
- Interface to your own or a third party’s library.
- Use code that already exists.

There are several ways to supplement the generated code:

- Attach existing code to the model through the Properties Editor and select one or more languages in which to implement it (K&R C, ANSI C, or Ada).
- Write new code directly in Rational Statemate using the Rational Statemate Action Language.
- Use a graphic to define a function or procedure in a Procedural Statechart.
- Create a Truth Table to implement a subroutine, define a “named action,” or describe an activity’s behavior.

These methods enable you to add code that is used by both the Simulator and the Code Generator. Rational Statemate stores the code in the model’s database and automatically includes it when you run simulation or code generation.

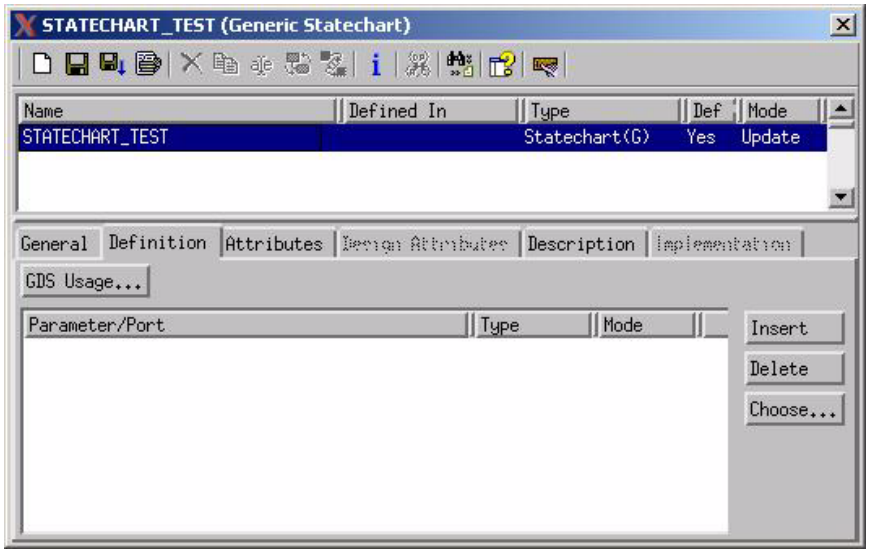
Supplementing the Model with Subroutines

The following sections explain how to add handwritten *subroutines* (functions, procedures, or tasks) to your Rational Statemate model.

The method for adding all three subroutines in the Properties Editor is similar. The major difference is that functions require a Return Type.

Note

In addition to storing subroutines in the Properties Editor, you can also store their formal parameters.



Entering Handwritten Code

Rational Statemate does not check your handwritten code. It is your responsibility to ensure that the code is legal and compilable. You can use **`with`**, **`use`**, **`include`** statements or any other mechanism supported by the language to reference packages or include files. Rational Statemate makes no attempt to interpret the code; it merely passes it on to the appropriate compiler.

To add your handwritten code to the template correctly, make sure you abide by the rules in the following sections:

- Referencing model elements in the code (Refer to **Referencing Model Elements**).
- Mapping Rational Statemate types (primitive or user-defined) into C types for variables and subroutine parameters (Refer to **Mapping Rational Statemate Types into C**).
- Using synchronization services in tasks (Refer to **Synchronizing Tasks**).

Using Subroutines

After you define a subroutine in the Properties, it becomes part of Rational Statemate and is stored as part of the model. Then, you can use the subroutine in the following ways:

- Called in Rational Statemate actions and expressions.
- Bound to a primitive activity of the modeled system, thus providing their implementation.
- Bound to an external activity to describe behavior of the environment.
- Bound as a callback to a textual or graphical element in the model, and called when the element changes its value or status.

Disabling Subroutines

To disable a subroutine, open the Properties Editor and select **Select Implementation > None**.

Rational Statemate does not implement the subroutine, and only generates a template (empty stub).

Supplementing the Model with a Procedure

This section explains how to add a handwritten procedure to your Rational Statemate model by showing the following:

- Windows and how to complete them
- Template that Rational Statemate produces
- Template filled in with an example of handwritten code

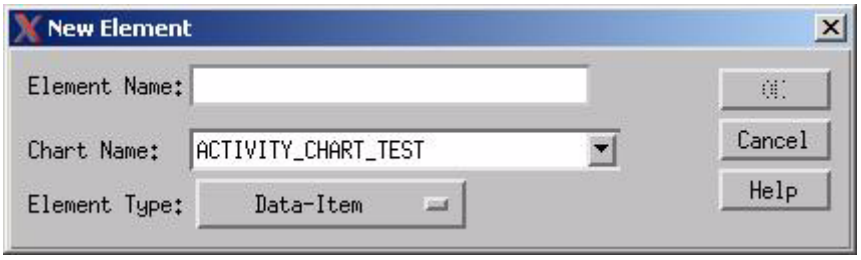
Note

Rational Statemate also provides templates for functions and tasks. The subroutine's template is a result of mapping the declarations into its C representation. This includes mapping the parameter types and, in the case of functions, the returned value.

Complete the following steps to add a handwritten procedure:

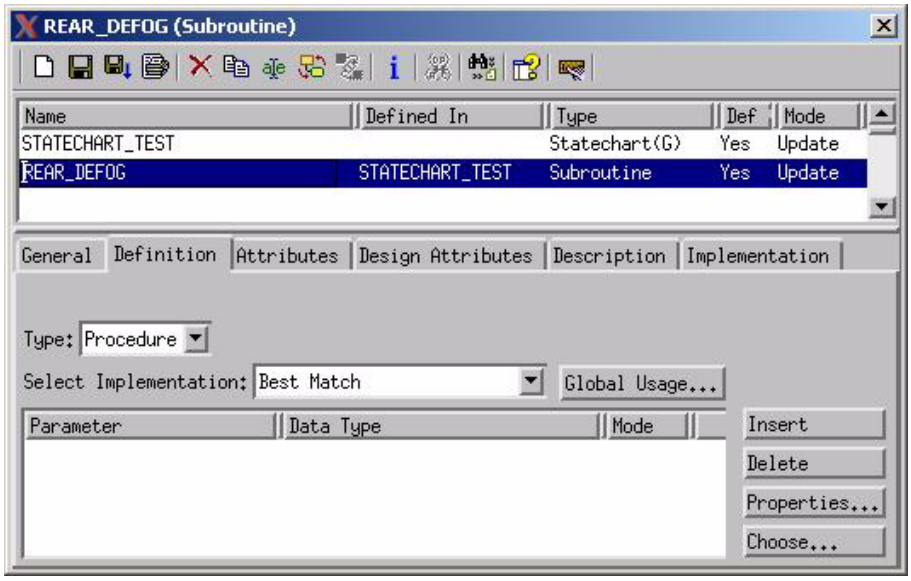
1. Click  in the Properties Editor.

   The New Element window opens.



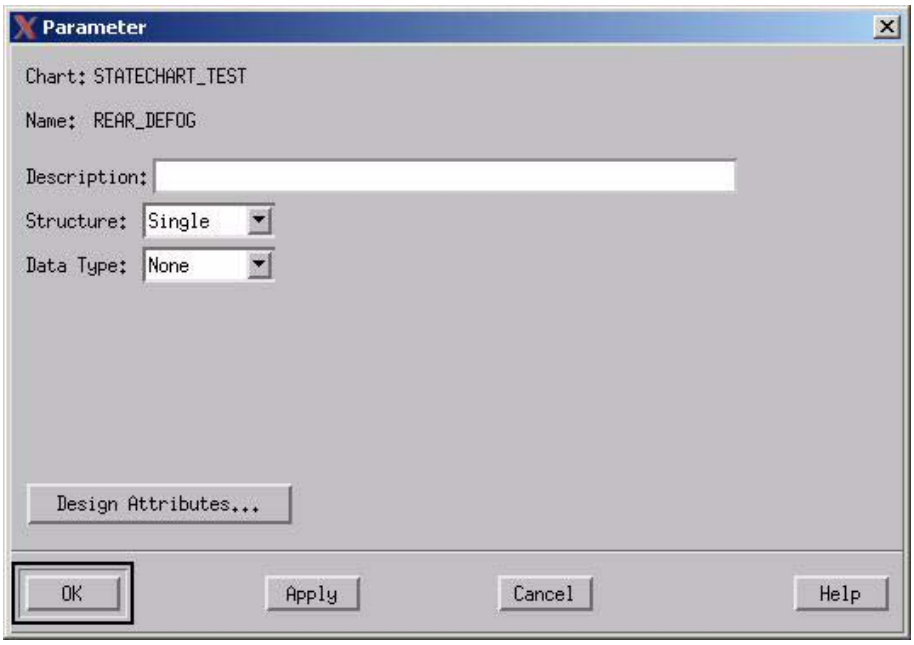
2. Enter the name of the new element.
3. Select its Chart Name.
4. Select **Subroutine** as the Element Type.

   The Properties Editor appears with the name of the new subroutine.



5. Define the Type as a **Procedure**.
6. Enter the procedure's parameters if you want to store them.
7. Select a parameter and click **Properties.**

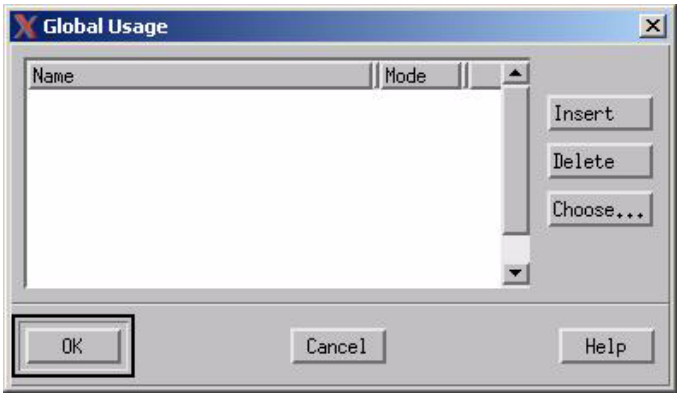
   The Parameter window opens.



Using Globals

If you wish to use the same parameters for multiple activities, you may want to define them as globals. If so, click **Globals Usage**.

The global Usage window displays:



Globals are elements that are external to the subroutine, but are not listed as parameters. The reading or writing of global data is called a *side effect*.

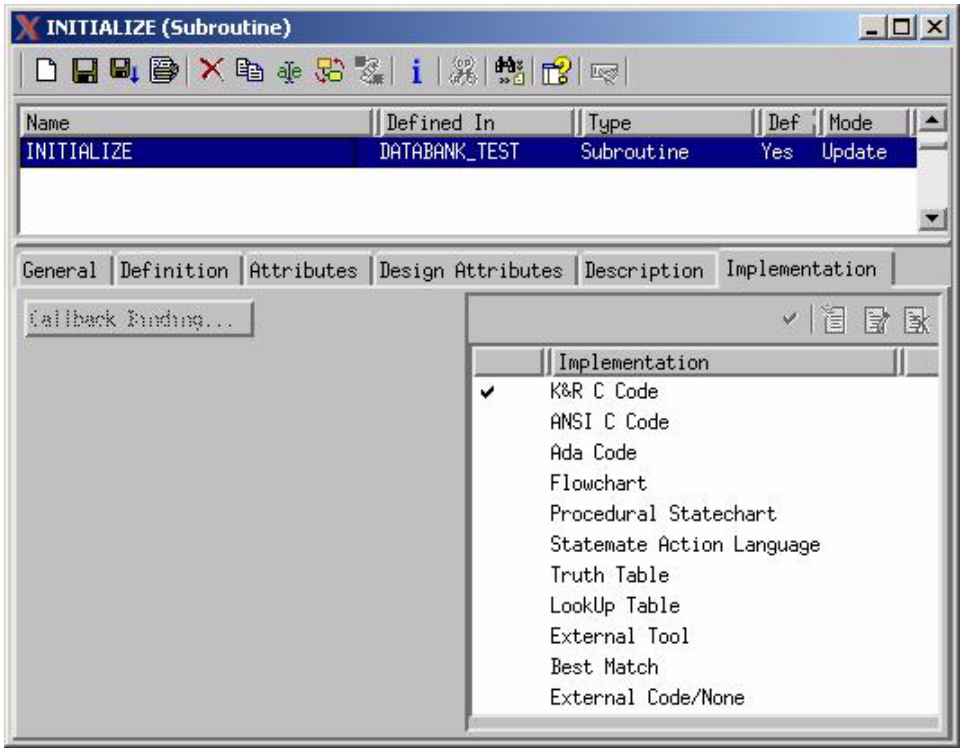
Writing more than once to a global element is considered *racing*. However, this racing differs from general racing where you have no way of determining which value will be assigned. In this case, the final value will be the resulting value of the global element. Therefore, it is your responsibility to ensure that the subroutine writes to global elements only a single time during its execution.

Note

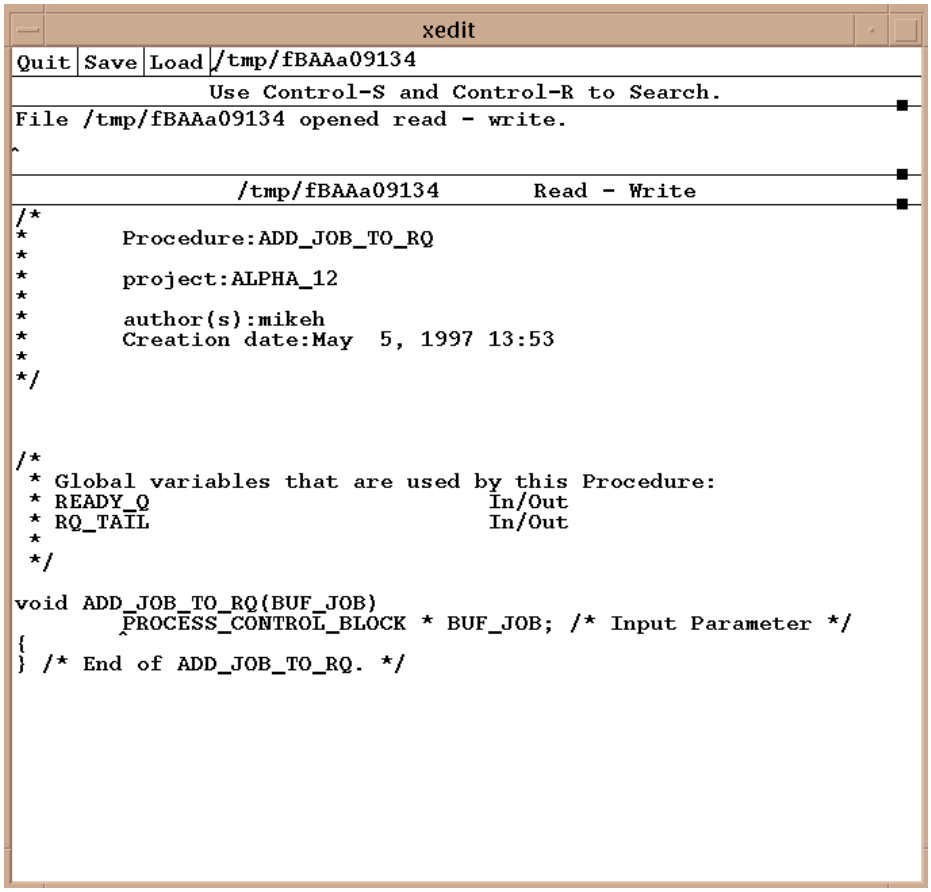
It is strongly recommended that you do not write global data in a function called in a trigger expression. Side effects written as part of a trigger will behave differently between simulation and code.

Producing a Template for a Procedure

To produce a template for a procedure, open the **Implementation** menu to select a language for the code. This example uses K&R C.

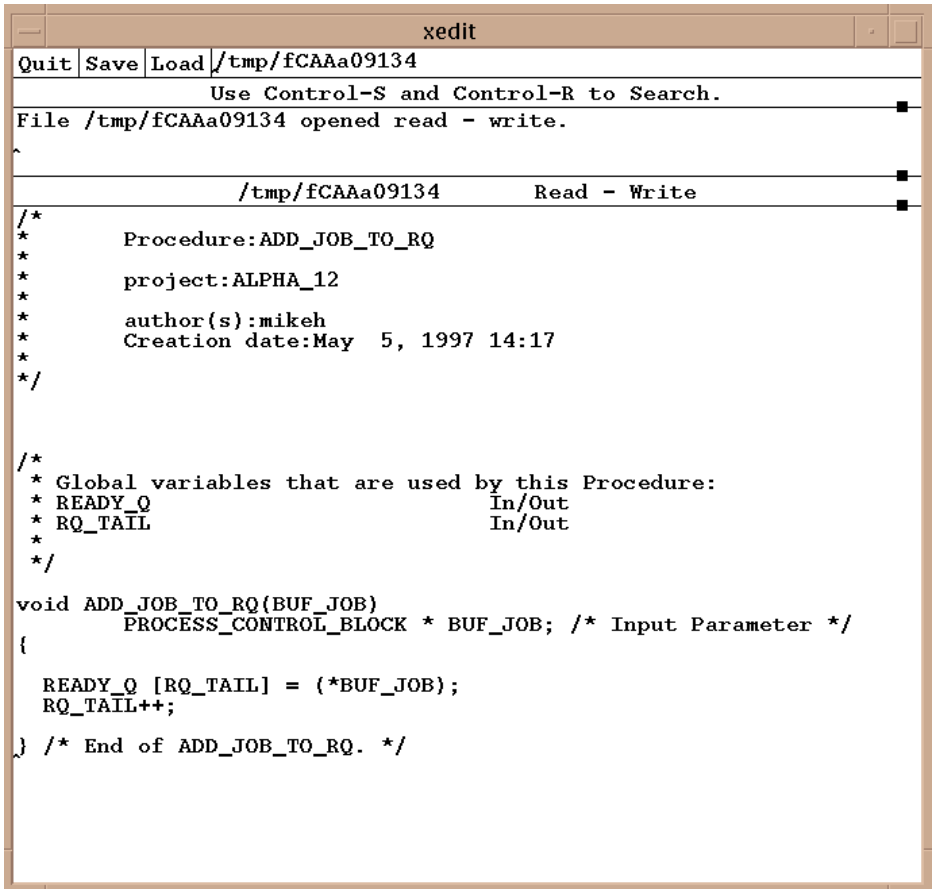


Rational Statemate opens an editor and provides a template for you to attach your handwritten code (refer to the following figure).



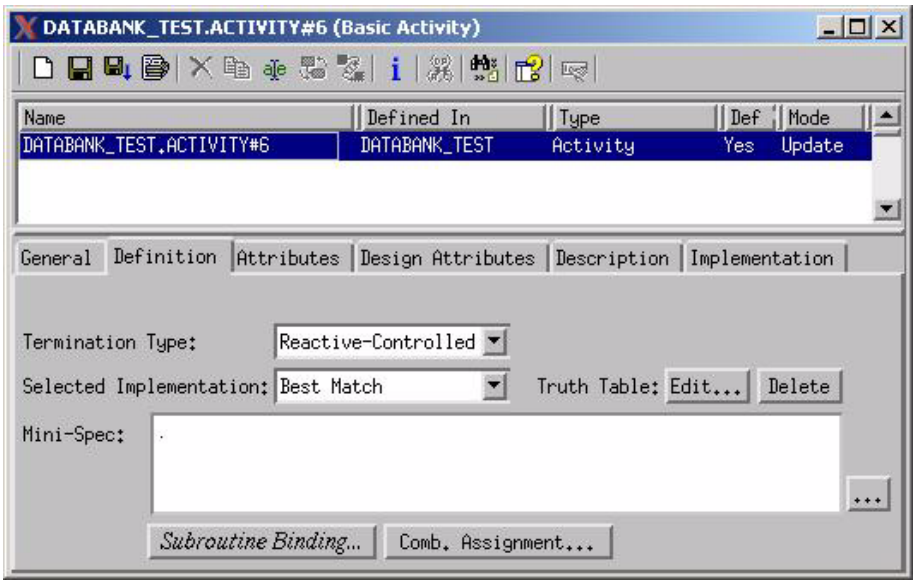
Filling in the Procedure's Template

The following example shows the template filled in with handwritten code for a complete procedure.

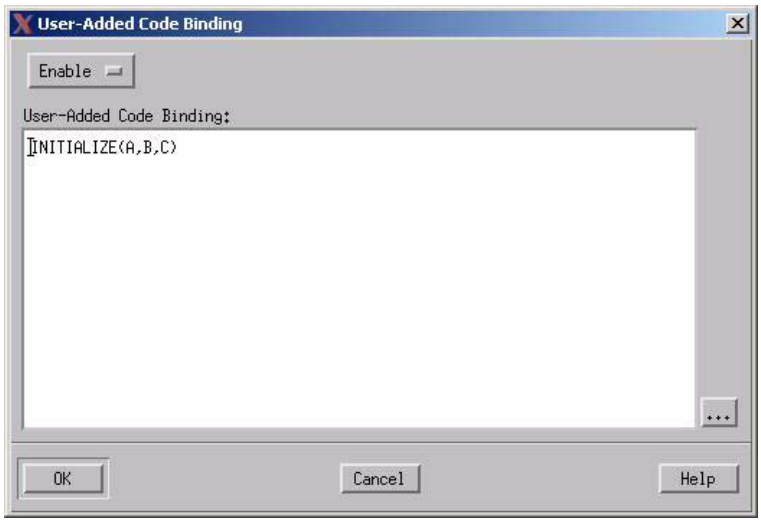


Subroutine Binding

To connect subroutines, open the Properties Editor for an activity and click **Subroutine Binding** (refer to the following figure).



The **User-Added Code Binding** window opens when you enter the name of the subroutine, which is to be bound to the activity (refer to the following figure).



Supplementing the Model with a Task

This section explains how to add a handwritten task to your Rational Statemate model by showing the following:

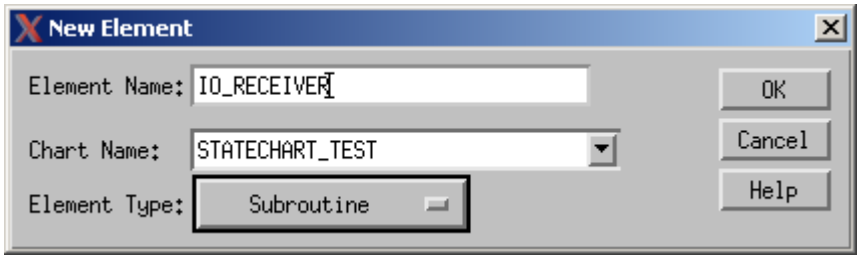
- Windows and how to complete them
- Template that the Code Generator produces
- Template filled in with an example of handwritten code

Note

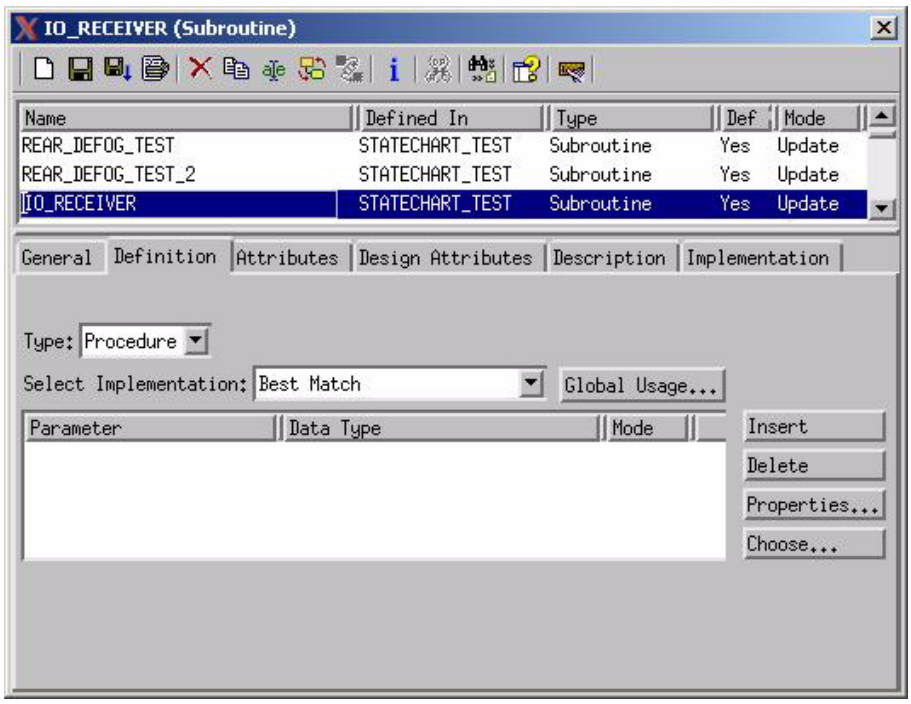
Rational Statemate also provides templates for functions and procedures. The subroutine's template is a result of mapping the declarations into its C representation. This includes mapping the parameter types and, in the case of functions, the returned value.

Complete the following steps to add a handwritten task:

1. Select **File > New** in the Properties Editor.
2. Name the new element (in this example **IO_RECEIVER**), then select its Chart Name.
3. Select **Subroutine** as the Element Type. The New Element window opens.

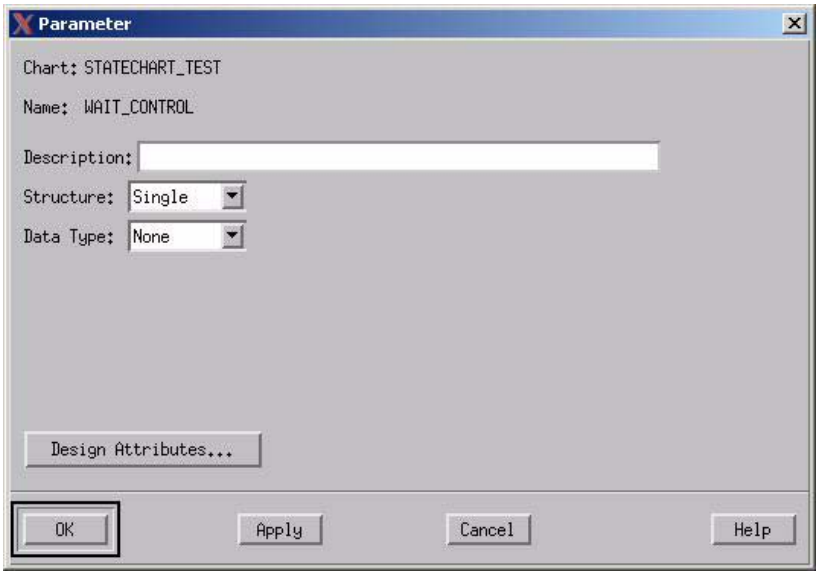


The Properties Editor window opens with the name of the new subroutine.



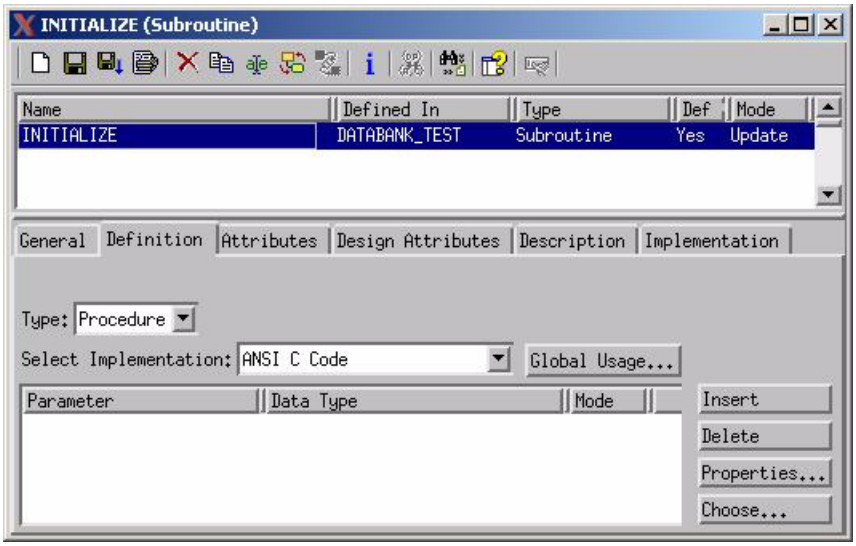
4. Define the subroutine Type as a **Task** (refer to the following figure).

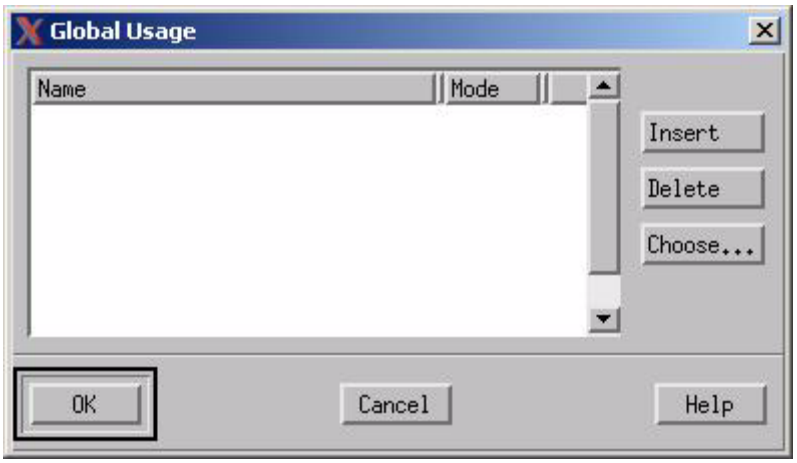
5. Enter the task's parameters if you want to store them in the . Select a parameter and click on **Properties** to display the Parameter window..



Using Globals

If you use the same parameters for multiple activities, you may want to define them as globals. If so, click **Globals Usage**.





Globals are elements that are external to the subroutine, but are not listed as parameters. The reading or writing of global data is called a *side effect*.

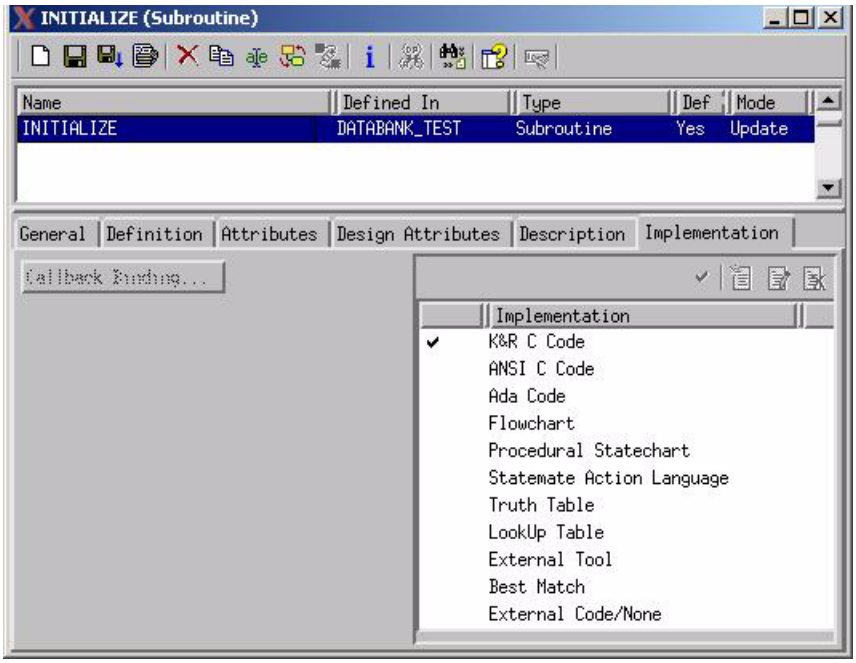
Writing more than once to a global element is considered *racing*. However, this racing differs from general racing where you have no way of determining which value will be assigned. In this case, the final value will be the resulting value of the global element. Therefore, it is your responsibility to ensure that the subroutine writes to global elements only a single time during its execution.

Note

It is strongly recommended that you do not write global data in a function called in a trigger expression. Side effects written as part of a trigger will behave differently between simulation and code.

Using the Template for a Task

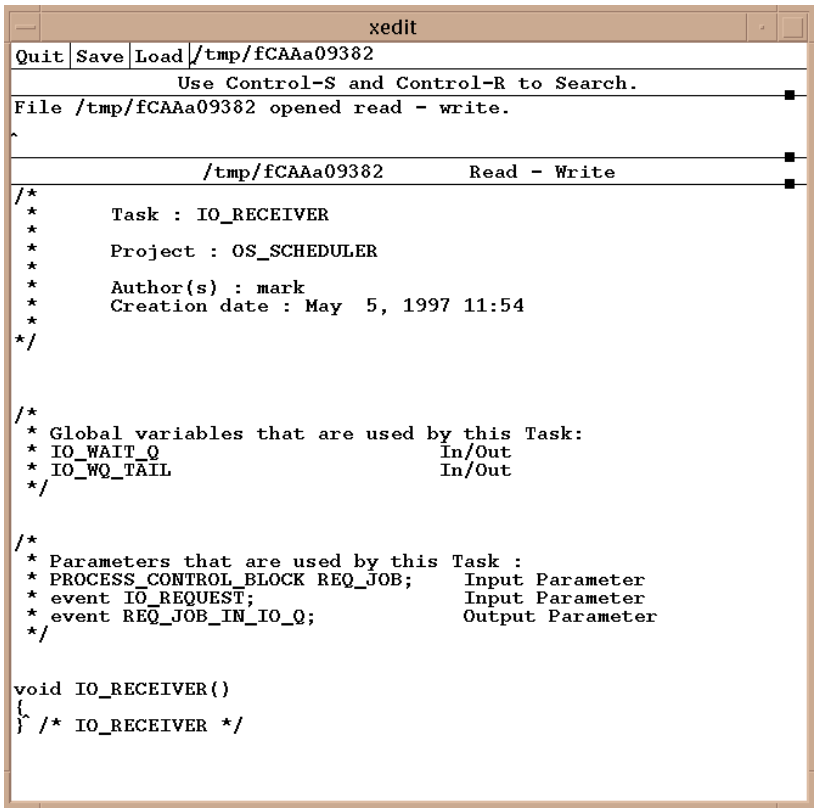
Select the Implementation tab to select a language for the code. Each available option is listed in the menu as shown in the following figure.



Note

This example uses K&R C.

Rational Statemate opens an editor and provides a template for your use to attach your handwritten code as shown in the following figure.

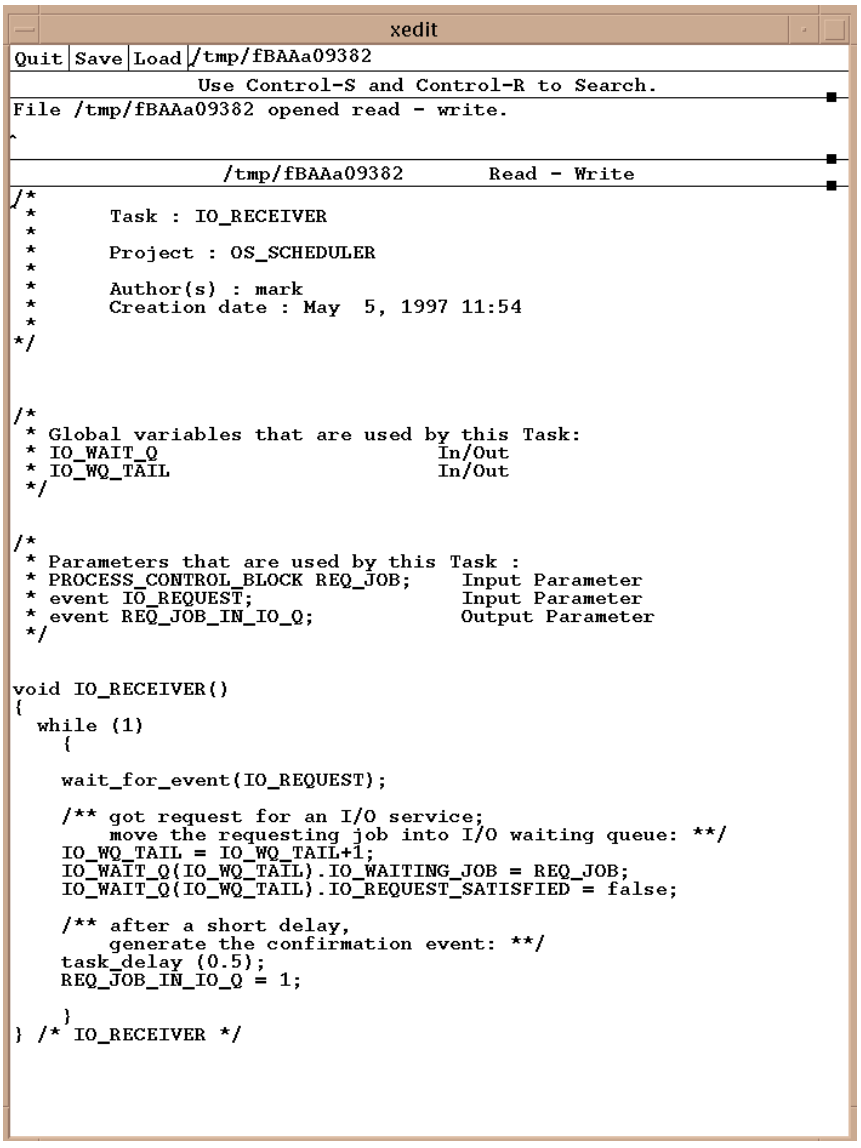

xedit
Quit
Save
Load
/tmp/fCAAa09382
Use Control-S and Control-R to Search.
File /tmp/fCAAa09382 opened read - write.
/tmp/fCAAa09382 Read - Write
/*
* Task : IO_RECEIVER
*
* Project : OS_SCHEDULER
*
* Author(s) : mark
* Creation date : May 5, 1997 11:54
*
*/
/*
* Global variables that are used by this Task:
* IO_WAIT_Q In/Out
* IO_WQ_TAIL In/Out
*/
/*
* Parameters that are used by this Task :
* PROCESS_CONTROL_BLOCK REQ_JOB; Input Parameter
* event IO_REQUEST; Input Parameter
* event REQ_JOB_IN_IO_Q; Output Parameter
*/
void IO_RECEIVER()
{
} /* IO_RECEIVER */

Filling in the Task's Template

The following figure shows an example of the template filled in with handwritten code for a complete task.

Note

The edited template must be saved when completed.



Synchronizing Tasks

This section discusses how primitive activities are integrated into the generated code.

User-written procedures are called when the system starts the corresponding activity (i.e., *st!(<activity>)*). In general, the user code and the generated code share the CPU time. That is, when the user code is executed, the Statechart's code (or other user activities) are suspended.

Tasks

The task mechanism allows you to integrate continuous or synchronized code into the primitive activity. For this purpose, the Code Generator provides a special library that extends the C language to support *tasking* or *multi-threading*. (Refer to **Scheduler Package**, for details). Tasks can be bound to either a primitive or an external activity.

The scheduler package allows you to define C functions as *concurrent routines* or *co-routines*. An activity that you choose to implement as a task is started by the control code as a co-routine, which is executed concurrently with the rest of the prototype. Since we are dealing with serial machines, concurrency means that the control is switched between these co-routines without interrupting their thread of control. That is, when the co-routine gets the control back, it resumes executing with the exact context it was before.

This mechanism allows the activity to use delay statements, wait for events, and perform continuous calculations without blocking the rest of the code from continuing execution. When a task is executed, however, the rest of the code is frozen. Thus, synchronization points are introduced. They allow the rescheduling of other tasks (or the control code) to proceed and actions (stop, suspend) to take effect.

Synchronization

There are three types of synchronization calls:

- wait_for_event(event)
- task_delay(delay_time)
- scheduler()

Each of these calls will suspend the calling task and reschedule another task or the `main_task` (statechart) on a round-robin basis.

The `wait_for_event` call suspends the activity until the specified event is generated. It is a way to synchronize the activity with other activities either user-implemented or statechart-controlled. When the event is generated, the code resumes execution after the wait call.

Example:

```
void sense_start()
{
  while (1) {
       wait_for_event(SENSE);
    /* here you are supposed to check status.*/
   printf("Time generated\n");
  }
} /* end sense_start */
```

The `task_delay` statement delays the activity for the time specified in the call. It is useful to implement polling processes that periodically perform checks on a time basis.

Example:

```
void poll_input()
{
   while (1) {
      mouse_input = read_input_from_mouse();
      if (mouse_input) {
      . . Do Something . . .
      }
      task_delay(0.1); /* delay 0.1 seconds */
   }
}
```

The `scheduler()` call is used when you have a calculation which is too long to be executed non-preemptively. For example, if you have to multiply two 10000x10000 matrices, you do not want the rest of the system to be blocked all that time.

The `scheduler()` call will allow other activities to proceed and the calling activity will resume execution in the next available time slot unless a stop or suspend command was issued. The call should be placed in a loop in which one cycle can be executed without preemption, but an outer loop may take too long.

Note

No synchronization call should be used by a procedure-implemented activity.

Example:

```
void multiply()
{
  for (i = 1; i<=10000; i++) {
    for (j = 1; j<=10000; j++) {
      /* internal loop is short
         enough to complete */
    }
    scheduler();
  }
}
```

Scheduler Package

The user can specify that some of the primitive activities are to be implemented as tasks in the Profile Editor. The tasks are actually C functions started as co-routines. The Statechart code itself is a task, which runs concurrently with the other running tasks.

Controlling all those tasks is the responsibility of statecharts, which issue different actions to the different activities (i.e., start, stop, suspend, resume). All this is handled by a scheduler package, which is supplied with the Code Generator and is available on Rational Statemate platforms only. This package supports multi-tasking programming within the context of a single process.

Below we describe how the user may add his own tasks, apart from those created for each task-like primitive activity, and how to use the scheduler for controlling them.

Status of a Task

Each task may be in one of four states:

- **Current** - The task is executing
- **Ready** - The task is ready for execution
- **Delayed** - The task is waiting for some event to occur
- **Stopped** - The task is not active

The calls that change the status of a task are described in the following section.

Scheduling Policy

The context switch between tasks is done only in the following synchronization points:

- When a task explicitly calls the scheduler. This is done by calling the following routine:

  ```
  scheduler()
  ```

- If there are other ready tasks - one of them (chosen in a round-robin manner) becomes current, while the calling task becomes ready. If there is no other task ready, the calling task continues its execution.
  - When a task issues a delay request by calling `task_delay`. The calling task then becomes delayed.
  - When a task calls a `wait_for_event` service. The calling task then becomes delayed.

    ```
    wait_for_event(EVENT)
        event *EVENT;
    ```

- After the task function performs a return, it stops.

Restrictions

Any call to process blocking functions (for example, sleep, scanf) of the operating system from a task will hibernate not only the calling task, but the whole process. Using `fork()` and signals is also not allowed, since it might confuse the scheduler.

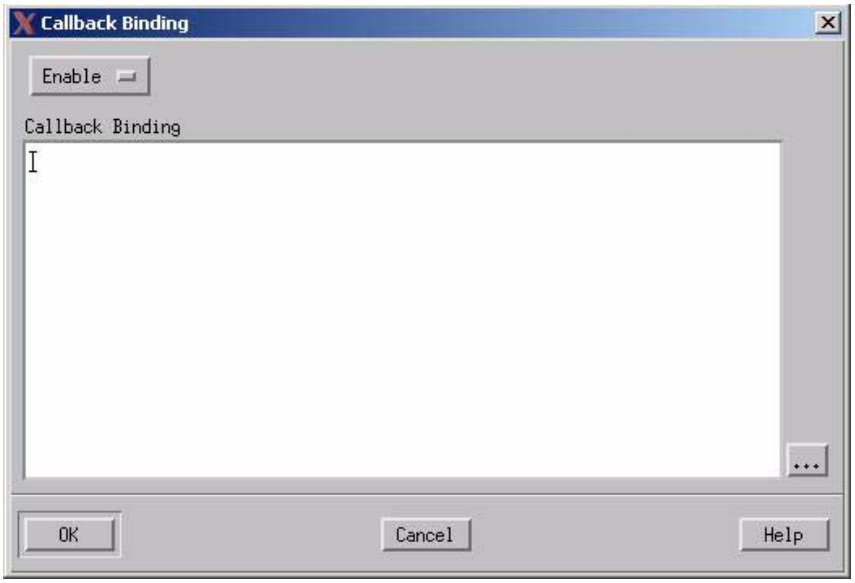
Binding Callbacks

Callbacks are a powerful mechanism that enable you to connect user-actions or procedures to any change in a Rational Statemate element during execution. This mechanism is very useful when you wish to tie your external environment to the behavior represented by the generated code.

Callback Binding

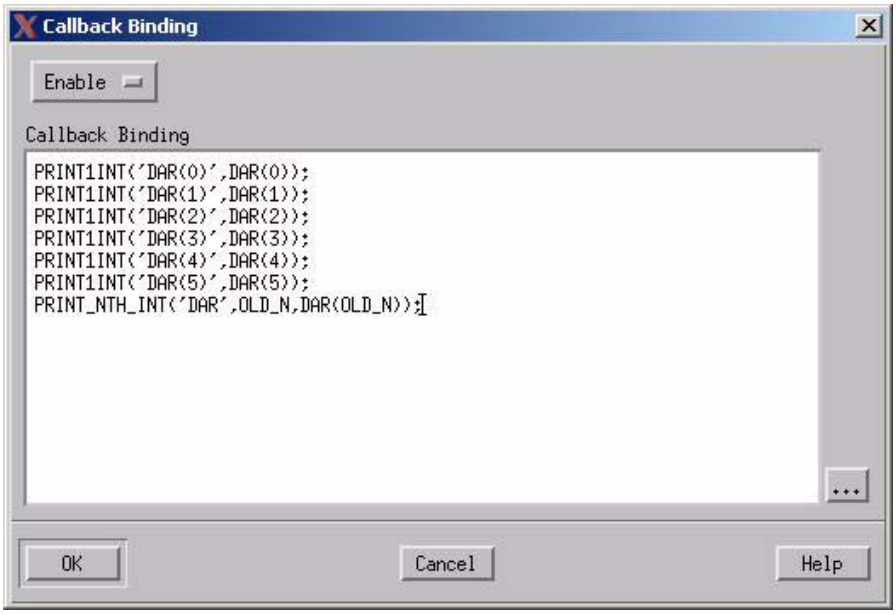
To connect elements such as events, conditions, data items, and user-defined types, use the following procedure:

1. Select the element in the Properties Editor
2. Select the **Implementation > Callback Binding** .



Note

The following figure is an example of a Callback Binding



Callback Statement

The connection and binding statement syntax for callbacks consists of:

```
proc_name(<"element_identifier">,param_1,param_2)
```

Where the `<element_identifier>` is required **when and only when** the callback is connected to an aggregate element. An aggregate element is an array, record, union, user-defined type, or any element referenced in a generic or instance. The <element_identifier> specifies what part of the aggregate element the callback is to be connected.

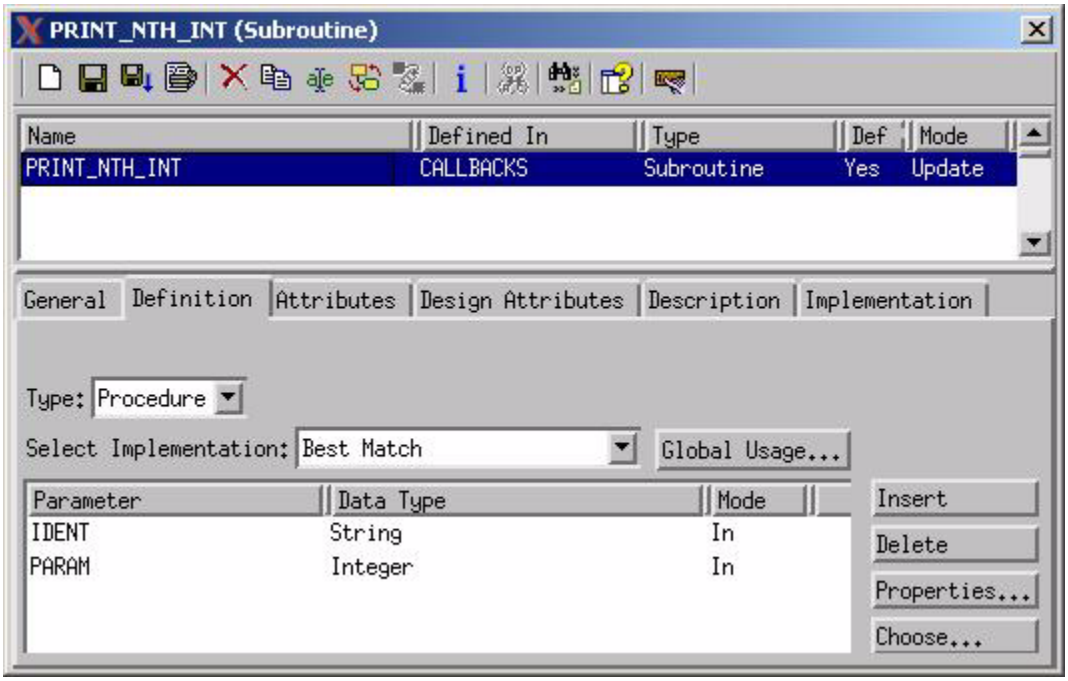
Disabling Callbacks

To disable a callback, change the **Enable** option in the **Callback Binding** dialog to **Disable**. This causes the Code Generator to generate code, but it "breaks" the code's connection with the element.

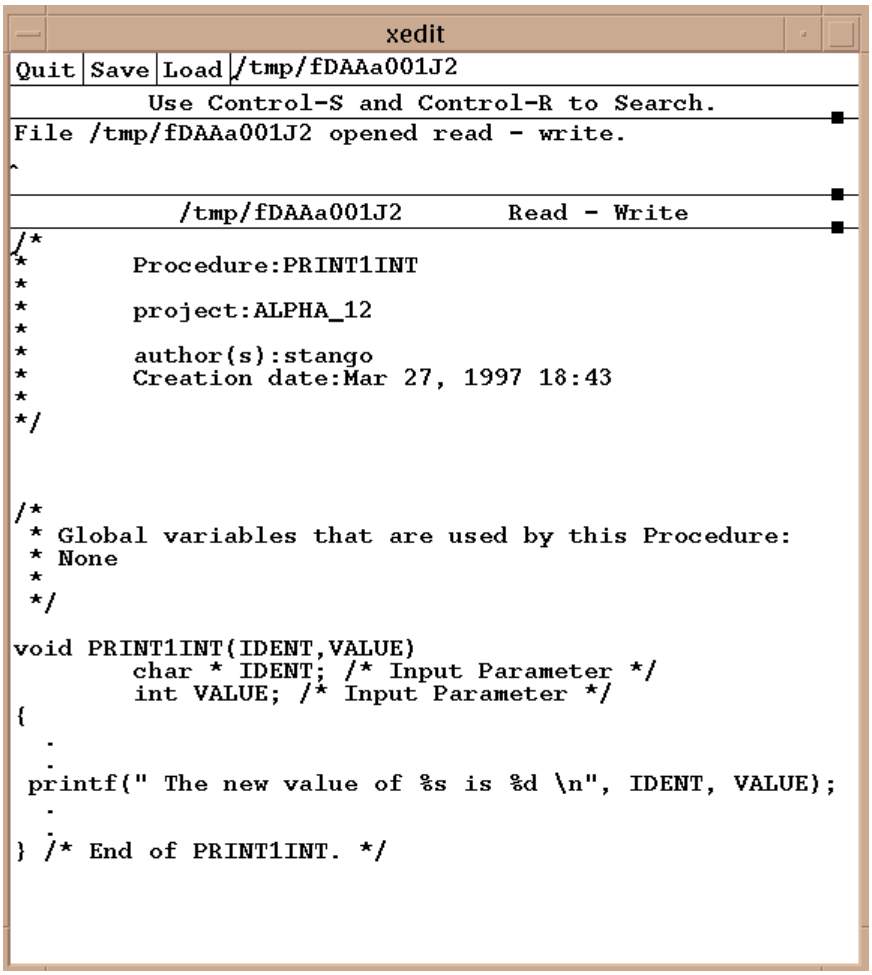
Callback Example

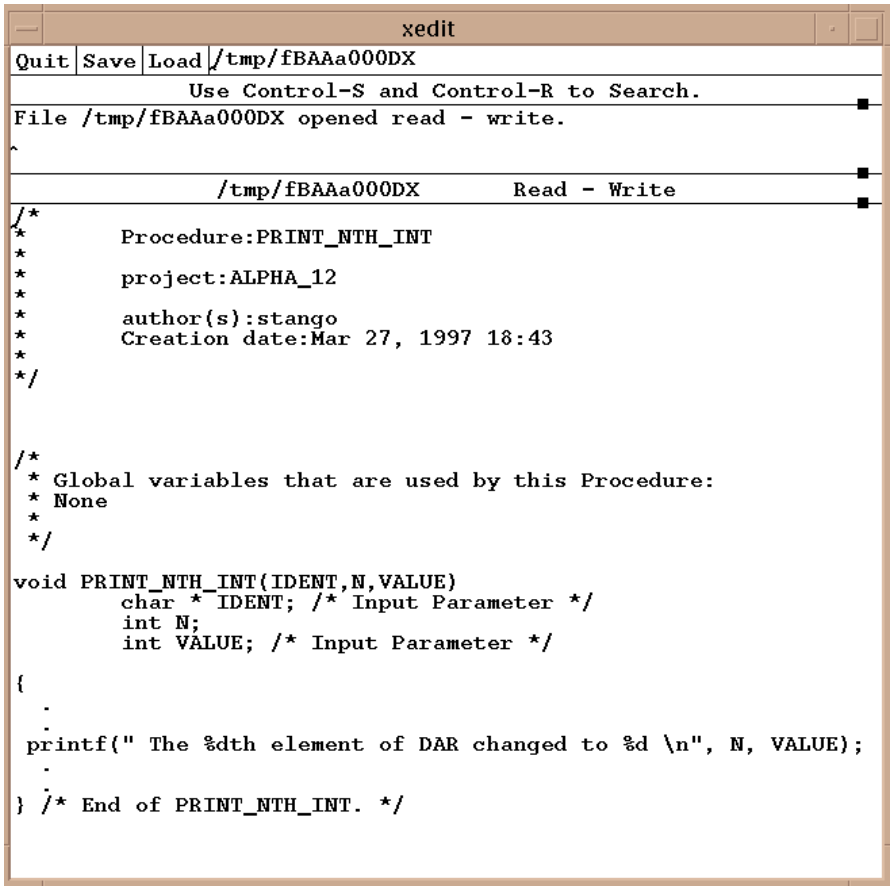
The following example illustrates the Rational Statemate **callback** utility. It shows two subroutines that are bound to the callback DAR. Every time the DAR element changes, Rational Statemate executes both of these subroutines.

To create a subroutine, start with the steps shown in **Supplementing the Model with Subroutines.**



The following figures show the code for the subroutines. The first one is the `PRINT1INT` procedure; the second one is the `PRINT_NTH_INT` procedure.


xedit
Quit
Save
Load
/tmp/fDAAa001J2
Use Control-S and Control-R to Search.
File /tmp/fDAAa001J2 opened read - write.
/tmp/fDAAa001J2 Read - Write
/*
* Procedure:PRINT1INT
*
* project:ALPHA_12
*
* author(s):stango
* Creation date:Mar 27, 1997 18:43
*
*/
/*
* Global variables that are used by this Procedure:
* None
*
*/
void PRINT1INT(IDENT,VALUE)
char * IDENT; /* Input Parameter */
int VALUE; /* Input Parameter */
{
.
.
printf(" The new value of %s is %d \n", IDENT, VALUE);
.
.
} /* End of PRINT1INT. */


xedit
Quit
Save
Load
/tmp/fBAAa000DX
Use Control-S and Control-R to Search.
File /tmp/fBAAa000DX opened read - write.
/tmp/fBAAa000DX Read - Write
/*
* Procedure:PRINT_NTH_INT
*
* project:ALPHA_12
*
* author(s):stango
* Creation date:Mar 27, 1997 18:43
*
*/
/*
* Global variables that are used by this Procedure:
* None
*
*/
void PRINT_NTH_INT(IDENT,N,VALUE)
char * IDENT; /* Input Parameter */
int N;
int VALUE; /* Input Parameter */
{
.
.
printf(" The %dth element of DAR changed to %d \n", N, VALUE);
.
.
} /* End of PRINT_NTH_INT. */

Referencing Model Elements

Communication between the handwritten code and the generated code is accomplished through the semantics of the following information elements:

- Events
- Conditions
- Data-items
- User-defined types

It is important to understand how to access the values of these elements and how to modify them. Each element has the following representation in the C target language:

- Conditions are represented as bytes
- Data-items are represented as integers, reals, strings or unsigned
- User-defined types are derived from primitive data-types

When you wish to pass structured elements (such as records and unions) from Rational Statemate to your handwritten code, you must define these elements as user-defined types.

When you write code in the template, refer to all elements by the names you assigned in the model. This applies to parameters of the subroutine, its local and global variables, to names of types, constants, and any other subroutines that you may use for the implementation.

Note

Write all *element names* in uppercase.

Referencing Events

Events are primitive elements and are special in the sense that software languages do not support them directly. Events are not allowed in subroutines as inputs, outputs, local variables, or accessible as global elements.

Events, in relation to handwritten code, are used in the following manner:

- **Callbacks** - You can associate a callback with a Rational Statemate event.
- **Tasks** - You can use the `wait_for_event` command to react to a Rational Statemate event.

Where Elements are Defined

An element can be local to a module or global to a profile. The element is globally defined when it is referenced by more than one module, for example, defined in the top-level module. Each module "exports" all its local elements as externals in its header file.

This allows other user modules to access them. If you want to reference an element you must refer to its scope by including the appropriate header file. An example is shown below.

Example:

If you want to reference (for example) an element BAUD_RATE in module *display*, you should include the header file "display.h" to make the element visible.

```
/* my module */
#include "display.h"
.
.
br = BAUD_RATE ;
.
.
```

Accessing an Element Value

Since the element is a *simple language element*, it can be easily accessed by referring to its name.

Example:

```
my_data = XXX + YYY ;
```

Mapping Rational Statemate Types into C

The following table shows how Rational Statemate maps primitive types into corresponding C types:

Rational Statemate Types	C Type
Conditions	char (byte 0-false, 1-true)
Integer	int
Real	double
Bit	bit_array[1]
Bit array	unsigned int
User Type	struct
Record	struct
Union	struct
Enumerated Types	typedef

Note

All Rational Statemate elements of type *string* are translated into allocated C elements.

Records	Records become C constructs. For example, a record `INVOICE_TYPE` might become a structure defined as:
	`typedef struct INVOICE_TYPE {`
	`char NAME[80+1];`
	`char ITEM[80+1];`
	`real AMOUNT;`
	`  } INVOICE_TYPE;`
	Note that the name `INVOICE_TYPE` is normally named the same as the User-Defined Type name. If, however, the Rational Statemate model contains multiple textual elements with the same name, the C code names will be modified to make all the names unique. This name mapping information is listed in the *.info* file.

Unions	Unions become C unions with a declaration that is similar to the construct definition for records.
Arrays	Elements of all arrays in C are enumerated starting from 0. In Rational Statemate, there is no such restriction.

Enumerated Types	An Enumerated Type is a user-defined type with a finite number of values. Enumerated values and other textual items cannot have the same name within the same scope. For example, data-item SUN cannot be declared in the same chart where an enumerated value SUN is declared. Enumerated range and indices of arrays are not supported in C. The C code generator shall approximate this capability in the generated code. There are two constant operators and five general operators for enumerated types:	
Constant Operators	`en_first(T)` `en_last(T)`	First enumerated value of T Last enumerated value of T
	Parameters to these constant operators are user-defined types that were defined as enumerated types.	

General Operators	`en_succ([T']VAL)`
	`en_pred([T']VAL)`
	`en_ordinal([T']VAL)`
	`en_value(T,I)`
	`en_image([T']VAL)`
		Successor enumerated value of T
		Predecessor enumerated value of T
		Ordinal position of VAL in T
		Value of the i'th element in T
		String representation of VAL in T

	Parameters to these operators are either enumerated values (literals) or variables. The T'VAL notation is used for non-unique literals.	
Bit Arrays	Bit-arrays are stored in unsigned ints. Since unsigned ints can hold a maximum of 32 bits, bit-arrays larger than 32 bits are stored in arrays of unsigned ints. Arrays of bit-arrays are stored in two dimensional arrays of unsigned ints. Notice that multiple bit-arrays smaller than 32 bits are NOT packed into the unsigned int.	

Data-Items*	**Results in these structures**
BA1 is array 1 to 10 of Bit-array 31 to 0	bit_array BA1[10][1]
BA2 is array 1 to 10 of Bit-array 48 to 0	bit_array BA2[10][2]
BA3 is array 1 to 10 of Bit-array 3 to 0	bit_array BA3[10][1]
* In `$STM_ROOT/etc/prt/c/types.h` you will find the statement: `typedef unsigned int bit_array`	

Bit-Array Functions

```
bit_array *AND(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;
  bit_array *ba2;
  int l_ba2;
  int from2;
  int to2;


bit_array  *NOT (ba1, l_ba1, from1, to1)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;


bit_array  *OR(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;
  bit_array *ba2;
  int l_ba2;
  int from2;
  int to2;


bit_array *XOR(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
  bit_array *ba1;
  int l_ba1;
  int from1;
  int to1;
  bit_array *ba2;
  int l_ba2;
  int from2;
  int to2;
```

The following bit array function names are mapped through macros to their internal names, because these names are used by Ada runtime libraries, therefore they cannot be defined as functions in the intrinsics. (These same intrinsics are used by C and Ada environment.) It is important to include the *types.h* header containing these macros.

```
#define ASHR ashr
#define LSHL lshl
#define LSHR lshr
#define BITS_OF bits_of
#define CONCAT_BA concat_ba
#define EXPAND_BIT expand_bit
#define SIGNED signed_b
#define MINUS minus_b
#define NAND nand_b
#define NOR  nor_b
#define NXOR nxor
```

The functions are:

```
bit_array *concat_ba
(ba1,l_ba1, from1, to1, ba2, l_ba2, from2,to2)
   bit_array *ba1;

   int l_ba1;
   int from1;
   int to1;
   bit_array *ba2;
   int l_ba2;
   int from2;
   int to2;

bit_array *lshr(ba, len_ba, from, to, shift)
   bit_array *ba;
   int len_ba;
   int from;

   int to;
   int shift;


bit_array *lshl(ba, len_ba, from, to, shift)
   bit_array *ba;
   int len_ba;
   int from;
   int to;
   int shift;


int signed_b(ba_val, len, from, to)
   bit_array *ba_val;
   int len;
   int from;
   int to;


bit_array *ashr(ba, len_ba, from, to, shift)
   bit_array *ba;
```

```
    int len_ba;
    int from;
    int to;
    int shift;


bit_array  *nand_b(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
    bit_array *ba1;
    int l_ba1;
    int from1;
    int to1;
    bit_array *ba2;
    int l_ba2;
    int from2;
    int to2;


bit_array  *nor_b(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
    bit_array *ba1;
    int l_ba1;
    int from1;
    int to1;
    bit_array *ba2;
    int l_ba2;
    int from2;
    int to2;


bit_array *nxor(ba1, l_ba1, from1, to1, ba2, l_ba2,
from2, to2)
    bit_array *ba1;
    int l_ba1;
    int from1;
    int to1;
    bit_array *ba2;

    int l_ba2;
    int from2;
    int to2;
```

Use the following functions to convert between integer and bit-array types:

```
bit_array *int2ba(int_val)
    int int_val;


int ba2int(ba, len, from, to)
    bit_array *ba;
    int len;
    int from;
    int to;
```

Rules for Mapping into C

The following table summarizes the rules of mapping into C for:

- Types of parameters for procedures and functions
- Returned type of functions

Note

- The first level of all arrays should be defined as `User-defined type` in order to restrict the “second” dimension.
- Unrestricted strings and bit-arrays are not allowed as returned type of a function.
- Numeric Input parameters can be mixed up i.e., integer, real and bit-arrays can be mixed when used as actual and formal parameters.

Type	Function Type	In Param	Out/InOut Param
Primitive (*)	int f();	int P;	int *P;
UDT defined as Primitive	UDT f();	UDT P;	UDT *P;
Record/Union	rec *f();	REC *P;	REC *P;
String	char *f();	char *P;	char *P;
UDT defined as String	char *f();	UDT P;	UDT P;
Bit	BIT_ARRAY *f();	BIT_ARRAY *P;	BIT_ARRAY *P;
Bit-array	BIT_ARRAY *f();	BIT_ARRAY *P;	BIT_ARRAY *P;
UDT defined as Bit-array	BIT_ARRAY *f();	BIT_ARRAY *P;	UDT *P;
UDT Array of Primitive	int *f();	UDT P;	UDT P;
UDT Array of String	-- *Illegal* --	UDT P;	UDT P;
UDT Array of Bit-array	-- *Illegal* --	UDT P;	UDT P;
UDT array of direct R/U	**-- *Illegal* --**	UDT P;	UDT P;
UDT array of UDT2	UDT2 *f();	UDT P;	UDT P;
Array of Primitive	**-- *Illegal* --**	int *P;	int *P;
Array of Record/Union	**-- *Illegal* --**	**-- *Illegal* --**	**-- *Illegal* --**
Array of String	**-- *Illegal* --**	char *P;	char *P;
Array of Bit-array	**-- *Illegal* --**	BIT_ARRAY *P;	BIT_ARRAY *P;
(*) Primitive type is one of: integer, real, condition, or enumerated type. In the above matrix, integers are taken as example.			

Running User Code on Solaris 2.9 or 2.10

Running user code on Solaris 2.9 or 2.10 needs a special treatment regarding the libraries libscheduler.so and libsim_scheduler.a.

These libraries should be replaced with the following ones - libscheduler2_9.so and libsim_scheduler2_9.a.or libscheduler2_10.so and libsim_scheduler2_10.a.

- Running Generated Code

  In order to compile and run generated code on Solaris 2.9 or 2.10, the Solaris target file should be modified by replacing the following library options:

  - lscheduler2_9 or lscheduler2_10
  - libsim_scheduler2_9a or libsim_scheduler2_10a

- Running a Simulation with User Code

  No change is required. The correct library is selected automatically according to the operating system.

- Compiling Runtime Libraries

  Runtime libraries for Solaris 2.9/2.10 must be compiled on a Solaris 2.9 or 2.10 system. In addition, the following compilation flags are required:

  ```
  “-D_MAKECONTEXT_V2_SOURCE -DSOLARIS_29”
  ```

  or

  ```
  “-D_MAKECONTEXT_V2_SOURCE -DSOLARIS_210”
  ```

Adding STM Code Modules

To obtain a working prototype of the system, you can extend your handwritten code with Rational Statemate code modules. Use this option when **most** of your model consists of handwritten code, but you want to supplement it with some Rational Statemate-generated code.

Note

When the majority of your model consists of Rational Statemate-generated code, refer to **Adding User-Written Code.**

This section explains how to generate Rational Statemate (or STM) code modules. The module's format makes it easier to take the generated code out of Rational Statemate and incorporate it into your handwritten code.

Normally, Rational Statemate-generated code consists of an entire executable that includes the main, scheduling, data management, interrupt handling, and all other necessary services. Since you are supplying these services, you only need a self-contained module. This option generates a module of code rather than the entire executable.

These modules are as follows:

- Callable from the handwritten code
- Accept and return values
- Perform either a step or a super step when started

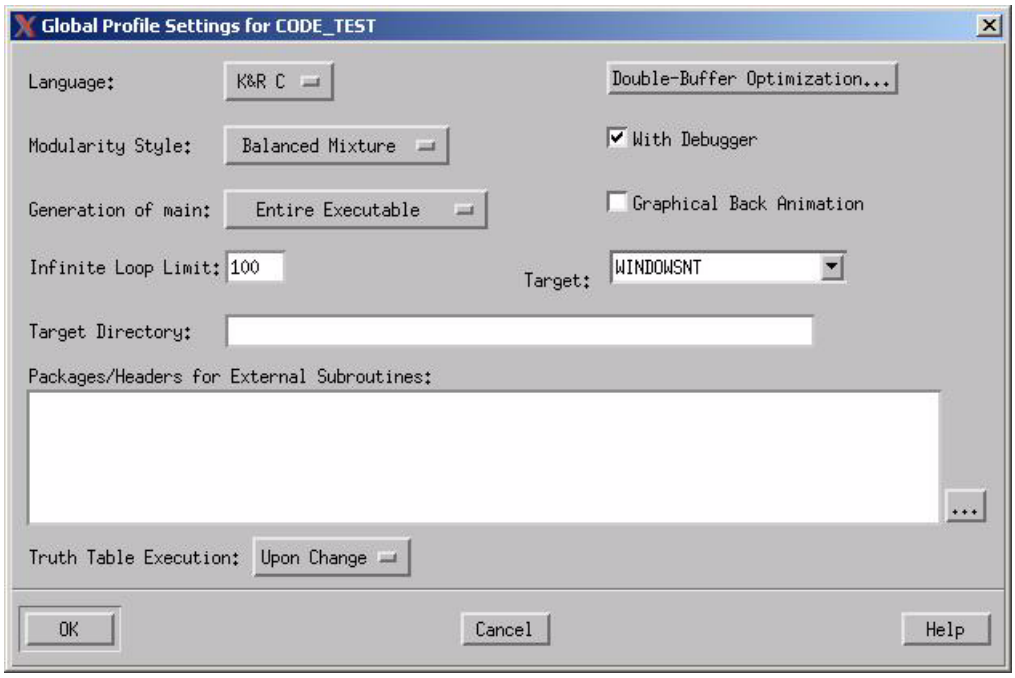
Modules, however, do **not** communicate or synchronize with any other Rational Statemate modules. Your handwritten main body must perform the communication or synchronization functions between Rational Statemate-generated modules.

Generating Modules of Code

Complete the following steps to generate a self-contained module instead of an entire executable:

1. Select the **Options > Global Profile Settings** menu item from the Profile Editor's main menu.

   The **Global Profile Settings** window opens.



2. Select **Module Procedures Only from** the Generation of main field.

3. Click **OK.**

Note

Because the modules are not executables, the Code Generator disables **Main Setting, With Debugger**, and **Graphical Back Animation**. The **Panels** button in the Profile Editor main screen is also disabled.

Setting Module Parameters

After selecting **Module Procedures Only,** you can set the parameters for an individual module.

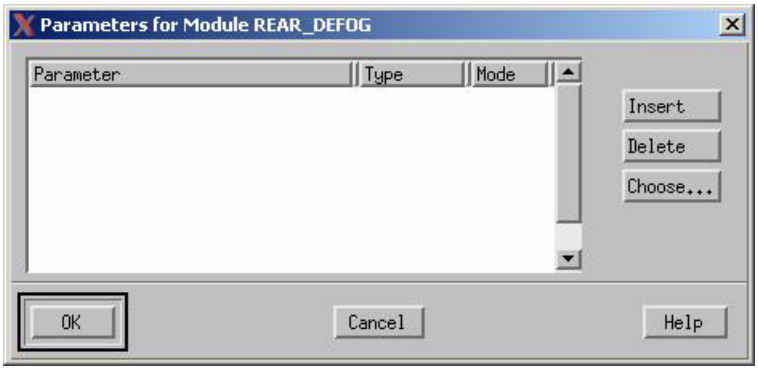
Complete the following steps to set parameters:

1. Select a module in the profile.
2. Select **Options > Module Settings** from the Profile Editor's main menu.

   The Module Settings window opens.

Note: Because you selected **Module Procedures Only** in the previous procedure, the Code Generator disables **Separate File per Statechart**, and enables **Parameter Setting.**

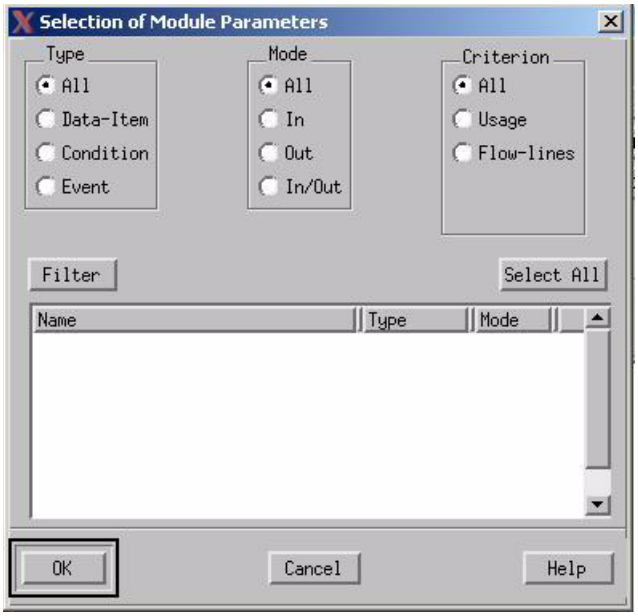
3. Click **Parameter Setting.** The Parameters for Module xx window opens.



Note: The order in which you enter the parameters is very important since this is the order in which they appear in the `<module_name>_init` function.

4. Enter the **Type** (Data-Item, Condition, Event) and **Mode** (IN, OUT, IN/OUT) parameters for the selected elements.

5. Click **Choose** to select additional parameters. The Selection of Module Parameters window opens



Generated Procedures and Files

When you generate code with **Module Procedures Only**, each module generates the following procedures and files.

Generated Procedures

Modules generate the following procedures:

- `<module_name>_init()` initializes the module. If the module is already active, calling it again re-initializes it. This procedure accepts all the elements that communicate with the module as parameters. Calling the module with these parameters performs the actual-to-formal binding.
- **<**`module_name>_exec()` calls the module and executes either a single or a super step depending on how the module is called.
- `<module_name>_status()` returns the module's status which can be in one of the following states:
  - return module_stable
  - return module_terminated
  - return module_working

Generated Files

Modules generate the following files:

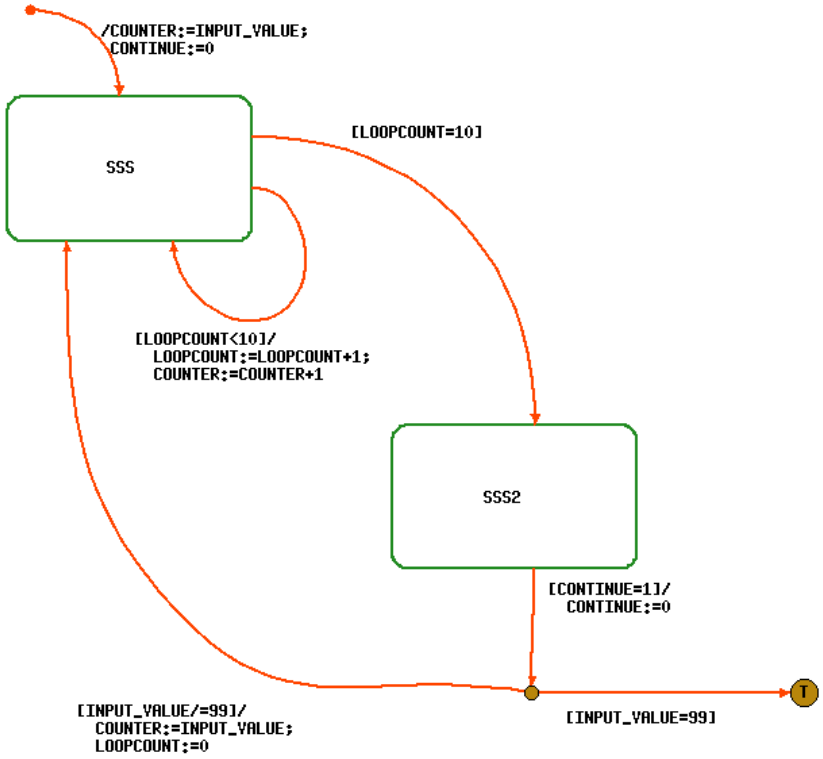
- `<module_name>.c` **-** Contains the code for the module, all the procedures listed above, and the declarations for all the textual elements in the procedures.
- `<module_name>.h` **-** Is the include file that includes all types and external declarations defined within the MODEL scope of the related module.
- `<procedure_name>.c` **-** Contains the code for functions/ procedures. This is consistent with the rest of the generated code where separate files are generated for each function and procedure used within the related model scope.
- `TOP_<module_name>.h` **-** Contains declarations of types defined outside (above), but used within the module scope.

Sample Code Module

The code samples show what the code looks like when you perform the following tasks:

- Generate code with **Module Procedures Only** selected. Refer to **example.c**.
- Generate the makefile. Refer to **Generated Makefile.**
- Generate the makefile and then modify it to work with your handwritten code. Refer to **Modified Makefile** for more information.
- Include the *main* in your code. Refer to **my_main.c** for more information.

The Statechart in the following figure describes the sample module.



example.c

The following code sample illustrates how the generated code looks when you select **Module Procedures Only**.

```
/*  */
/* Created: 16-MAY-1997 */
/* Compilation Profile: prof1 */
/* File Name: example.c */
/*  */
 .
 .
 .
void example_init_module(instance, _INPUT_VALUE, _COUNTER, _CONTINUE)
  sw_module_ptr  *instance;
  int  *_INPUT_VALUE;
  int  *_COUNTER;
  int  *_CONTINUE;
{
 .
 .
 .
} /* example_init_module */

sw_module_status example_exec(instance, single_step)
  sw_module_ptr  instance;
  boolean        single_step;
{
 .
 .
 .
} /* example_exec */
```

```
sw_module_status example_get_status(instance)
  sw_module_ptr  instance;
{
 .
 .
 .
}
```

Generated Makefile

The following shows the generated makefile.

```
all : out_lib.a


CC = cc


OBJECTS = \
    job_man_dictionary.o\
    example.o\
    init_queue_tail_sc.o\
    job_priority_sc.o\
    remove_job_sc.o\
    empty_queue_sc.o
CFLAGS = -O  -I$$STM_ROOT/etc/prt/c -I$$STM_ROOT/etc/sched -DPRT  -
Dsparc


out_lib.a :$(OBJECTS)
    ar rvu out_lib.a $(OBJECTS)
    ranlib  out_lib.a


example.o :example.h
```

Modified Makefile

The following shows the makefile that has been modified to work with handwritten code.

```
all : my_main


CC = cc


OBJECTS = \
    example.o \
    my_main.o
CFLAGS =  -g -I$$STM_ROOT/etc/prt/c -I$$STM_ROOT/etc/sched -DPRT -
Dsparc


out_lib.a :$(OBJECTS)
    ar rvu out_lib.a $(OBJECTS)
    ranlib  out_lib.a


my_main : out_lib.a Makefile
           $(CC)   $(CFLAGS) -o $@ \
            my_main.o example.o \
           -L$$STM_ROOT/lib -lintrinsics -lscheduler -lm


my_main.o :


example.o :example.h
```

my_main.c

The following shows a sample .c file that you would have to write. It contains the main for `example`.

```
/* my_main.c */

#include <stdio.h>
#include “types.h”
#include “example.h”

main()

{

 int CONTINUE=0;
 int ACTUAL_DI=0;
 int COUNTER_INIT=0;

 sw_module_ptr MODULE_HANDLE=0;
 sw_module_status MODULE_STATUS;


 example_init_module(
          &MODULE_HANDLE,
          &CONTINUE,
          &ACTUAL_DI,
          &COUNTER_INIT);

 while (MODULE_STATUS != module_terminated)
 {

   printf(“Enter value to initialize counter (99=quit)\n”);
   scanf (“%d”, &COUNTER_INIT );
   CONTINUE=1;
```

These *include* files enable you to:

- Print
- Access Rational Statemate’s type definitions ($STM_ROOT/etc/prt/c/types.h)
- Access the model’s type definitions

These variables map to the model’s parameters.

These are the handle and status variables for this model.

This call initializes the module and “binds” or “maps” the parameters that you defined when you created the profile.

Loop until module terminates.

Get an initial value for the counter. Notice that to set the input parameters, there are no Statemate-specific calls.

```
while ((MODULE_STATUS == module_working) || CONTINUE)
   {
     MODULE_STATUS = example_exec(MODULE_HANDLE, 1);
     printf (“Counter value =  %d \n”, ACTUAL_DI);
     printf (“Module Status - %d \n\n”, MODULE_STATUS);
   }
 }

 printf(“ Statemate module has terminated \n”);
}
```

Loop as long as the module is working or the *CONTINUE* variable is true. Set *CONTINUE* after changing the inputs to the module.

Execute a single step. The second parameter is true (1) for a single step, and false (0) for a superstep.

Print the desired variable and status. Notice that there are no Statemate-specific calls.

To reach this point, the module must return a module_terminated status, which means it reached a termination connector.

Debugger

The Code Generator Debugger helps you find errors in the specification in a manner similar to symbolic debuggers used for conventional high-level programming languages.

The Debugger can locate specification errors in terms of the Rational Statemate specification objects (states, activities, events, conditions, and data-items), rather than in terms of the generated code. In fact, most users of the prototype code do not know the structure and content of the code.

This section explains how to:

- Control the execution by stopping at chosen breakpoints (e.g., when selected events occur)
- Monitor the execution by examining the current status of states and activities, values of events, conditions, and data-items and by inspecting scheduled timeouts
- Affect execution by modifying event, condition, and data-item values
- Create execution trace files for off-line analysis by choosing specification objects to be traced

The Debugger has an interactive command language through which you control the debugging session. Predefined command sequences can be stored in files and started during the session.

The Debugger also has an online help facility. It provides information on command syntax and usage.

Generating Prototype Code With Debugging Facilities

To enable the Debugger and produce prototype code with debugging facilities, check the **With Debugger** option under the **Options** menu. Refer to **Selecting Code Parameters** for more information.

Compilation and link of the generated code result in an executable file called `<profilename>_dbg`. By default, the Code Generator stores this file in your workarea under the "`prt`" subdirectory.

A program generated with the Debugger facility consumes more memory and is slower to execute. Therefore, at the point in the specification development that you no longer need the debug facility, you may want to generate a prototype without the Debugger to obtain code with better performance.

A Debugging Session

A debugging session consists of alternating modes of operation when the prototype is executing and when the prototype pauses its execution and allows debugging commands to be entered. When the prototype is executing, it behaves as defined in the specification. When the execution pauses, you may enter debugging commands.

The executing prototype enters the debug mode in the following cases:

- When you first invoke the debugging prototype
- When a breakpoint event occurs
- After completing a specified number of steps through the execution
- By explicitly interrupting the execution with **`Control-C`**

Prototype Behavior In Debugging Session

Switching from execution mode to debug mode and entering debugging commands can have unintended side effects in a real-time system. This is because, in debug mode, input events are not processed and the Debugger commands may put the system into a status that could not have otherwise been reached. More specifically, during the time the debugging prototype is in debug mode, the following takes place:

- Controls of all nonprimitive activities remain in their current state configurations
- Each primitive activity remains at one of its synchronization points
- All changes of events, conditions and data-items produced by the specification's environment are ignored. In debug mode, these elements can be changed only via debugging commands.
- System time is not advanced. As a consequence, time-out events and scheduled actions are delayed. They progress in time only when the execution is not in debug mode.

Note that in many practical situations, you may have enough control of the actual environment to avoid the loss of events and values. For example, assume that the interaction with the prototype is performed via a graphic control panel and events are generated by *clicking* the mouse on graphical buttons. In this situation, no events are generated and lost if you refrain from *clicking* any of the graphical buttons while in the debug mode.

Also, if necessary, you can explicitly enter the events and other value changes that are needed to get correct prototype behavior manually, via the appropriate Debugger commands (**SET EVENT**, **SET CONDITION**, and **SET DATA_ITEM**). This must be done with caution since all these events and value changes are considered as happening simultaneously, at the first step that follows the resumption of the execution.

Switching from execution mode to debug mode does not, by itself, cause any change in the current status. Entering debug mode is different from the Rational Statemate suspend action. In particular, conditions of type hanging(A) do not become true. Similarly, resuming the execution does not make these conditions false.

Debugger Command Conventions

The Debugger commands, which are described in detail later in this section, are divided into four groups.

Session Control

QUIT	Exits the debug mode.
GO	Starts or resumes execution of the prototype.
STEP	Executes a number of steps of the prototype.
LOAD	Reads and executes Debugger command files.
SET OUTPUT	Logs a transcript of the session to a file and/or the terminal.
HISTORY	Re-executes previously entered commands.
HELP	Provides on-line help for Debugger commands.

Monitoring and Modifying Object Values

LIST	Lists the specification objects.
SHOW	Displays the status of the objects.
SET	Modifies the status of the objects.
PUT, UPUT, FLUSH	Modifies the status of queues.

Creating Trace Files

SET TRACE	Instructs the Debugger to report every time an object changes its status.
SHOW TRACE	Shows which objects are being traced.
CANCEL TRACE	Cancels a previously requested trace.
SET TIME	Puts time stamps in trace messages.
CANCEL TIME	Cancels time stamps in trace messages.

Handling Breakpoints

SET BREAK	Instructs the Debugger to pause the specification's execution and enter debug mode when a specified event occurs.
SHOW BREAK	Shows which breakpoints are active.
CANCEL BREAK	Cancels previously set breakpoints.

Reference to Rational Statemate Objects

This section describes how to refer to Rational Statemate objects in Debugger commands.

Rational Statemate Objects Classes and Subclasses

Different Rational Statemate objects can be manipulated in different ways in the debugging prototype. In Rational Statemate specifications there are nine basic classes of objects:

```
Action    Activity    Condition
Data-item Event       Flow Line
State       Transition  User-Defined Type
```

Some classes of Rational Statemate objects are further subclassified. Some Debugger commands work only with specific subclasses. The following table summarizes the classes and subclasses:

CLASS	SUBCLASS	ACCESSIBLE IN DEBUGGING PROTOTYPE
Activity	Primitive Non-Primitive	Yes
Condition	External Internal	Yes
Data-item	External Internal	Yes
Event	External Internal	Yes
State	Basic Non-basic	Yes
Action		No
Flow-line		No
Transition		No
User-Defined Type		Yes

States

You can review the model's current state by typing the **SHOW STATE** command. You cannot change the configuration to be a set of specific states by directly naming those states.

Subclasses: Basic/Non-Basic - a state is basic if it is not decomposed into substates.

Activities

The status of activities can be inspected while in debug mode via the **SHOW ACTIVITY** command. There is no direct way to start, stop, suspend and resume an activity from the Debugger.

Subclasses: Primitive/Non-Primitive - an activity is considered primitive in one of the following cases:

- It is part of the lowest level of decomposition in the activity-chart, i.e. it does not contain any subactivities and has no controlling statechart.
- It contains subactivities and/or has a controlling statechart but the user has requested that it be treated as a stub when creating the prototype.

For primitive activities, the Code Generator generates only templates. These templates must be completed manually with the code describing their behavior.

Events, Conditions and Data-items

You can refer to events, conditions and data-items by name, and inspect the values of any event, condition or data-item (with the exception of structured data-items). You can modify the values of any primitive condition or data-item as well as generate any primitive event. An event that is generated is said to be active.

Subclasses: External/Internal - Events, conditions and data-items produced by an external or primitive activity are external. If produced by a control activity, then they are internal. Note that it is possible for an event, condition or data-item to be both internal and external. When you modify the value of an item that does not come from the environment, then you are cheating and changing the model's behavior. If you change data coming from the environment, then you are simulating the environment.

User-Defined Types

User-Defined Types can be accessed by name. The **SHOW** and the **LIST** commands can be used to view the type and its structure. User-Defined Types cannot be modified from the Debugger.

Actions

There is no way to invoke a Rational Statemate action by its name. Most of the action syntax is available through the appropriate **SET** command. Only scheduled actions and actions on activities cannot be issued from the Debugger.

Flow Lines

Flow lines in activity-charts are not modeled in the prototype. Flow lines signify that data from one activity are available to another activity; they do not model a specific channel through which the data flow. Therefore, though the data that flows through the flow lines is accessible to the Debugger, the flow lines themselves are not. The existence of flow lines in the activity-chart determines the externality of the element and whether it is available for use by the environment and primitive activities.

Transitions

You cannot command the prototype to perform a particular transition. However, you can set the proper events, conditions and data-items so that the trigger of a transition leading from a currently active state becomes enabled - and the transition is taken in the following step. Also, there is no query of available transitions or relevant object.

Names and Synonyms

In Debugger commands, you refer to all objects by name. If the object has a synonym, then you can also refer to it by synonym. For example, if an event POWER_ON has the synonym P_ON, you may generate this event with either of the following commands:

```
SET EVENT power_on:=true
SET EVENT p_on:= true
```

You may also refer to a particular element by its name in one command and its synonym in another command.

You can use the name of the chart the objects belongs to as a prefix to its name with a separating colon.

For example:

```
SET EVENT ews:p_on:=true
```

Whenever names are displayed, either in trace messages or in the output of the Debugger, this last convention, including the chart name, is used.

If a synonym also exists, it is printed in parentheses next to the name. For example, in response to the input:

```
Pdb > LIST EVENT
```

you would get:

```
       List of events
. . .
ews:power_on(p_on)
. . .
```

Referring to Unnamed Objects

Rational Statemate objects need not be assigned a name. This is an acceptable practice in many situations. For example:

- Orthogonal components of an and-state may not have a name since you usually refer to substates of such components but not to the components themselves.
- Expressions are often used in transition triggers without giving names to events and conditions they represent. For example, a transition may be triggered by the event **entered(S)** without this event having a defined name.

Note

Recall that an instance state or an instance activity has no explicit name (for example, was labeled as `@CHART_NAME`), then this element is given a default name `CHART_NAME`. Hence, such elements are not considered as unnamed in the following discussion.

Unnamed Activities and States

For all charts that you did not name, the Debugger assigns internal names. This allows you to reference them when using Debugger commands. Internal names of unnamed states and activities are constructed as follows:

```
STATE#id_number
ACTIVITY#id_number
```

Using the Debugger command **LIST STATE** results in the following display:

```
ews:ST_OUT
       ews:STATE#1201
       ews:ST_IN
ews:STATE#1202
```

Since the hierarchy of the states is used in formatting the output of the **LIST STATE** command, it is easy to see that the unnamed state within `ST_OUT` has the internal name `STATE#1201`.

You can use the internal name to reference the state or activity. For example:

```
Pdb > SET BREAK bp entered(STATE#1201)
```

Unnamed Events and Conditions

Unnamed basic events and conditions may be referred to exactly as in the specification. For example, you may define a breakpoint on a basic event:

```
Pdb > SET BREAK user stopped(A)
```

even if this event was not referenced at all in the prototyping scope.

Resolving Name Ambiguity

Two or more elements may have the same name. For example, the specification might contain an and-state ON with two orthogonal components READING and MONITORING, each of them containing a substate called WAIT. Another example is two events with the same name E defined in two different charts.

Many Debugger commands operate on a group of elements (**LIST**, **SHOW** and the different **TRACE** commands). If such a command is given with a non-unique object name, then it is applied to all the objects with that name.

However, there are commands which expect the argument to be uniquely defined. If these commands are given a non-unique object name, they are ignored and an error message is displayed. To make the reference unambiguous, you can either use the chart name as described above and/or prefix the object's name with its ancestor(s) name(s) separated by periods, up to a point where the full pathname given is unique.

In our example of the ON state, we could uniquely identify the two orthogonal components in a show state command by entering the following:

```
Pdb > SHOW STATE reading.wait
Pdb > SHOW STATE monitoring.wait
```

In another example, to uniquely identify a specific event E, precede it with the name of the chart in which it is defined:

```
Pdb > SET EVENT chart1:e:=true
```

Wildcard Abbreviation (*)

In all Debugger commands, you can abbreviate the names of objects and breakpoints.

To abbreviate, you type a wildcard symbol (*) anywhere in the name. The symbol means that any sequence of characters may replace it. The specified command is then applied to all objects, whose names match the pattern. For example, the command sets a trace on all the states whose names begin with the letter "s" including the state whose name is S.:

```
Pdb > SET TRACE STATE s*
```

The following command sets a trace for all states whose names begin with the letter "a" and end with the letter "t" and which belong to charts whose names begin with the letter "e".:

```
Pdb > SET TRACE STATE e*:a*t
```

The command deletes all breakpoints whose names contain the letter "b" immediately followed by the letter "p".:

```
Pdb > CANCEL BREAK *bp*
```

Note

The use of **SET BREAK** requires explicit use of the breakpoint name. No wildcards are permitted.

Subobjects Operator (^)

To apply a Debugger command to a hierarchical object (activity or state) and all its descendants, type the subobjects operator "^" character immediately after the name of the state or activity.

For example, in response to the command:

```
Pdb > SHOW STATE S^
```

the Debugger lists the state S and all its currently active substates.

Referencing Multiple Rational Statemate Objects in Commands

As already mentioned above, many of the commands can operate on more than one Rational Statemate object. In addition to the above ways of specifying more than one item (the use of non-unique names, wildcards and subobjects operator), you may just list the object names separated by commas. Spaces on either side of the comma are optional.

Some examples follow:

```
Pdb > SHOW STATE x1, x2, x3
Pdb > SET TRACE CONDITION p*, cc, *a*, *w*:*
Pdb > LIST ACTIVITY act1^, act2
```

Referencing Records and Unions in the Rational Statemate Debugger (Pdb)

User-Defined Types can be referenced in generated code debug using the standard form of naming conventions described earlier in this section. Use of partial names is allowed. For example, if an array of 20 invoices was defined, where invoice is a record, the command:

```
SHOW data INVOICE(0..2)
```

might produce the output

```
(array of INVOICE_TYPE) USAGE_TEST:INVOICE(0) .NAME = 'Fred B'
                        .ITEM = 'Biscuit'
                        .AMOUNT = 2.45
    (1)    .NAME = 'Joe M'
           .ITEM = 'Milk'
           .AMOUNT = 0.69
    (2)    .NAME = 'Jim M'
           .ITEM = 'Toothpaste'
           .AMOUNT = 1.55
```

and the command:

```
SHOW data INVOICE(0).NAME
```

might produce the result

```
(array of INVOICE_TYPE) USAGE_TEST:INVOICE(0).NAME = 'Fred B'
```

Union structures are displayed in the same way, but fields that are not current may show unusual values. It is only the field that had its value assigned most recently that shows a valid result.

The command interface has been extended to allow types to be shown, so using the previous example, the command...

```
SHOW TYPE INVOICE
```

would give the result:

```
USAGE_TEST:INVOICE is  array (0..19) of Record
              INVOICE_TYPE
 end of record
```

and the command...

```
LIST TYPE INVOICE_TYPE
```

will give the result:

```
INVOICE_TYPE record
       NAME is string(80)
       ITEM is string(80)
       AMOUNT is real
 end of record
```

Referencing Queues in the Rational Statemate Debugger (Pdb)

Commands to allow queues to be displayed and modified in the debug environment are:

```
PUT <QUEUE_name> QUEUE_element_value
UPUT <QUEUE_name> QUEUE_element_value
FL <QUEUE_name>
```

The existing command **SHOW DATA** can be applied to queues, and lists the elements of the queue. The queue element tagged number 1 is the top of the queue, and the highest number is the end of the queue.

Keywords

There are no reserved words in the Debugger. It is clear from the context whether a word is used as an object name or a Debugger keyword. For example:

```
Pdb > SET BREAK br_label go
```

Here, "go" is obviously the name of an event and not a command keyword.

However, in some of the Debugger commands there are cases where both a Debugger keyword and an object name may be given. This occurs in all the **SHOW**, **LIST** and **TRACE** commands - where you can give either an object name or a keyword that denotes a subclass of Rational Statemate objects (for example, "basic," primitive, etc.) as an argument.

If an object exists whose name is the same as a keyword, and the user wishes to specify this object in a command, its name must be specified within quotes. For example:

```
Pdb > SET TRACE ACTIVITY PRIMITIVE
```

means trace all the primitive activities, while

```
Pdb > SET TRACE ACTIVITY "PRIMITIVE"
```

means trace all the activities whose name is PRIMITIVE.

Generally, each of the keywords used in Debugger commands may be abbreviated. If the abbreviation is ambiguous, then an error message is displayed followed by the list of possible meaning for abbreviation. For example:

```
Pdb > L
```

causes the following response:

```
Ambiguous keyword abbreviation: L. Possible meanings:
LIST
LOAD
```

Debugger Commands

The following sections describe the debugger commands.

Activating the Debugger

You activate the debugging prototype by invoking the executable file called `profileg_name_dbg`.

The Debugger responds:

```
Welcome to Debugger of Generated Code
```

A prompt is displayed to show that the Debugger is ready to accept Debugger commands:

```
1 Pdb >
```

Each prompt is preceded by its sequence number, thus enabling the identification of every command entered during the debugging session. This allows you to easily re-enter these commands using the **HISTORY** option.

You may then start the execution of the prototype by typing the **GO** or **STEP** command.

Quitting the Debugger

To terminate a debugging session, use the **QUIT** command:

```
Pdb > QUIT
```

This stops the execution of the prototype code. To also stop those tasks that you might have added in the user-written code, the command performs a call to the `profile_name_user_quit` routine. This routine resides in the file `user_activities`. Before creating the debugging prototype, you may want to edit this routine's template.

Entering Debugger Commands

You can enter Debugger commands any time the Debugger prompt is displayed:

- At the beginning of the debugging session
- When the execution reaches a breakpoint
- After the prototype finishes executing a step command
- When prototype execution is interrupted via the **Ctrl-C** interrupt

There are two ways of entering commands. One way is by interactively typing them at the terminal's keyboard. Each command can be up to 256 characters long. No special symbol is required in order to continue the command on a new line. You can use abbreviations of command keywords. Complete the command by pressing **carriage return**.

Another way of entering commands is via pre-existing command files. If there is a sequence of Debugger commands that you frequently use, you can collect them in a file and invoke the file from the Debugger using the **LOAD** command:

```
Pdb > LOAD file_name
```

The Debugger reads the commands in the given file and executes them. After reading the last command of the file, the Debugger displays its prompt - at which point you can type in more commands.

You may `LOAD` as many command files in one session as you desire. Command files can themselves load other command files, up to a nested level of ten calls.

The argument `file_name` can be a full pathname explicitly specifying the directory in which the file resides. If no pathname is provided, the Debugger searches for the file in the current directory (the one from which the debugging prototype was started).

The HELP Facility

The Debugger comes with its own on-line help. When the `Pdb` prompt is displayed, you can get a list of available commands, their syntax, and usage. To activate the help facility and display a table of topics and commands for which help information is available, simply type:

```
Pdb > HELP
```

TOPIC	COMMANDS
Break	SET BREAK, CANCEL BREAK, SHOW BREAK
Execution	GO, STEP, INTERRUPT, QUIT
Help	HELP
History	HISTORY `number, !number, !text, !!`
Input	LOAD
Output	SET FILE, SET OUTPUT, CANCEL OUTPUT
Time	SET TIME, CANCEL TIME
Trace	SET TRACE, CANCEL TRACE, SHOW TRACE
Values	SET OBJECT, SHOW OBJECT, SHOW SCHEDULE. PUT QUEUE, FLUSH QUEUE
CE Evaluation	SET CE_UPDATE, CANCEL CE_UPDATE
List	SHOW CE_UPDATE
Notes	LIST OBJECT

Help is organized hierarchically, from overall help to topics and from each topic to the commands that belong to it. After choosing a topic, the Help facility displays a general explanation of the topic and lists its corresponding commands. You can get further information about an individual command by referring to its name.

If you know exactly which topic or command you want help, you can avoid successively going through all hierarchical steps. Instead, ask for the needed help directly from the Debugger command level. For example:

```
Pdb > HELP TRACE
```

gives you general information on the topic of tracing, while:

```
Pdb > HELP CANCEL TRACE
```

provides specific information on the **CANCEL TRACE** command.

Starting and Controlling Execution

As mentioned earlier, when you activate the debugging prototype you get the prompt `Pdb >`. The debugging prototype is in the debug mode and execution of the prototype has not yet started. At this moment:

- Selected conditions, data-items, and events are initialized in accordance with the definitions you put into the `program_name_user_init` routine.
- All activities of the model, as well as states of their controlling statecharts are non-active.

For all objects not initialized in the `program_name_user_init` routine, the following holds:

- Events are not generated.
- Conditions have the initial value “false.”
- Numeric data-items have the initial value zero.
- String data-items are empty.

Use the **STEP** and **GO** commands to start or resume execution of the prototype whenever the debugging prototype is in debug mode. The main function of **STEP** is to execute a number of steps and stop at the beginning of the next step. The main function of **GO** is to execute the prototype until suspension at the next breakpoint. There is also the interrupt option providing immediate suspension of the execution.

STEP Command

The **STEP** command provides the most elementary way of advancing and suspending the execution. You need not define any breakpoints. As a reaction to this command, the model executes the specified number of steps, passing from its current configuration to another, where the execution is again suspended. The command is entered as follows:

```
Pdb > STEP number
```

If no number is specified, one step is taken.

In particular, you can start the proper run of the code by using **STEP** as the first execution command in the debugging sessions. As a result, the model enters its default configuration and performs all actions attached to the corresponding default transitions.

GO Command

The **GO** command instructs the prototype to execute. If the prototype was suspended, **GO** resumes execution. The syntax of the **GO** command is:

```
Pdb > GO
```

As a result, a series of successive steps is executed until a breakpoint is encountered. The Debugger then displays a message:

```
Stopped at breakpoint BP_NAME on event :
EVENT_NAME
```

Several messages may appear when several events, on which you have set breakpoints, occur simultaneously (in the same step).

You can use **GO** not only to resume execution after the prototype was suspended, but to start prototype execution. You usually begin the debugging session by defining the required breakpoints and then issue the **GO** command. The difference between using the **STEP** command for this purpose is that, here, the execution is not suspended after entering the default configuration.

The **GO** command can also be used without any breakpoints. This option is useful when you want to perform a prototype run containing debugging facilities without suspension of the execution, check of object's status, etc. Though any run of such code causes activation of the Debugger, you can perform such a run by typing GO as the first and only command in the session.

In all cases, when you issue a **GO** command, the execution continues until one of the following occurs:

- A breakpoint is reached
- An interrupt is issued
- The root activity terminates on its own

Interrupting Prototype Execution

Interrupting the prototype execution causes it to immediately pause, thus enabling the Debugger to read and execute debugging commands. The execution of the controlling statecharts is paused, as are all primitive activities, each of which pauses at its next synchronization point.

When interrupted, the prototype execution is not stopped in the middle of a step but finishes the current step and only then the Debugger prompt appears, allowing you to enter debugging commands. When this is done, you can resume the execution of the prototype using the **GO** or **STEP** commands.

To interrupt the prototype execution, use **Ctrl-C**. Issuing an interrupt while the Debugger prompt is displayed does not produce any results.

For example, suppose that the specification enters a loop of transitions in which there are no breakpoints and you decide that you want to trace certain objects while in the loop. You can interrupt the prototype, turn on traces to the desired objects and then resume execution.

HISTORY Command

The **HISTORY** command allows you to easily invoke any previously entered Debugger command. Each time the Debugger prompt appears, a sequential number is displayed. You can later use this number to reference the command entered at the prompt. To see the list of commands used in the session, perform the following:

```
Pdb > HISTORY
```

The Debugger then displays the most recently entered commands, up to a maximum of 20, with their reference numbers.

To re-enter a specific command, enter the following:

```
Pdb > !command_number
```

where `command_number` is the command reference number.

For example, if, in the course of the debugging session, you gave the command:

```
90 Pdb > SET CONDITION cstop := true
```

You could re-execute this command later by entering:

```
103 Pdb > !90
```

To re-enter the last performed command, enter:

```
Pdb > !!
```

Another way to execute a previous command is to enter:

```
Pdb > !text
```

where `text` is a text string uniquely matching a previously entered command.

For example, if, after the command number 90, no command starting with the letter “s” was given, then you can execute this command by entering:

```
Pdb > !s
```

LIST Command

The **LIST** command instructs the Debugger to output a list of objects belonging to the prototyped specification. You can choose whether to list all the objects in the specification, or to select only those of a certain class, subclass or name. The listing does not show the values or status of the objects in the current prototype execution. It merely lists those objects that are within the prototyping scope.

The **LIST** command can be used in each of four basic forms:

```
Pdb > LIST
Pdb > LIST object_class
Pdb > LIST object_class subclass
Pdb > LIST object_class list_of_objects
```

For example, the following are valid **LIST** commands:

```
Pdb > LIST EVENT EXTERNAL
Pdb > LIST DATA input_value, y*
Pdb > LIST ACTIVITY
```

The order in which the information is displayed is hierarchical for activities and states and alphabetical for events, conditions and data-items. Remember that the command shows all objects in the system, regardless of their current status in the execution.

For example, the command: **Pdb > LIST ACTIVITY PRIMITIVE** lists the names of all primitive activities in the specification, regardless of which ones are currently active.

Similarly, if you type:

```
Pdb > LIST STATE NON_BASIC
```

the Debugger lists all the non-basic states in all the controlling statecharts in the prototype scope—not only those states which belong to the current configuration.

When applied to non-graphical objects (events, conditions, data-items), the **LIST** command displays the requested information and, in addition, marks all compound objects as in the following example:

```
Pdb > LIST EVENT S*
(c)    chart1:  signal
       chart2:  switch
(c)    chart2:  scroll
```

The compoundness attribute allows you to easily identify those elements to which command **SET OBJECT** cannot be applied.

SHOW Command

In the Debugger, you can monitor the status and value of objects using the **SHOW** command. Unlike the **LIST** command, **SHOW** displays the actual value or status of the Rational Statemate object at the current execution point. Thus, at a breakpoint, you can examine the values of different objects such as data-items, conditions and records to check if their actual values correspond to the expected values.

Using the **SHOW** command, you can modify the values of conditions and data-items, or generate and reset events. When you resume the execution of the prototype, the new values take effect.

You can also check which time-outs and scheduled actions are currently pending with the **SHOW SCHEDULE** command.

The **SHOW** command can be used in the same basic form as the **LIST** command:

```
Pdb > SHOW
Pdb > SHOW object_class
Pdb > SHOW object_class subclass
Pdb > SHOW object_class SHOW_of_objects
```

This provides great flexibility in limiting your request to only the information that you need.

SHOW with no arguments gives you the most complete information about the current status of the system. This information includes the following:

- Current step number.
- Status of all activities (active, suspended, nonactive), organized by the activity's hierarchy. Descendants of nonactive activities are not shown explicitly, since they are all nonactive.
- Current state configurations of all controlling statecharts of active and suspended activities, ordered according to activity and state hierarchies.
- Current values of all conditions, in alphabetical order.
- Current values of all data-items, in alphabetical order.
- Currently active events, in alphabetical order.

The second form of the **SHOW** command provides you with information on all objects of the selected class. To get this information, use one of the following commands:

- **SHOW ACTIVITY**
- **SHOW STATE**
- **SHOW CONDITION SHOW DATA_ITEM**
- **SHOW EVENT**
- **SHOW TYPE**

Additionally, the current step number is displayed by entering the **SHOW STEP** command.

The third form of **SHOW** allows you to restrict the displayed information to a subclass of a particular object class. For example:

```
Pdb > SHOW ACTIVITY PRIMITIVE
Pdb > SHOW STATE NON_BASIC
Pdb > SHOW EVENT EXTERNAL
```

The fourth form of **SHOW** restricts the information to specific objects of a certain class. You provide a list of object names as a command argument. You may use wildcard abbreviations and the subobjects operator.

For example:

```
Pdb > SHOW DATA signal_level, y*
```

would display the values of the data-item signal_level, as well as the values of all data-items beginning with the letter “y”.

```
Pdb > SHOW ACTIVITY act*^
```

displays the status of each activity that starts with letters “act” and all of their descendants, until it reaches nonactive activities.

SHOW SCHEDULE Command

The **SHOW SCHEDULE** command shows you which timeouts and scheduled actions are pending and how much time remains until the expiration of each one. A timeout is pending from the moment the event which triggers it is generated, until its delay time elapses.

It is important to remember that the timeouts measure the elapsed time while the prototype is actually executing. Therefore, when the debugging prototype is in the debug mode, the system time is frozen. The measuring of elapsed time resumes when the prototype execution is resumed.

For each pending timeout and scheduled action, the Debugger displays the:

- Name of the timeout or scheduled action, if it has a name in the specification.
- Name of the event on which the timeout is defined (the timeout’s trigger), in case this event was given a name, and if not, its expression.
- Original length of the timeout, both in time units and seconds.
- Amount of time remaining until the timeout or scheduled action elapses

SET OBJECT Command

To change the current value of a condition or a data-item, or to change the current status of an event, use one of the following three forms of the SET command:

```
Pdb > SET CONDITION condition_name := boolean_expression

Pdb > SET DATA_ITEM data_item_name := data_expression

Pdb > SET EVENT event_name := event_expression
```

Note that the command is applicable only to primitive objects; you cannot change the value or the status of a compound object.

On the right-hand side of an assignment, you can put any legal Rational Statemate expression whose type corresponds to that of the object on the left-hand side. The value on the right-hand side of the assignment is evaluated, and assigned to the object whose value is being set.

Examples:

```
Pdb > SET CONDITION c1 := true
Pdb > SET DATA_ITEM int_var := 5*y
Pdb > SET DATA_ITEM str_var := “new string”
Pdb > SET EVENT e1 := e2 or e3
```

To generate event *e* independently of other elements’ statuses, type:

```
Pdb > SET EVENT e := true
```

or

```
Pdb > SET EVENT e
```

To turn off or, reset event *e*, type:

```
Pdb > SET EVENT e := false
```

The SET command does not impose the redefinition of the system’s object. In the example above that sets `event e1`, the current statuses of `e2` and `e3` are examined immediately, and if either event is generated, then `e1` becomes generated this time only. Later in the run, having `e2` or `e3` generated does not cause e1 to be generated. Thus, `e1` remains primitive and is not redefined by the command as a compound event.

The **SET OBJECT** command is a very powerful command to use in debugging sessions when you discover an error in the specification, and you want to continue debugging, without first correcting the specification. This command can be combined with the breakpoint operations to temporarily correct mistakes in the specification.

Suppose, for example, that there is a static reaction on entering state S which refers to condition *C* defined as:

```
C = (X=1)
```

and the correct definition should have been:

```
C = (X=1 and Y>0)
```

You can then enter the following:

```
Pdb > SET BREAK bp entered (s)
       DO SET COND c:=(x=1 and y>0); GO END
```

The prototype then behaves as if the error is fixed, allowing you to concentrate on looking for other problems. Later you should correct the definition of C using the Properties Editor.

PUT QUEUE Command

To add an element to the “back” of a queue, use the following command:

```
Pdb>put QUEUE_NAME VALUE
```

The VALUE must be a legal data-item for that queue. The **put** command places the value at the end of the queue opposite from where the next **get** retrieves values. Using **put** and **get** commands in a Rational Statemate model, treats the queue as a first-in/first-out (FIFO) type of queue.

UPUT QUEUE Command

To add an element to the “front” of a queue, use the following command:

```
Pdb>uput QUEUE_NAME VALUE
```

The VALUE must be a legal data-item for that queue. The **uput** command places the value at the end of the queue where the next **GET** retrieves values. Using **uput** and **GET** in a Rational Statemate model, treats the queue as a last-in/last-out (LIFO) type of queue.

FLUSH QUEUE Command

To completely empty a queue use the following command:

```
Pbd> flush QUEUE_NAME
```

This insures that the queue is completely empty at the end of the upcoming model step. If queue **put** (or **uput**) statements are executed, either within the model or through the Pdb debugger, within the same step as the **FLUSH**, the **FLUSH** command takes precedence.

TRACE Command

The Debugger can provide a trace of the prototype execution. If the trace mode is on, the Debugger issues a message whenever a change occurs in the system. You can restrict the Debugger and have it report only certain types of changes. The trace is a history of how the system performed in its actual execution.

The TRACE facility reports whenever a traced object changes value or status. Specifically, it reports:

- Starting, stopping, suspending and resuming activities
- Entering and exiting of states
- Generating events
- Changing values of conditions
- Changing values of data-items

You can optionally have each trace message contain a time stamp specifying when the reported change occurred.

Since the definition of the system's behavior is based on the notion of step, it might be useful in the course of debugging to see step bounds. For this, you use an additional trace option - step trace. In this case, messages are issued upon starting and ending each step.

By storing a trace in a file, you can perform a post-run analysis and check whether the actual behavior matches the expected one. Since the trace refers to specification objects, it is easy to interpret the results of the code run in terms of the original Rational Statemate specification.

SET TRACE Command

The **SET TRACE** command has six forms:

```
Pdb > SET TRACE
Pdb > SET TRACE object_class
Pdb > SET TRACE object_class subclass
Pdb > SET TRACE object_class list_of_objects
Pdb > SET TRACE STEP
Pdb > SET TRACE SCHEDULE
```

Similar to the **LIST** and **SHOW** commands, the **SET TRACE** command can be started on all system objects or restricted to only specified objects. The rules of naming object classes, subclasses and object lists are the same as in the **LIST** and **SHOW** commands.

For example, the command:

```
Pdb > SET TRACE EVENT INTERNAL
```

sets a trace for all internal events and conditions.

Another command:

```
Pdb > SET TRACE STATE sampling^, c*, disconnected
```

sets a trace for `sampling` and all its substates, all states beginning with the letter “c” and the state `disconnected`.

The Debugger displays trace messages on the terminal screen and/or stores them in a file (when used in conjunction with the **SET FILE** command).

Format of Trace Messages

The format of the trace messages is shown in the following examples:

Activity trace:

```
Activity ews:SET_UP started
Activity ews:SET_UP stopped
```

State trace:

```
State ews:OFF entered
State ews:OFF exited
```

Condition trace:

```
Condition ews:IN_CONNECTED changed value to
TRUE

Condition ews:IN_CONNECTED changed value to
FALSE
```

Data-item trace:

```
Event ews:SET_UP generated

Event ews:HALT reset
```

Step trace:

```
/---------------------\starting step 1

\---------------------/ending step 1
```

Timeout events are a special case. Each timeout event causes two trace messages. The first message is printed when the timeout is triggered and the second message is printed when the timeout expires and the corresponding timeout event is generated.

For example, suppose that the specification contains an event TMO defined as “timeout(E,5)” where E is external and that tracing of events is requested. Assuming that the time unit specified is 2.5 seconds, whenever event E occurs, the following message appears:

```
Event ews:E generated

Timeout ews:TMO on event ews:E started for 5
time units (12.5 seconds)
```

and then, after 5 time units elapse and the timeout occurs:

```
Timeout ews:TMO on event ews:E ended after 5
time units (12.5 seconds)
```

SET TRACE SCHEDULE Command

The **SET TRACE SCHEDULE** command traces scheduled actions. Scheduled actions are a special case similar to timeouts. Each scheduled action causes two trace massages. One when it is scheduled, and one when the scheduled time is up and the action which was scheduled is executed.

Schedule Trace:

```
Schedule action CHART1:D1 of PROCESS_ONE
started for 200 time units


Schedule action CHART2:COMM of FFT started
for 5 time units


Schedule action CHART2:COMM of FFT ended
after 5 time units


Schedule action CHART1:D1 of PROCESS_ONE
ended after after 200 time units
```

SHOW TRACE Command

To see what objects are currently traced, you use the **SHOW TRACE** command in one of the following forms:

```
Pdb > SHOW TRACE
Pdb > SHOW TRACE object_class
Pdb > SHOW TRACE object_class subclass
Pdb > SHOW TRACE object_class list_of_objects
Pdb > SHOW TRACE STEP
```

The Debugger presents a list of all the traced objects of the requested class. For example, in response to the first of the previous commands, the following could be displayed:

```
Activities traced:
   system:SAMPLE_DEVICE
   system:SET_UP
Conditions traced:
Data-items traced:
   ews:SAMPLED_DATA
Events traced:
   system:DISCONNECT
   ews:OUT_OF_RANGE
   ews:RESET
   system:TIME_CLICK
States traced:
   ews:MONITORING
      Step trace: OFF
```

In each group of objects, the names are displayed in alphabetical order. An empty group means no object of the corresponding class was traced, as in the case of conditions in the above example. For step trace, its current status is either **OFF** or **ON**.

If you requested a trace and used abbreviations to specify which objects to trace, the list shows the actual names of all the objects being traced, rather than the original abbreviation.

For example:

```
Pdb > SET TRACE EVENT s*
Pdb > SHOW TRACE EVENT
```

This produces:

```
Events traced:

ews:SAMPLED
system:STUCK
ews:SWITCHING
```

CANCEL TRACE Command

The **CANCEL TRACE** command allows you to turn tracing off for some or all objects previously set by one of the **SET TRACE** commands. You may turn tracing back on by re-entering the appropriate **SET TRACE** command.

Specify which traces to cancel by using the same syntax as when they were set (using **SET TRACE**). For example, here are some valid commands to cancel traces:

```
Pdb > CANCEL TRACE

Pdb > CANCEL TRACE ACTIVITY

Pdb > CANCEL TRACE EVENT INTERNAL

Pdb > CANCEL TRACE CONDITION
disconnected,  c*

Pdb > CANCEL TRACE STATE sampling^

Pdb > CANCEL TRACE STEP
```

The commands to cancel traces do not have to exactly correspond to the commands that turned them on. For example:

```
Pdb > SET TRACE EVENT e*g, power_on, s*

Pdb > CANCEL TRACE EVENT energizing,
submerging

Pdb > CANCEL TRACE EVENT *g
```

would leave a trace on for the event `power_on` and all events that begin with “s” and do not end with “g”. Request to cancel a trace that was not set is ignored.

SET TIME Command

The **SET TIME** command tells the Debugger to put time stamps on each trace message and specifies the format of these stamps. A time stamp shows the elapsed time since the prototype execution began. As noted previously, time is not incremented when the prototype being debugged is in the debug mode.

This command has three forms:

- Pdb > SET TIME
- Pdb > SET TIME SECONDS
- Pdb > SET TIME FORMATTED

The first two forms are the same with format SECONDS being default. Time stamps are printed in this case as SS.LLL, where SS is seconds and LLL is milliseconds.

For example:

```
AT 00:00:01:530 : Data-item ews:FAC changed
value to 1

AT 00:02:03:890 : Condition ews:C changed
value to FALSE
```

CANCEL TIME Command

To disable display of time stamps within the all trace messages, use the **CANCEL TIME** command:

```
Pdb > CANCEL TIME
```

To renew time stamps in the trace messages later in the debugging session, you may re-enter the **SET TIME** command.

The Set File, Set Output And Cancel Output Commands

You can record a transcript of the entire debugging session (or any portion) in a log file. This transcript includes the Debugger commands you entered during the session. It also includes the corresponding Debugger output and trace messages. You can control whether the Debugger output is written only to this file or also displayed at your terminal. The appropriate commands are:

- SET FILE
- SET OUTPUT
- CANCEL OUTPUT

The saved transcript can be used for off-line analysis of a prototype's execution. It can also be used as a Debugger batch (command) file, since the Debugger is able to extract commands from the saved transcript. This batch file can be loaded using the **LOAD** command in a later execution of the same or a corrected debugging prototype. This saves you from having to recreate the same scenarios that were already tested in previous runs. It also facilitates comparison of two runs. In this way you can easily check that a specification error detected in the first run has been properly corrected for the second run.

SET FILE Command

The **SET FILE** command specifies the name of the file in which the debugging session is to be recorded. This file is called the log file. To actually start the recording, use the **SET OUTPUT** command. The command is started as:

```
Pdb > SET FILE file_name
```

As an argument, you enter any string which is a legal `file_name` for your environment. If no `file_name` is given, the default name becomes `debug.log`.

The file name can be a full pathname explicitly specifying the directory in which the file resides. Otherwise the file is created in the directory from which the prototype was run. You must have appropriate write privileges in this directory.

When you use multiple **SET FILE** commands in the same session, the recording is written into the log file specified in the most recently entered command. Moreover, in such a case, you lose the ability to log information into the former file since the file is recreated each time the **SET FILE** command is issued.

SET OUTPUT Command

The **SET OUTPUT** command determines where the output transcript is written.

The command has three forms:

- Pdb > **SET OUTPUT FILE**
- Pdb > **SET OUTPUT TERMINAL**
- Pdb > **SET OUTPUT**

In the first case, the transcript is written to the file whose name was specified in the **SET FILE** command. If the **SET FILE** command was not yet issued, then the file `debug.log` is used.

The second form directs the output to the terminal. This impacts trace messages only, since the transcript of user commands and corresponding Debugger responses are always displayed at the terminal.

Note

If you do not use the **SET OUTPUT** command, trace messages are sent to the terminal, but not to any file. Therefore, it is reasonable to use the **SET OUTPUT TERMINAL** command only after the commands **CANCEL OUTPUT** and **CANCEL OUTPUT TERMINAL** to renew the full display of the Debugger output on the screen.

Finally, the form **SET OUTPUT** without any parameters directs output to both the file and the terminal.

Format of a Log File

Each record in a log file is one of the following:

- Command entered in the debugging session.
- Immediate response of the Debugger to the command (for commands such as `SHOW, LIST, HISTORY, HELP`).
- Trace messages.
- Breakpoint occurrence message.

Only entered commands and various forms of the Debugger's responses are recorded in the log file. Outputs produced by the prototype code itself (and printed on the screen and/or in a file) are not recorded in the log file.

While commands are recorded as entered, all Debugger messages are preceded by a double hyphen "--."

For example, the following commands were entered in the debugging session:

```
Pdb > SET OUTPUT
Pdb > SET BREAK br reset
Pdb > SET TRACE EVENT
Pdb > GO
Pdb > SHOW DATA *bound
Pdb > SET DATA lower-bound := 25.0
```

The transcript of the session is recorded in the file `debug.log` and appears as follows:

```
SET BREAK br reset
SET TRACE EVENT
GO
-- Event ews:EXECUTE generated
-- Event ews:GO generated
-- Stopped at breakpoint BR on event: ews:RE
  SET
SHOW DATA *bound
-- Current values of data_items:
--     system:LOWER_BOUND = 20.5
--     system:UPPER_BOUND = 84.7
SET DATA lower_bound := 25.0
```

Using a Log File

One possible use of the log file is for an off line analysis of the prototype run. You use the saved transcript to see what events occurred in the run and in which order. Relationships between various other elements can also be examined. For example, you can check whether two specific activities were ever active simultaneously in the run. Such post-run analysis is very helpful in localizing errors in the specification.

You can also use the log file as a command file in another run of the same or corrected debugging prototype. For this, you simply refer to the log file's name in the **LOAD** command. For example:

```
Pdb > LOAD debug.log
```

As a result, the Debugger reads and performs in turn all commands recorded in the file. The Debugger easily extracts the commands from the entire transcript since they are not marked by a double hyphen. Thus, you can save a debugging scenario and then re-use it in several runs of the prototype.

Recording Comments in a Log File

Whenever the executing prototype pauses and enters the debug mode, you may enter not only Debugger commands, but also comments. Comments have absolutely no influence on the execution. However, they are recorded into the log file as part of the debugging session transcript. This option supports a better understanding of the saved transcript contents in a post-run analysis. You can use comments to:

- Describe the scenario of the environment's behavior under which you are going to check the system's reactions
- Explain your motivation for entering various commands, especially those which change values of specification elements
- Express your immediate impression concerning certain aspects of the observed prototype behavior

Comments are extremely helpful when the transcript is not analyzed immediately after the run or by persons other than those performing the run.

A comment is a free text string starting with a hyphen. You enter it when the Debugger prompt is displayed:

```
Pdb > - free_text
```

Comments are recorded in the log file literally, including the hyphen.

To illustrate the use of comments, suppose that in the above example of the debugging session, you entered the following comments between the commands **SHOW DATA** and **SET DATA**:

```
Pdb > - ***** The difference between the bounds
Pdb > - ***** shouldn't exceed 60.0
```

Then the corresponding portion of the log file appears as:

```
SHOW DATA *bound
- Current values of data_items:
-      system: LOWER_BOUND = 20.5
-      system:UPPER_BOUND = 84.7
- ***** The difference between the   bounds
- ***** shouldn't exceed 60.0
SET DATA lower_bound := 25.0
```

CANCEL OUTPUT Command

To stop recording the debugging session, you use the **CANCEL OUTPUT** command in one of the following three forms:

- Pdb > **CANCEL OUTPUT FILE**
- Pdb > **CANCEL OUTPUT TERMINAL**
- Pdb > **CANCEL OUTPUT**

The first command stops recording into the most recently used logging file (either specified by a **SET FILE** command or `debug.log`).

The second command stops displaying trace messages on the terminal.

Finally, the third command halts the logging of the session in the file and at the terminal.

Remember that in all cases, the commands and Debugger's responses continue to be displayed.

For example, you enter:

```
Pdb > SET OUTPUT
```

This causes all the output of the Debugger to be written into the file `debug.log` as well as displaying it on the screen. If later in the session, you invoke:

```
Pdb > CANCEL OUTPUT TERMINAL
```

then, from that moment (until changed by other commands), the session log is only written into the file.

Breakpoints

Breakpoints specify which events cause the prototype execution to pause and enter the debug mode. You can trigger a breakpoint on every event and condition used in the specification.

The normal cycle for working with breakpoints is:

- **SET** the breakpoints where you want the execution to pause.
- Execute the prototype with the **GO** command to advance from one breakpoint to another.
- When a specified breakpoint occurs, inspect and/or modify the prototype using the **SHOW**, **SET** or other Debugger commands.
- Repeat the cycle.

When stopping at a breakpoint, you can enter any Debugger command and define new breakpoints, or check the status and values of objects. Also, when setting a breakpoint, you can associate it with a sequence of commands which are performed automatically when the breakpoint occurs, with or without actually stopping the prototype execution.

At any moment in the debugging session, you can ask for a list of all the currently active breakpoints and cancel any of them.

SET BREAK Command

You define a breakpoint with the **SET BREAK** command:

```
Pdb > SET BREAK breakpoint_label trigger_expression
```

The execution is suspended each time the event specified by the `trigger_expression` occurs that is after the step in which the event was generated and before the step in which it is actually sensed.

The breakpoint label is used to refer to the breakpoint in the **SHOW BREAK** and **CANCEL BREAK** commands.

In cases where the label is used more than once, the latter one takes effect, overriding the previous definition of the breakpoint.

The second argument of the **SET BREAK** command is any legal Rational Statemate trigger expression using the same syntax used for transitions. These consist of the:

- Named events and conditions defined in the specification of the prototyped system
- Unnamed basic events and conditions (except the “timeout” “read” and “written” events) referring to the specification’s objects

For example:

```
Pdb > SET BREAK label_1 TRUE(ACTIVE(A))
```

As a result, the execution is suspended each time the activity *A* is activated, while after:

```
Pdb > SET BREAK label_2 ENTERED(S) or
E [ACTIVE(A)]
```

it is suspended when either the system enters the state **S** or event **E** occurs and activity **A** is active at the same moment.

Finally, after:

```
Pdb > SET BREAK label_3 [ACTIVE(A)]
```

the execution is first suspended when **A** is activated, and then, after each successive step in which **A** remains active. This differs from the case of the breakpoint `label_1` above which occurs only when **A** is activated but not in the next steps unless **A** is reactivated.

DO Clause

With each breakpoint, you can associate a sequence of Debugger commands to be performed each time the breakpoint is reached. You can also specify whether you want the prototype execution to pause after performing these commands, or to continue. To define the command associated with the breakpoint, use the **DO** clause when setting the breakpoint:

```
Pdb > SET BREAK label trigger_expression

      DO sequence_of_commands END
```

The **DO** clause can contain any sequence of Debugger commands separated by semicolons.

For example, each time event `e1` occurs, you want to stop and check the current values of conditions and data-items. Instead of retyping in the same commands on each arrival of the breakpoint, you enter them only once when defining the breakpoint:

```
Pdb > SET BREAK bp_1 e1 DO SHOW COND;
      SHOW DATA END
```

Immediately upon reaching the breakpoint, the Debugger suspends the execution and displays the requested values:

```
Stopped at breakpoint BP_1 on event: E1
Current values of conditions:
      chart1:CONNECTED = FALSE
      chart2:NORMAL = TRUE
Current values of data-items:
      chart2:X = 1
      chart3:Y = 2.3
```

It then places the debugging prototype into debug mode. After examining the values, you may perform more Debugger commands.

If you want to perform the **DO** clause without suspension of the execution, you put **GO** as the last command in the **DO** sequence.

For example:

```
Pdb > SET BREAK bp_1 e1 DO SHOW COND; SHOW DATA; GO END
```

differs from the previous one in that the execution is not stopped after displaying the values. You would not be able to enter more commands at breakpoint `bp_1`.

If in a **DO** clause, the **GO** is followed by other commands, they are ignored by the Debugger.

DO clauses can, themselves, set breakpoints. This may result in nesting of **DO** clauses, as follows:

```
Pdb > SET BREAK bp_2 d2 DO SET BREAK bp_3 e3
        DO SHOW DATA x END END
```

There are no restrictions on the depth of nesting.

A breakpoint can be reached only after the end of the step in which its trigger occurred. This is also the point where the **DO** clause is initiated. At this time, all that occurred during the last step is available in the **DO** clause.

At each step, all breakpoints are checked according to the alphabetical order of their labels. Consider, for example, triggers of two breakpoints named *a* and *b* which occurred in the same step. If the first has a **GO** command in its **DO** clause, the second breakpoint is not reached.

SHOW BREAK Command

To see the list of all active breakpoints, use the **SHOW BREAK** command in one of two forms:

- Pdb > **SHOW BREAK**
- Pdb > **SHOW BREAK** `breakpoint_list`

The first form allows you to see the list of all active breakpoints. The Debugger displays each active breakpoint, the corresponding `trigger_expression`, and the **DO** clause, as in the following example:

```
Breakpoint LABEL_1 on event : TRUE(ACTIVE(A))
Breakpoint NEW on event: E1 or E2
Breakpoint LABEL_3 on event: [ACTIVE(A)]
Breakpoint BP_! on event: E1
       reaction: SHOW COND; SHOW DATA
```

The second form displays only selected breakpoints. For example, in response to:

```
Pdb > SHOW BREAK lab*, bp_1
```

the Debugger displays the following list:

```
Breakpoint LABEL_1 on event: TRUE(ACTIVE(A))
Breakpoint LABEL_3 on event: [ACTIVE(A)]
Breakpoint BP_1 on event: E1
    reaction: SHOW COND; SHOW DATA
```

CANCEL BREAK Command

To delete breakpoints that become unnecessary for controlling the prototype execution, use the **CANCEL BREAK** command with a list of breakpoint labels as the argument.

For example:

```
Pdb > CANCEL BREAK label*, new
```

deletes all breakpoints whose name starts with “label” and also the breakpoint “new.”

Note

Canceling a breakpoint does not mean that the effect of its associated **DO** clause is also automatically cancelled.

Consider again the following breakpoint definition:

```
Pdb > SET BREAK bp_2 e2 DO SET BREAK bp_3 e3
       DO SHOW DATA x END END
```

Suppose that `bp_2` is cancelled after it was reached at least once, that is, after its **DO** clause was executed. Then breakpoint `bp_3` remains active, until explicitly cancelled by another command.

Another example:

```
Pdb > SET BREAK bp_4 e4 DO SET TRACE ACTIVITY;
       CANCEL BREAK bp_4; GO END
```

Here, activity tracing is started after the first occurrence of event `e4`, and continues until explicitly cancelled.

To delete all breakpoints, enter:

```
Pdb > cancel break *
```

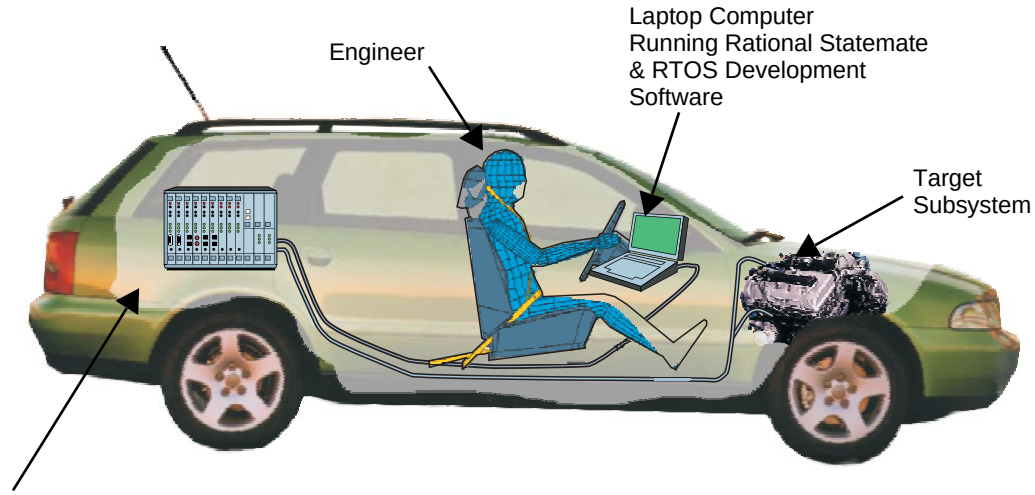
Rapid Embedded Prototyping Basics

Rational Statemate allows designers to graphically model, simulate, analyze and verify the functionality and behavior of complex embedded systems. However, the ultimate verification of any embedded development project is to run the specification in the form of code on a prototype target system, typically a development system designed to allow convenient hardware and software modifications as the project develops. To facilitate this, Rational Statemate has been enhanced to allow the rapid development of code based on the simulated model that can be downloaded into a prototype target development system. The sections in the rest of this manual describe how to use these new Embedded Rapid Prototyping features.

Background

The major reason for Rational Statemate users to perform Embedded Rapid Prototyping is to verify that the model functions properly in a real world environment. Today with Rational Statemate, when a model is tested it must simulate the environment that the system ultimately interacts with. Often the environment is very complex and difficult to completely describe via modeling, test vector files or programs. In the final analysis, the most accurate description of the target environment that insures the accuracy of the specification is only found by bringing the specification (i.e. model) to the actual target environment. This is the essence of Embedded Rapid Prototyping.

The following figure illustrates one such application of embedded rapid prototyping. Here, the engineer is able to run the Rational Statemate software on a laptop computer placed within the passenger compartment of a test vehicle. The laptop is linked to a convenient rack mounted embedded prototyping system located in the back of the vehicle. This embedded development system uses standard CPU hardware and embedded operating system (frequently a Real Time Operating System, or RTOS) with a mix of standard and specialized I/O interface cards connected (by cable or bus) to, and acting as the control components for, some prototype subsystem of the test vehicle.



The Rational Statemate model is used to generate code which is compiled and downloaded to the development system. When executed, the prototype code's features can be observed both in the target hardware and in the Rational Statemate software. When a change needs to be made, the engineer simply changes the Rational Statemate model, regenerates code, compiles and downloads it to the target system. The next test is then ready to begin. This process can be repeated as many times as necessary until the model has been refined to the point where an accurate and complete specification can be finalized.

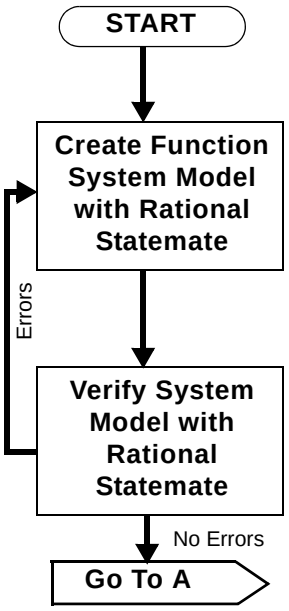
Goals of Embedded Rapid Prototyping

There are three things that must be verified when running such an embedded prototype:

- The basic specified functionality is correct for the target system.
- The interaction between functions is correct.
- The time lines for the execution of these functions is correct.

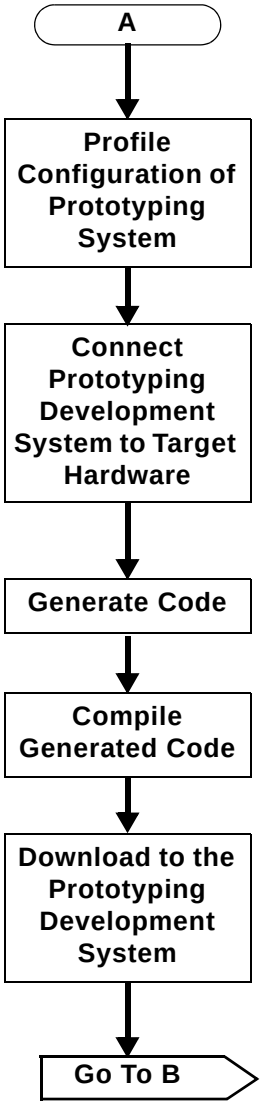
Embedded Rapid Prototyping Process Model

The Process Model for embedded rapid prototyping is slightly different than the classic Rational Statemate model. The following figure shows the process as a flow chart. Each step in the flowchart is described below.



1. Modeling and Simulation - First you create a functional system model using Rational Statemate. Next, you simulate this model to verify that the functionality is correct. If during simulation any errors are found, you correct these errors in the model and re-simulate. This simulation/model correction loop continues until you are satisfied that the model functions correctly.

2. Embedded Rapid Prototype - The code and prototyping unit now is hooked up to the target hardware (or in some cases a test bed).

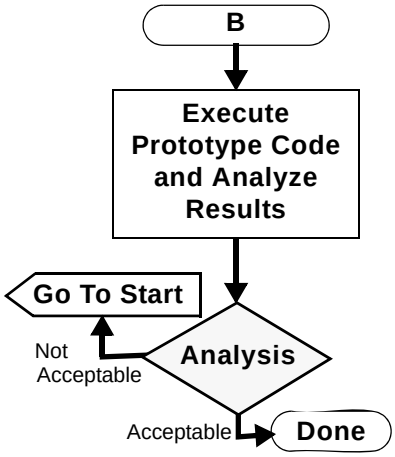

A
Profile Configuration of Prototyping System
Connect Prototyping Development System to Target Hardware
Generate Code
Compile Generated Code
Download to the Prototyping Development System
Go To B

3. At this point in the process, you next move to verify this functionality in an environment that is closer to the actual product environment than that which was used for simulation. This requires the use of a hardware/software prototyping system which can be hooked up to the actual target product's hardware and act as its control system. See the sidebar discussion in **The Embedded Prototyping System** regarding embedded prototyping systems.

   a. The first step in moving to the prototyping unit is to capture information (target O/S, I/O mapping) which is specific to the prototyping unit.

   b. Next, code is generated from the model.

   c. The code and prototype unit information is then compiled and downloaded to the embedded rapid prototyping unit.



4. Analysis is performed on the execution runs to verify that the functionality, behavior and timing for executing the functions are correct.

The Embedded Prototyping System

The embedded prototyping system will vary from product to product (and even from project to project) but will typically consist of a processor-based hardware development system. Such a system requires four major elements:

- boot source code
- device drivers
- some I/O interfacing capability
- custom software needed to interface the application code and real time operating system (RTOS) to the target hardware

Such systems may be rack mounted in a bus-based card cage, or be self-contained on a single board computer with I/O components added/ modified as necessary. It could even be a totally custom development system. The Board Support Package (BSP) is the name typically given to such a prototyping system because it encompasses more than just the development system hardware.

If any changes need to be made, the model is updated and resimulated as needed. Code is re-generated, compiled and downloaded to the embedded prototyping system. This updated prototype is then again executed and analyzed. This iterative cycle continues until the user is satisfied with the results of the test runs. At that time the specification (based on the model) will be handed off to software designers for implementation of the production version of software. This code can be tested against the test criteria generated from the model to verify that it meets functional specifications.

The next three main sections will examine each step of this process in detail, as well as related design issues. **Simple Embedded Code Example** presents a example of a small rapid embedded prototyping project to demonstrate how all of these elements come together.

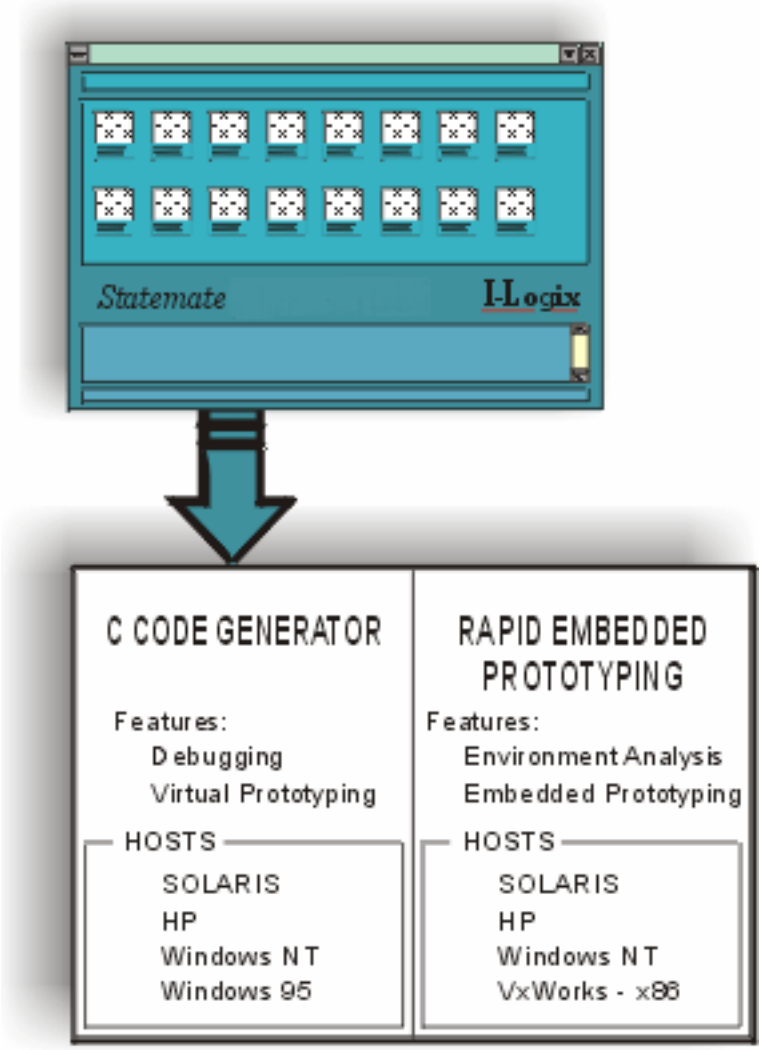
Embedded Rapid Prototyping in Rational Statemate

Rational Statemate implements four major features relating to embedded rapid prototyping:

- Retargetable to different target platforms
- Flexible signal mapping to any I/O card
- Target trace facilities for debugging
- Remote connection to different support tools such as:
  - Panels
  - Graphical Back Animation (GBA)

Note

Embedded Rapid Prototyping is essentially an extension to the standard Rational Statemate C code generator (refer to **Generating Native Code**) All other standard features of Rational Statemate remain unchanged.


Statemate
I-Logix
C CODE GENERATOR
Features:
Debugging
Virtual Prototyping
HOSTS
SOLARIS
HP
Windows NT
Windows 95
RAPID EMBEDDED PROTOTYPING
Features:
Environment Analysis
Embedded Prototyping
HOSTS
SOLARIS
HP
Windows NT
VxWorks - x86

Target Requirements

The most significant variable in any embedded application is the nature of the target hardware and its operating system. Because the Rational Statemate rapid embedded prototyping capability is designed to work with many different target hardware/OS combinations, it is necessary to communicate the nature of the target hardware/OS so that appropriate code can be generated. This section examines how to communicate this information to Rational Statemate.

The many possible configurations of target system hardware and software requires a means to configure Rational Statemate to the specific details of the system used. The following items are critical to successfully generate and compile working code:

- RTOS boot file
- Driver software
- I/O port assignments
- Terminal communications capability to monitor test system's operation

Other items may be necessary for a specific prototype, but these are required by virtually all systems.

To configure Rational Statemate for this kind of information requires three specific items:

- A communications link for data upload/download between the Rational Statemate host system and the prototype development system; commonly an ethernet link, an RS-232C link, or an RS-485 link
- A target file that specifies details of the target RTOS and hardware environment
- An I/O file that specifies the configuration of all I/O port assignments

Describing Different Target Platforms

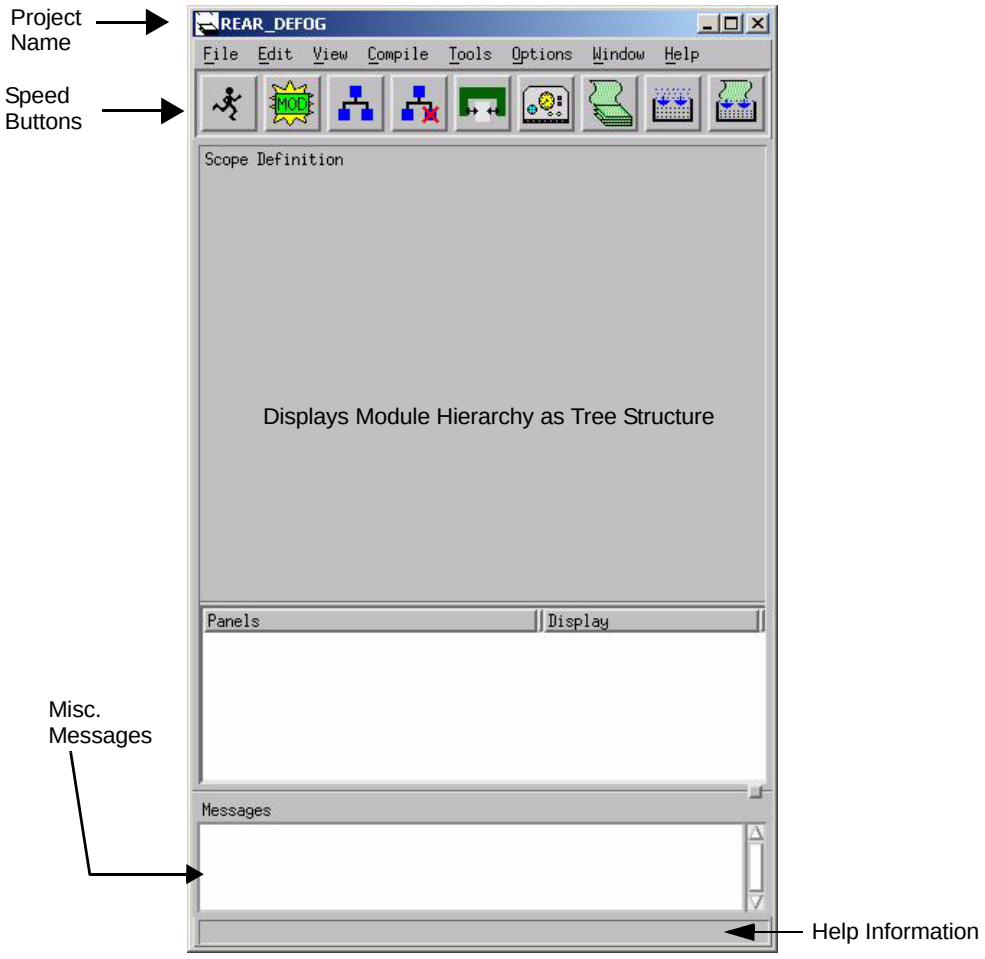
The Embedded Rapid Prototyping Code Generator supports an open set of user defined targets. All target-related parameters are defined and maintained in an ASCII file named `<OS name>.rtrg` in the `prt/rprt` directory. Files are currently provided for the following targets:

- Unix
- Solaris
- Windows
- VxWorks

This selection of the target platform and all other parameters for a specific project are created and maintained using the Rapid Prototyping Compilation Profile Editor.

To start the editor, click this  icon, found on Rational Statemate's main screen.

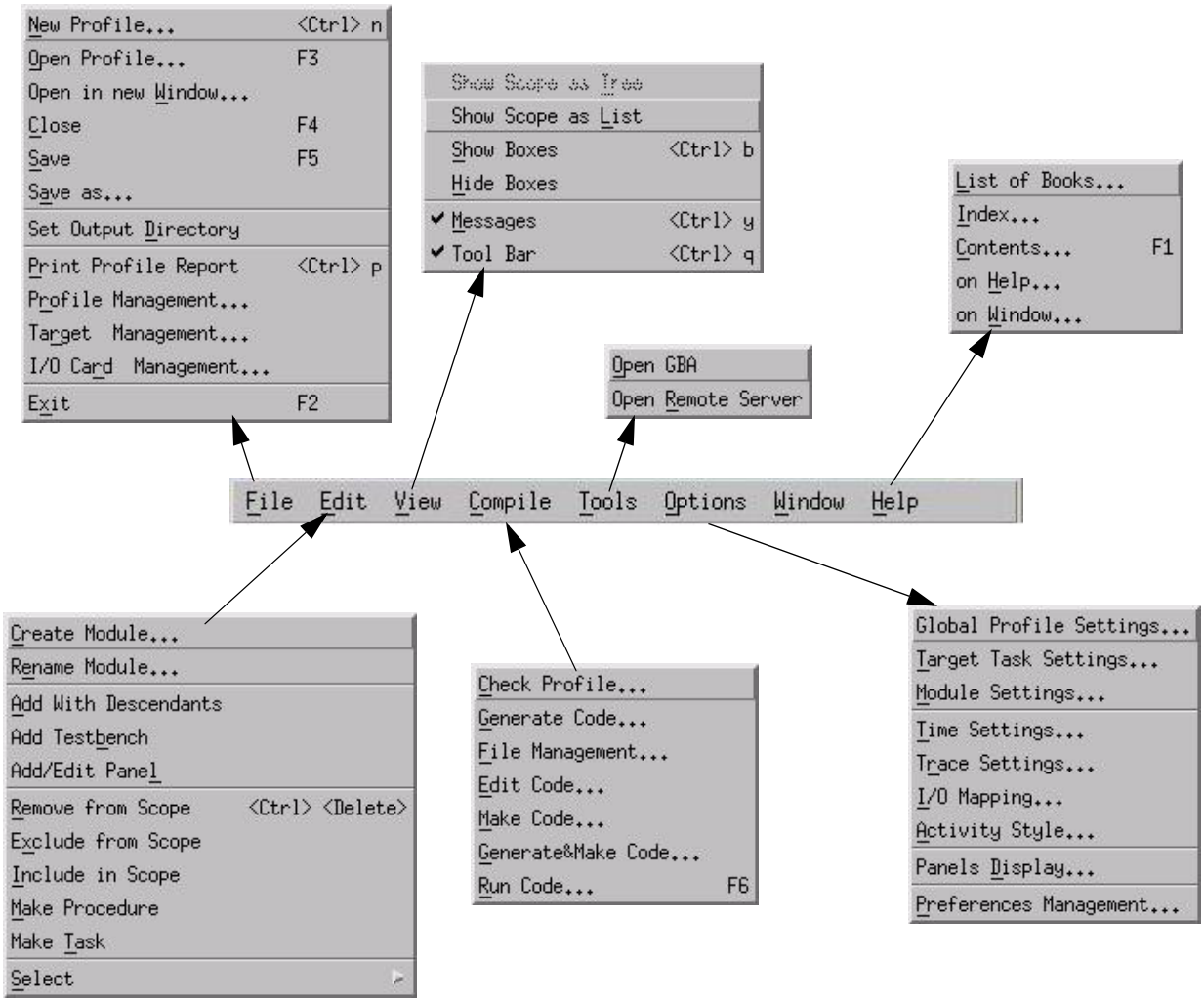
The following is an example of the Profile Editor screen.



Compilation Profile Management

Compilation Profiles for embedded rapid prototyping projects function in much the same way as they do in normal C and Ada projects (refer to **Generating Native Code** and **Ada Code Generation** for more information). Descriptions of most of the menu items and speed buttons are provided in the on-line help files. A summary of the Profile Editor's menus is illustrated in the following figure.

Typically, Compilation Profiles must be tailored to the requirements of a specific target system and the prototyping system interfacing to it. Many of these parameters may be standardized and (if this is the case) should be available from your project leader. Otherwise, you will need access to information relating to the exact configuration of the prototyping system and the target hardware.


New Profile... <Ctrl> n
Open Profile... F3
Open in new Window...
Close F4
Save F5
Save as...
Set Output Directory
Print Profile Report <Ctrl> p
Profile Management...
Target Management...
I/O Card Management...
Exit F2
Show Scope as Tree
Show Scope as List
Show Boxes <Ctrl> b
Hide Boxes
Messages <Ctrl> y
Tool Bar <Ctrl> q
Open GBA
Open Remote Server
List of Books...
Index...
Contents... F1
on Help...
on Window...
File Edit View Compile Tools Options Window Help
Create Module...
Rename Module...
Add With Descendants
Add Testbench
Add/Edit Panel
Remove from Scope <Ctrl> <Delete>
Exclude from Scope
Include in Scope
Make Procedure
Make Task
Select
Check Profile...
Generate Code...
File Management...
Edit Code...
Make Code...
Generate&Make Code...
Run Code... F6
Global Profile Settings...
Target Task Settings...
Module Settings...
Time Settings...
Trace Settings...
I/O Mapping...
Activity Style...
Panels Display...
Preferences Management...

Creating the Profile

Each of the following tasks must be performed to create a successful profile.

Scope Definition

To define the scope, the user must declare the modules, add the charts, and add the panels into the scope using the workarea browser.

Target Definition

The list of targets contains the `.rtrg` file names detected in the `<workarea>/rprt` directory. If it is necessary to add, replace or delete some flags, libraries, compiler name, etc. or add custom modules (e.g. objects or/and libraries), you must change the `.rtrg` file appropriately.

For example, an I/O card driver object file may be added into the `#Intrinsics library:` paragraph. It would then appear something like:

```
#Intrinsics library:"$(STM_ROOT)/lib/VxWorks/libintrinsics$(CPU).a :/tmp/io/
rprt/onyx.o"
```

If the full path of the target output directory differs from the host output directory path (for example, if the host directory is Unix-like `/stmw.qa/qa_20` and the same target directory on Windows is `q:\qa_20`), you would need to fill out the **Target Directory** field of the Global Profile settings dialog form.

Using Remote Panels

It is possible to use remote panels by setting the *toggle button* **With Remote Panel Server** on the Global Profile Settings dialog form.

Note

Resetting this button does not cause deletion of panel(s) from scope.

Input/Output Mapping

I/O mapping is the main feature of the Rapid Prototyping Code Generator. It allows you to map the textual elements of a Rational Statemate model into the input/output signals of I/O card.

The mapping process consists of the following steps:

1. I/O card description file creation
2. I/O card driver functions creation
3. I/O mapping
4. Polling rate selection
5. Input card task(s) parameters set

The two first steps are usually made only at the beginning of the process. The other steps may be performed every time, when you want to change something in the model or in run time execution of the model.

Writing the I/O card description and driver code are correlated processes. In fact, the `.crd` file is input information for the driver functions, so the content of this file depends on the needs of the driver. For example, the *port offset* field contains some string. In the example (the `onyx_mm_dio.crd` file shown in the following figure), every element of the `#port list` paragraph corresponds with some port of the card. However, it is sometimes only necessary to take one bit of the real port and to map it to some condition. In this case, it is more comfortable to declare every pin as a separate virtual port and to use the `#port offset` field to declare the two information elements: *real port offset* and *pin number of the port*. Then the driver can unpack this field and read the two values as specified. It might look like the following string:

```
#port offset:"1:5"
```

where 1 is the port offset, and 5 is the pin number.

Although such a driver tends to be a little more complicated, this approach sometimes makes sense because it allows for the simplification of the model.

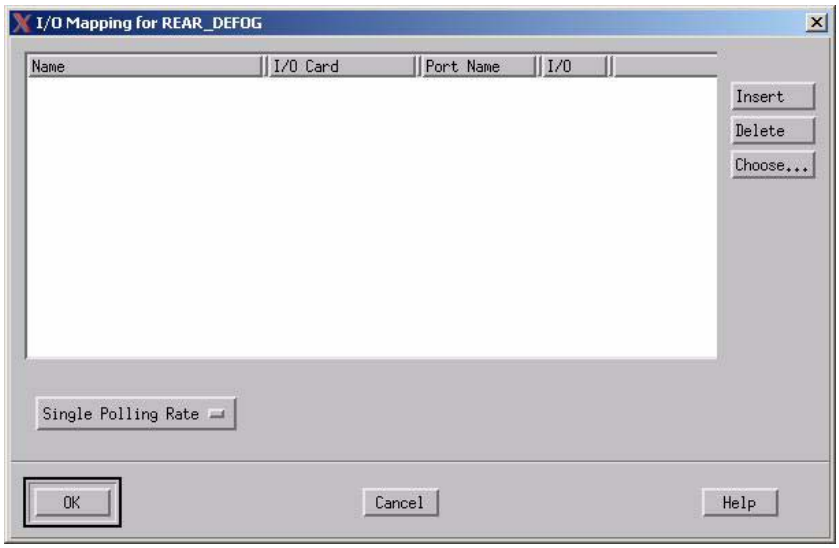
Example of the Driver Functions

```
The following is the real driver functions which are implemented
  to support the ONYX–MM-DIO I/O card from Diamond Systems Corp.


#include <vxWorks.h>
#include <stdio.h>
#include <stdlib.h>
#include <syslib.h>
#include "types.h"
#include "symbols.h"
#include "string.h"

#define DIO_1A    0
#define DIO_1B    1

#define DIO_1C    2
#define DIO_1CR   3

#define DIO_2A    4
#define DIO_2B    5
#define DIO_2C    6
#define DIO_2CR   7


int onyx_base_addr;  /* The card base address , converted into int
  format */


/* -- generic card initializer for both input and output mapping  */
void  onyx_init(card_desc_p card_p)
{ int addr;
sscanf(card_p->base_addr,"%x",&onyx_base_addr);
printf ("base_addr = 0x%x\n",onyx_base_addr);
sysOutByte(onyx_base_addr+ DIO_2A,0x00); /* Reset of output
  registers before setting of 2A port to be output  */
```

```
sysDelay();
addr = onyx_base_addr+DIO_1CR;/* control register 1CR address
  */
                /*  Port 1A , 1B and 1C set to OUTPUT/MODE 0  */
sysOutByte(addr, 0x80);
printf ("Ports 1A , 1B and 1C set to OUTPUT/MODE 0\n");
sysDelay();

addr = onyx_base_addr + DIO_2CR;  /* control register 2CR address
  */
sysOutByte(addr,0x9B); /*   Port 2A , 2B and 2C set to INPUT/MODE 0   */
printf ("Ports 2A , 2B and 2C set to INPUT/MODE 0\n");

}

/* -- generic card driver for both input and output mapping  */
void  onyx_in(report_link elem)
{
  genptr  new_value = (genptr)elem->received_val;
  int     b, offset;

  sscanf(elem->pin_offset,"%d",&offset);
  b = sysInByte(onyx_base_addr+offset);   /* input from 2A port */

 switch(elem->elem_type) {
  case el_integer:
  case el_enumer:
  case el_bit_array:
      *(int*)new_value = b;
      break;
  case el_condition:
  case el_event:
      *(char *)new_value = b;
      break;
  case el_real:
```

```
  case el_real:
      *(double*)new_value = b;
      break;
  default:
      *(int*)new_value = 0;
      break;
  }
}

#define OUTPUT_BUF_SIZE   16
double  output_buf;

void onyx_out(report_link elem)
{
  void *actual_val = (void*) &output_buf;
  int offset;
  sscanf(elem->pin_offset,"%d",&offset);

switch(elem->elem_type) {
  case el_integer:
  case el_enumer:
  case el_bit:
  case el_bit_array:
  if (elem->pin_inverse)
      (*(int*)actual_val) = ~(*(int*)elem->elem_value);
    else
     (*(int*)actual_val) = (*(int*)elem->elem_value);
    sysOutByte(onyx_base_addr+ offset, *(char*)actual_val);   /*  a=> output
  to 1A port */
    break;
  case el_real:
    (*(double*)actual_val) = (*(double*)elem->elem_value);
    …
    break;
```

I/O mapping itself is performed using the I/O Mapping dialog form shown above in the following figure. It contains the mapping matrix and the Polling Rate pop down menu. Note that there are four fields in every line of the matrix.



The name is selected by using the **Choose** window or by typing in the name directly from the keyboard.

The I/O card field may be filled out by placing the mouse cursor over the cell in the desired row and column and pressing the right mouse button. Then, a pop-down menu appears containing the current list of `.crd` files detected in the `<workarea>/rprt` directory.

When the card name is chosen, the user can choose the port name in the next field of the matrix. This is done in the same manner as the previous field. The pop-down menu presents the list of available ports.

When the port name is chosen, the I/O field is automatically filled out by the proper value for this port. Usually there is no need to change it, because this value is an inseparable part of the I/O card and is installed in the `init_driver` function or built-in.

After the I/O mapping is done, it’s time to set the parameters of the INPUT tasks, (if input mapping exists).

You can define one or two tasks for the polling of I/O cards. The second task is needed only if there are more than a few I/O cards, all participating in the input mapping, and with different polling rates. In this case, the code generator separates all of the I/O cards into two groups: one group with high a polling rate and the other with a low polling rate. The mapped elements belonging to the first group are polled by the `HINPUT_TASK`, and the mapped elements from the second group are polled by the `LINPUT_TASK`. The actual polling rate values can be set using the **Target Task Settings** window shown in the following figure.

Detailed View of I/O Card Description File

Key Word	Type of Possible Values	Description
#card name	string	The name of the I/O card.
#card polling rate	integer	The size of time interval (in ticks) that should pass between two consecutive card read operations.
#card number of ports	integer	The number of ports in the card. The port list should contain exactly this number of elements.
#card base address	string	The hexadecimal string defining the real address of the I/O card on the bus. For example: “0x240”.
#card init function	string	The name of the function that initializes the I/O card. It is called once when the program starts
#card driver function	string	The name of the driver function that actually reads data to, or writes data from the card.
#card closure function	string	The name of the function that shuts down the card when it is no longer needed. This routine performs any housekeeping tasks that the I/O card might require.

Key Word	Type of Possible Values	Description
#Port list	list of strings	The list of port descriptors. Fields that constitute the port description are described below.
#Port name	string	The name of the port, it appears in the GUI when users have selected a card in the I/O mapping settings dialog, and they are choosing the port for each I/O mapped model element.
#Port inverse logic	yes/no	If yes, the value read from or written to the port will be inverted bitwise. If no, the card driver function reads the value directly.
#Port default mode	in_mode/out_mode	The default input/output configuration mode of the port.
#Port default buffer	yes/no	If yes, the output to the port is buffered.
#Port offset	string	The offset of the register, relative to the port's base address. It is expressed as a hexadecimal string.

Trace Settings

The trace settings option enables you to trace data items of basic type, conditions, events, states and activities without using the Rational Statemate debugger in a less intrusive manner.

The selection of the traced elements is done using three matrices in the Trace Settings window of the Options menu.The trace of every selected element can be turned off or on.

In addition, there is a pop-down menu that sets the format of the trace file or disables it. There are two formats of trace file:

- Compact format
- PDB-like format

You can supply your own function, which will be called every time that the traced element is changed. It allows creation of the trace file in the format appropriate to different tools.

The tracing text data is written to the `<output directory>/<profile name>.trc` file by the Remote Server that receives the messages from the `TRACE_TASK` via TCP/IP socket communication, which usually has lower priority than the other tasks. It reads the trace lines from the buffer, where they were put during execution between two sequential steps in the `TRACE_TASK`. It means that the buffer size, which the user can change, should be big enough. The buffer size is set in the **Target Task Settings** window.

In fact, polling rate, priority and buffer size are interconnected. If the priority is low (e.g. 255 is the lowest available priority for VxWorks), the buffer size should be as big as possible. If the polling

rate is big (i.e. it is a number of ticks of the delay between two sequential steps in a task loop), the buffer size and the priority should be higher.

The run-time trace process is controlled by two conditional expressions: `start trace` and `stop trace`. The first is evaluated at the beginning of the step in model, and the second at the end of the step. The empty start trace field is equal to TRUE. The empty stop trace field is equal to FALSE. This makes the trace continuous.

Target Task Settings

The new dialog can be started by the **Target Task Settings…** button.

Every line in this dialog corresponds to one task. Each task is created to support one of the following features:

- Remote Panel client task
- Remote GBA client task
- Trace task
- One or two Input Card task(s)

Each line contains the task name, the priority, the polling rate and the buffer size fields.

Any of these lines will appear/disappear if the proper feature is enabled/disabled:

- With Remote Panel Server
- Graphical Back Animation (GBA)
- Trace Enable (and there is at least one item to be traced)
- Input mapping list is not empty

Note

If the Single Polling Rate is set, only one task will appear in the dialog list; otherwise, two tasks will be created.

You can establish the priority, polling rate, and the buffer size for any of these tasks. But be very careful, because improper settings may cause unpredictable behavior of the generated code. For example, if the priority of one of the tasks is too high, the other tasks won't work and the generated code will hang. Too small a buffer size may cause the buffer to overflow and physically adjacent data would be lost.

Target Management

This section describes the target description file in detail. Some of the keywords are actually strings that basically are copied to the makefile created along with the generated code. Their type

is “makefile string.” The user of this feature should have certain basic knowledge about Makefile language.

In particular, if the user wants to put $STM_ROOT as part of the value of string or makefile string keywords, the $ (for a Unix target) sign would be duplicated, like “$$STM_ROOT” or write “$(STM_ROOT).”

Some of the keywords contain OS paths. The user should be aware about proper directory separator character.

Key Word	Type of Possible Values	Sample Value	Description
#Link command	makefile string	`"LINK = $(CC)"`	Linker command on the target OS
#System libraries	makefile string	`"SYS_LIBS = -lm"`	Standard system libraries for the target OS
#Library extension	string		Extension of precompiled library files
#Executable extension	string		Extension of executable files
#Output file keyword	string		Name of the parameter that toggles the name of the output file for target OS C compiler
#Intrinsics library	string		Where the Rational Statemate intrinsics libraries are put in the given Rational Statemate installation
#Scheduler library	string		Where the (Statemate) scheduler libraries are
#Simulated scheduler library	string		Where the (Statemate) simulated scheduler libraries are
#Debugger library:	string		Where the (Statemate) debugger libraries are
#GBA library	string		Where the Rational Statemate GBA library is put in the given Statemate installation
#Panel library	string		Where the Rational Statemate Panel library is put in the given Statemate installation
#Additional libraries	string		Additional, perhaps user-supplied, libraries that should participate in the linking of final executable

Key Word	Type of Possible Values	Sample Value	Description
#Object extension	string		Extension of object file on the target OS
#Archiv command	string		Command that should be run when building the output_lib.a (on UNIX) library
#File deleting command	string		Self explanatory. It is important that this command is not supposed to be interactive.
#Make command	makefile string	`"$(MAKE) -f"`	Self explanatory
#Main file directory	string		Name of the directory where the final executable should be put
#CPU name	string		Self explanatory
#Ranlib command	string		Self explanatory
#K&R C compiler name	makefile string	`"CC = gcc"`	Self explanatory
#K&R C compiler flags	makefile string	`"CFLAGS = -g"`	Self explanatory
#ANSI C compiler name	makefile string	`"CC = gcc"`	Self explanatory
#ANSI C compiler flags	makefile string	`"CFLAGS = -g"`	Self explanatory
#Link flags	string		Linker-specific flags
#ADA compiler name	string		Self explanatory
#ADA compiler flags	string		Self explanatory
#ADA linker name	string		Self explanatory
#ADA linker flags	string		Ada linker specific flags
#Download script name	string		Name of the script/command that should be run to download the final executable to the target OS
#Remote exec name	string		Name of the script/command that executes the final executable on the target OS

Note

The internal double quote character should be replaced by the single quote character. For example, the line **“ /D “PRT”** should be replaced with **“ /D ‘PRT”**.

You can manually change or create a new `<target name>.rtrg` file based on your project's specific target prototyping development system.

Only the structure of the file and the names of the paragraphs should be unchangeable. So, the best way to change or create a new `*.rtrg` file is to copy the current file under a new name (e.g. the name of your target) and to change only the values in each line of the file that is affected by the new target.

Note

Each line of the `*.rtrg` file is terminated by a "hard return," which means that each line is effectively a paragraph. This is important to the accurate parsing of the `*.rtrg` file.

Every line in the `*.rtrg` file affects a certain part of the `makefile`, which is created during code generation, or the running of the generated/compiled code.

Pay attention to the `#Run script name` line. Its value, if it is not empty, concatenates with the name of executable and runs as a shell command after the user selects the **Run Code** option of the **Compile** menu. So, users can write their own batch files that will take as a parameter the name of the executable and do any required operation. Such a technique is used to download and execute the generated code in a remote manner (refer to **Downloading and Execution**)

The content of the `vxworks.rtrg` file is listed for reference in **Target Description File**.

It is necessary to define the nature of the Target for the prototyping system. The initial step in this process is done using the dialog screen shown in the following figure. It is started using the **File** > **Target Management** menu item from the Profile Editor. The Target Management window opens.

There are several predefined targets presented in this dialog screen for you to choose from. Select the desired target by clicking on it. The selection is highlighted.

Note

No special license is required for targeting to the different platforms.

Because the rapid prototyping code generator supports an open set of user-defined targets, the list of supported targets can be supplemented with new targets or variations of existing targets. This is possible because each target selection corresponds to a target definition file using the naming format `<OS name>.rtrg`. All of the target related parameters are defined in that file. The buttons along the right hand side of the dialog screen allow for management of this file.

Advanced users or project teams may decide to modify the target description files. Always work with a copy of a working file if at all possible. Save it under an appropriate name and document your changes so you can back up if something doesn't work like you expected it to.

Note

The target definition file is similar to the file used for the standard C Code Generator, which uses a different extension (**.**trg). The file is an ASCII text file.

Note that the targets listed here are the files' prefix. All of the targets listed are those that are found (i.e, `*.trg/*.rtrg` files) in the `prt/rprt` directory.

I/O Card Description File Management

The **I/O Card Management** window of the **File** menu should be run for this purpose. For the description of the file structure, refer to the previous table and the sample code in **I/O Driver Functions.**

Describing Signal Mapping to I/O Cards

The embedded rapid prototyping code generator supports mapping of basic data items and conditions, relevant to the current scope, to an open set of user defined I/O locations in the prototyping hardware. There is no support for events, user- defined types, array elements, records and fields in a record (except for enumerated types, which are regarded as integers).

The definition of an I/O card is done using an ASCII file, with the extension ". crd". This .crd file contains the definition of the I/O card's configuration, including:

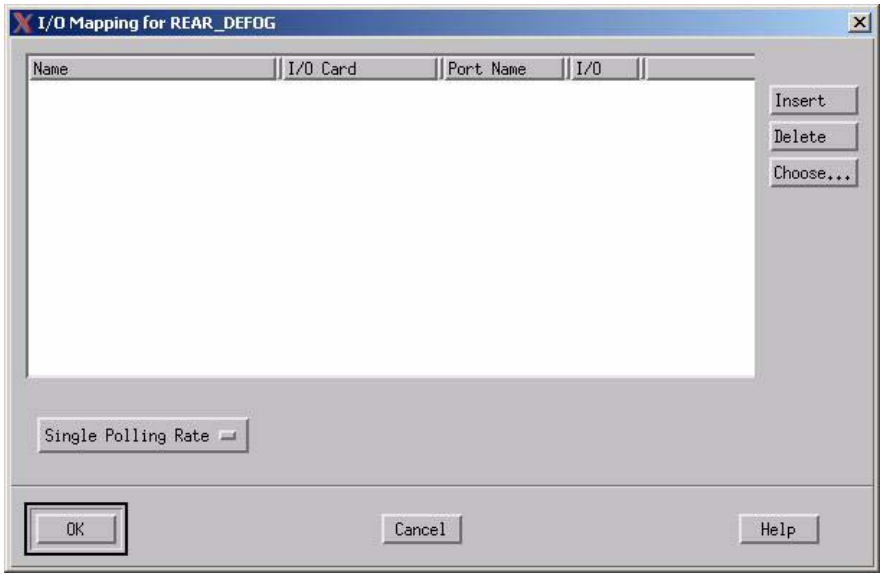
- available channels or ports
- card base address

- channel/port offsets
- driver function name
- initialization function name

  Note: It is assumed that an I/O card will require initialization in order to set it up for the desired port configuration, control mode, data handling, etc.

- closure function name
- minimal polling rate

The compilation profile editor's Options menu lists a menu item for I/O Mapping. This menu item invokes the I/O Mapping dialog (see the following figure). This dialog presents a *mapping matrix* that is used to identify the basic identity and type for each I/O card included in the system. Using a selection mechanism similar to that used in the Simulator, this mapping matrix guides the user in selecting data items relevant to the current scope and saves selected filled lines in the `<profile name>.rgenset` file. It does this by displaying a drop-down list of available selections when you right-click on the matrix cell for a given row and column.



As previously discussed, the mapping matrix dialog contains the following fields:

Name	String Describing the Name of the Port
I/O card name	Displays all of the files named `*.crd` in the `<workarea>/rprt` files directory
Port	Displays all of the available ports described in the selected card

I/O	Specifies the In/Out mode of the port

Note

A single I/O card can have multiple ports. Each should be assigned a meaningful name.

At the bottom left corner of the window is a button labeled **Single Polling Rate** allows selection of one of two modes:

- Single polling rate
- Double polling rate

The definition of the polling rates is done using another menu item, **Files >I/O Card Management**.

Signal Mapping to I/O: Semantics

There are two issues here: one is inputs, the other outputs.

- **Inputs**:

  Whenever a signal is mapped to an input port it is regarded in a function named `top_do_inputs()`**,** if the Single polling rate is selected, or in two functions named `top_do_high_inputs()` and `top_do_low_inputs()`, if the Double polling rate is selected. In case of two polling rates, each input signal is handled only in one of `top_do_high_inputs()` or `top_do_low_inputs()` according to the polling rate specified in the card definition file:

  - All input signals whose polling rate is lower than the high polling rate is handled in the `top_do_low_inputs()`function.
  - Others, whose polling rate is larger or equal to the high polling rate, are handled in the `top_do_high_inputs()` function. Those functions are started from separate tasks (one for `top_do_high_inputs()`, the other for `top_do_high_inputs()` in case of two polling rates, and one task in case of single polling rate). The new values are in effect in the following step.

- **Outputs**:

  There are three categories of data items:

  - Whenever a **non-Double Buffered** element is assigned with a value, a call to the output device is done immediately.
  - Whenever a **Static Double Buffered** element is assigned with a value, a call to the output device is done at the end of the current step. The call will be from the generated code, near the place where it swaps the next/current values. In order to trace the writing event, a flag will be added to the generated code, near the definition of the current/next variables.
  - Whenever a **Dynamic Double Buffered** element is assigned with a value, a call to the output device is done at the end of the current step from the update() function inside of RT library.

Target Trace Facilities: Description

The rapid prototyping compiler supports tracing of basic primitive data items, conditions and events. The user-defined types, array elements, records, and fields in a record are **not** supported. Enumerated types are regarded as integers.

Tracing is done through a buffer, meaning that the code, while running, sends report text to a buffer. That report buffer is automatically flushed to the `<profile>.trc` file in the output directory.

In the compilation profile editor, the menu item **Options** > **Trace Settings…** invokes the Tracing dialog.

The Tracing dialog contains the following items:

- Trace format
- Trace for: States/Textual Elements/Activities
- Start Trace text field
- Stop Trace text field

The trace file `<profile_name>.trc` is written into the current directory where the generated code is running. Its format depends on what the user has selected, such as:

- Compact format
- PDB-like format
- User-supplied format

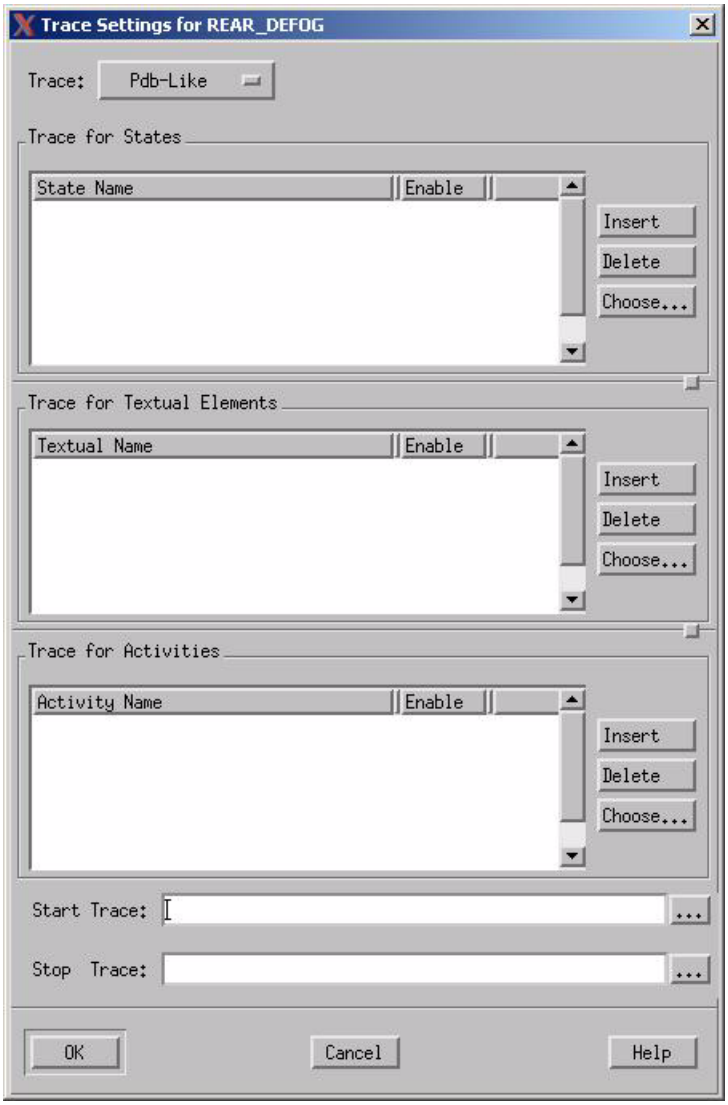
In the latter case, the user defines the name of the trace function that is called every time the traced element value changes. The function has the following definition:

```
char  * user_func (report_link  elem).
```

It returns a text message string which is then output to the `<profile_name>.trc` file.

Both the **Start Trace/Stop Trace** fields are for defining trigger expressions. The goal here is to support compound expressions that are composed of panel bindings-like basic elements.

Using a selection mechanism similar to that for the Simulator, right-clicking on a row/column field results in a pop-up list of available selections relevant to the current scope.



Each row contains two fields:

- An editable text string identifying the data item name
- The trace mode (trace of the element is *enabled* or *disabled*)

Note

It is also possible to start/stop tracing using two API functions: `enable_trace()`, and `disable_trace()`.

Target Trace Facilities: Semantics

Trace messaging is implemented is similar to the mechanism applied to the I/O output reporting. Whenever the tracing is **On**, meaning the **Start Event** occurred, a report line is added to the report buffer and subsequently output to a text file:

- If a **non Double Buffered** element is assigned with a value, the trace will be done immediately.
- If a **Static Double Buffered** element is assigned with a value, the trace will be done at the end of the current step. The trace function will be called from the generated code, near the place where it swaps the next/current values. In order to trace the writing event, a flag is added to the generated code, near the definition of the current/next variables.
- If a **Dynamic Double Buffered** element is assigned with a value, the trace function will be called at the end of the current step from the `update()` function.

Data Types Introduced to the Intrinsics Library

This section includes the following information:

Data Types Related to the Data Items

For each element that is either traced or has I/O mapping associated with it, the instance of the following data structure is generated:

```
typedef struct report_elem {
  /* -- general */
  char*        elem_name;  /* -- name of the reported element */
  char*        inst_name;  /* -- name of the generic instance */
  genptr       elem_value; /* -- pointer to the value of the
                           -- __model__ element */
  el_enum      elem_type;  /* -- type of the reported element */
  int          str_len;  /* -- if non-zero - length of the
   string */
  genptr       user_data;  /* -- user defined misc data */
  /* -- tracing */
```

```
  rep_funcp    trace_func;   /* -- trace function pointer */

  boolean      immediate;    /* -- if TRUE bypass the list */

  /* -- input/output mapping */

  /* -- common for both in and out */

  card_desc_p  card_p;       /* -- the card associated with this model --
element */

  char*        pin_offset;   /* -- offset off the base */

  char*        pin_name;     /* -- name of the pin */

  int         pin_number;  /* -- pin position within pin
    array of card_p, starting
    from 0 */

  /* -- input mapping specific */

  rep_funcp    in_func;      /* -- for input mapping */

  genptr       received_val; /* -- buffer for storing the new value of -- the
model element received from the -- card */

  genptr       new_value;    /* -- double buffering info: -- if ZNIL then DB
is dynamic -- else the read value is directly -- written to *new_value */

  /* -- output mapping specific */

  rep_funcp    out_func;     /* -- for output mapping */

  boolean      pin_inverse;  /* -- if TRUE inverse the reported value bitwise
*/

  boolean      buffered;     /* -- if TRUE output is buffered */

  /* -- list management */

  report_link  next_elem;    /* -- next element of the list of reported
elements */

} report_elem;
```

This declaration is copied from `types.h` of the Intrinsics library. The variable of this type is called “report element”. It contains all information about data item and its tracing and I/O mapping specifics that at the moment seems necessary. The purpose of `user_data` field is to provide users with the capability to add whatever data they desire. It is important because report elements are visible to the code that should be written by the user.

Each report element that is associated with a pin of a I/O card has its `card_p` field non ZNIL. This allows Rational Statemate to serve I/O mapping requirements in a per-element fashion.

Report Elements for Output Mapping and Tracing

For each place where data item is changed, Rational Statemate generates the following instrumentation: { X = 5; add_report(&rep_X); }. By calling the `add_report()` function, the report element corresponding to X to the list of elements that were changed during the current step is added. At the end of this step, a special function passes the list and calls functions that do the actual tracing and/or out mapping. If immediate field of report element is true, the `add_report()` function bypasses the list mechanism and calls immediately to the function that does the tracing.

If at the current step the tracing is switched off, the `add_report()` function does nothing. The list of the report elements is static; that is, there is no dynamic memory allocation. The list management is done statically by manipulating the `next_elem` field of the report element.

All buffered tracing messages are sent to the trace buffer. The buffer has a fixed, predefined size. The same moment the buffer becomes full, its contents are out and cleaned up. The same is done at the end of each step, regardless of whether the buffer is full. This process is run in the low-priority task.

In output mapping, the buffered field of the report element dictates the way the output message is handled. Either it is buffered in the manner similar to tracing, or it is sent to the output card immediately.

Report Elements for Input Mapping

If there are model elements that require input mapping, Rational Statemate generates the tree-like structure of input mapping functions that is similar to the tree of `init()` functions in the model. Each such function calls to the `in_func()` field of those report elements that belong to the current scope.

There are three fields of report element that control the input mapping mechanism:

- **elem_value** - The pointer to the actual value of the element. It is used in the input mapping if the model element that corresponds to this report element is dynamically double buffered and the `set*()` function must be called.
- **new_value** - The double buffering information. If this pointer is ZNIL, the model element is dynamically double buffered. Otherwise, the value received from the card is directly assigned to the variable in the code referenced by this pointer.
- **received_val** - The buffer where the data read from the card is stored until it is assigned to the model element. This field is part of API between user-written card driver and generated code. The card driver should put there the value that it reads from the card.

Report Elements for Generic Charts

If a traced or I/O mapped element is passed to some generic chart as a parameter, its report element is also passed to the generic chart as well. In this case, an additional parameter of the generic chart is generated. All other attributes of data item in the generic are generated for the report element as well. This includes a macro for accessing the report element, its declaration, and so on.

Suppose now that user decides to do tracing or I/O mapping for some local variable in some generic chart. In this case, the user must provide the full name of the data item (including the instance name) for every instance of the generic in the model. In this case, a report element is generated within the generic and the context-switching mechanism ensures that each instance has a separate report element. The generated code ensures also that such report elements are initialized separately for each instance of the generic.

Data Types Related to I/O Cards

The following data structure describes the I/O card:

```
typedef struct card_desc_elem {
  card_drvp     card_drv;    /* -- card driver function */
  card_funcp    card_init;   /* -- card init functions */
  card_funcp    card_close;  /* -- card closing function */
  char*         base_addr; /* -- base of the target memory
                              location */
  report_link*  pin_array; /* -- array of pins of this card/
                             array of associated model
                               elements */
  genptr        user_data;   /* -- user-defined misc data */
} card_desc_elem;
```

This declaration is also copied from `types.h` file of the Intrinsics library. The variable of this type is called “card element”. It contains all the information about the I/O card. The purpose of the `user_data` field is to allow users to add whatever data they want to the card element, because card elements are visible to user-written functions.

Each card element contains information about all its ports. It is represented as an array of report elements, each of which describes a single port of the card. Thus, the cross-referenced data structure is built in the generated code. Each report element involved in I/O mapping has a pointer to its card element, whereas each card element has an array of pointers to the report elements that are its I/O ports. This data relation is static and is initialized in the `init()` functions generated in the code.

It is assumed that each card is controlled by three functions that do the following:

- Initializes the card.
- Serves as a driver. Its purpose is basically to read data from, and write to, the card.
- Closes the communication with the card.

Remote Connection to Different Tools: Panels, GBA, Tracing: Description

Rational Statemate rapid prototyping includes provisions that allow data exchange between the executable that runs on the target, and a host. This is done while trying to minimize any negative effects on the regular execution flow of the embedded code.

All communications between the executable embedded code and the host tools is done through special, statically allocated buffers. As low-priority tasks, they are started periodically and/or according to other criteria, sending the communication buffer’s contents to the host. The user should run the remote panel server or Remote GBA server on the host machine before running the generated code on the target system.

Note

Remote panels are not supported.

BSP Configuration

Before configuring of the BSP, the user should change the `config.h` file based on the following information:

- Network board type
- I/O network bus base address
- IRQ level
- Target name
- Target IP address

After changing the configuration file, the user can run `WindConfig` (part of the Tornado development environment) to add or delete the options for the BSP build target.

The user must build the following:

- `bootrom_uncp`
- `VxWorks`
- `VxWorks.sym`

Refer to the *VxWorks Programmer's Guide* for a detailed explanation of how to do this.

Environment, Directories, Libraries, Files

Just as with other Rational Statemate features, there are environment variables that must be properly set. These include:

WIND_BASE	Tornado home directory.
WIND_HOST_TYPE	Host OS type (x86-win32, sun4-solaris2).
STM_ROOT	Rational Statemate home directory.
PATH	The path. It should be added by:
%WIND_BASE%\host\%WIND_HOST_TYPE%\bin	

The following libraries and object files can be used by the linker for the build of the target executable:

- `libintrinsicsI80486.a`
- `libdbgI80486.a`
- `librpgertlI80486.a`
- `libgbaI80486.a`
- `libschedulerI80486.a`
- `libsim_schedulerI80486.a`
- `real_mainI80486.o`
- `real_main_dbgI80486.o`
- `sync_mainI80486.o`
- `sync_main_dbgI80486.o`
- `async_mainI80486.o`
- `async_main_dbgI80486.o`

They are placed into the `%STM_ROOT%\lib\VxWorks` directory. If an alternate location is desired, it is necessary to change the appropriate lines in the `vxworks.rtrg` file. This file is created automatically when the workarea is created and is located in the `<workarea>\rprt` directory.

Note

The `rprt` directory also contains the profile files (`.rgenset`), the target description files (`.rtrg`), and I/O card description files (`.crd`).

Getting Ready: Connecting the Target to the Host

Depending on the specifics of the prototyping system's hardware implementation, it is necessary to establish a communications link between the target and the host. This is usually a serial interface such as an RS-232/485 port or an ethernet port.

If you are using the WindRiver® Tornado development environment, run the FTP server as described in the *Tornado User's Guide*. Here is the example of the *.sh* file, which can be used for this purpose:

```
cd %WIND_BASE\host/%WIND_HOST_TYPE%\bin
wftpd32.exe &        -- The Windows FTP Daemon running
wtxregd.exe –V       -- Tornado registry daemon running
```

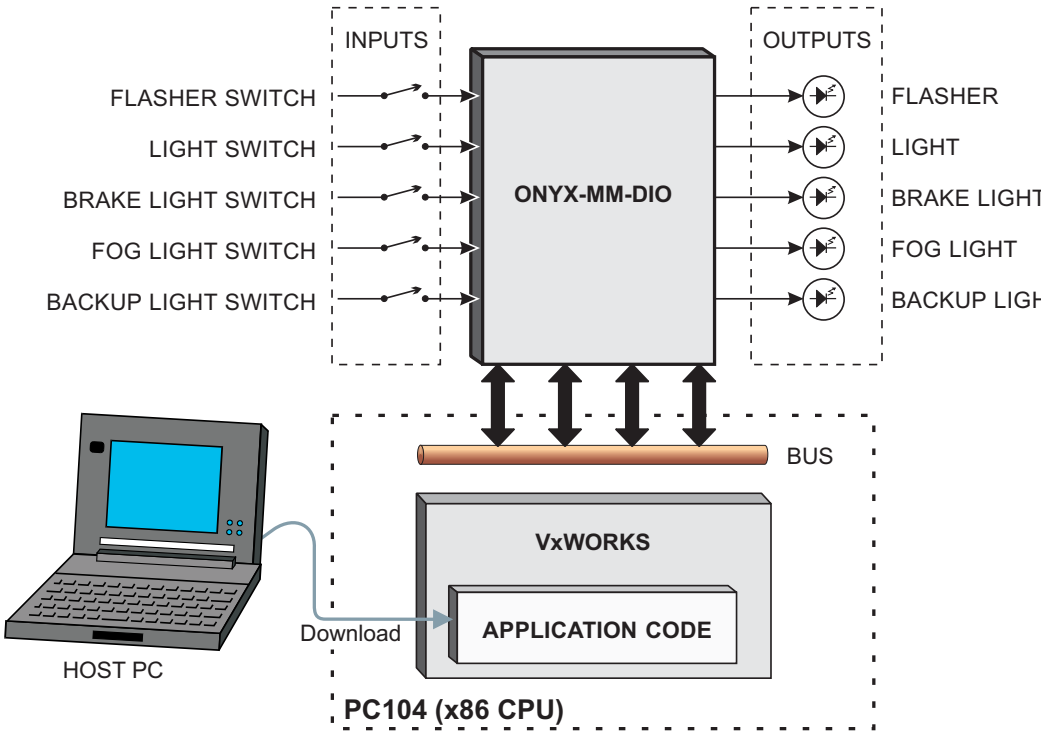
You should add the name and IP network address of the host and target into the file called `hosts`, which is located into the `\winnt\system32\drivers\etc` directory (further details can be found in the *Tornado User's Guide*.

Compiling Embedded C Code

Once the target requirements have been defined and the model designed, code can then be generated and compiled. This section describes these steps, as well as general considerations of coding for an embedded prototyping system.

Code Generation Sample Model Description

The following sections describe how to generate code using a sample model. This sample model uses a single I/O card on an x86 prototyping development system running the VxWorks RTOS. The following figure illustrates the I/O card configuration.



Report and Card Elements Declarations

The following code is generated in `r2main.c`:

```
card_desc_elem card_card_1_desc;
report_link pins_card_1_desc[2];

card_desc_elem card_card_2_desc;
report_link pins_card_2_desc[2];
```

Corresponding `extern` declarations are generated in `r2main.h`. As you can see, this profile includes two I/O cards. Each of these cards has two pins. The number of pins is exactly the number of elements in `pins_*arrays`.

Initialization

The following is a sequence of initialization actions required for I/O mapping:

- Initialize all card description elements. At this stage, the `pin_array` field is set so correspondence between card elements and their pin arrays is established.
- Initialize all pin arrays by 0.
- For every module, initialize report elements that belong to the module. At the same moment, the corresponding element of the pin array is set to be a pointer to the current report element. The pin arrays are completely initialized. The `card_p` field of the report element is also initialized. The cross-referenced data structure is built.
- Call the card initialization routines provided by the user. Note that at this moment, all data structures related to the I/O mapping in the code are built.

The following portions of `r2main.c` and `m1.c` illustrate these points:

```
r2main.c:

void lo_init()

{

  init_card_desc(&card_card_1_desc,
    card_generic_driver, card_generic_init,
    “card_1",pins_card_1_desc);

 init_card_desc(&card_card_2_desc,
    card_generic_driver,card_generic_init,
    “card_2",pins_card_2_desc);

  memset(pins_card_1_desc,0,2*sizeof(report_link));

  memset(pins_card_2_desc,0,2*sizeof(report_link));

  m1_init();

  dbg_init();

  (*card_card_1_desc.card_init)(&card_card_1_desc);

  (*card_card_2_desc.card_init)(&card_card_2_desc);

}


m1.c:

  int                  X_IN1;

  report_elem          rep_X_IN1;

  int                  X_IN2;

  report_elem          rep_X_IN2;

  int                  X_OUT1;

  report_elem          rep_X_OUT1;

  int                  X_OUT2;

  report_elem          rep_X_OUT2;

  ...

void m1_init()

{

  init_int(&X_IN1,0);

  init_report(&rep_X_IN1,"A2:X_IN1","",&X_IN1,
    el_enumer,0,trace_f,FALSE,&card_card_1_desc,
    "0x0012","1",2,input_mapf,NULL,NULL,FALSE,FALSE);

  init_int(&X_IN2,0);

  init_report(&rep_X_IN2,"A2:X_IN2","",&X_IN2,
    el_enumer,0,NULL,FALSE,&card_card_2_desc,
    "0x0012","1",2,input_mapf,NULL,NULL,FALSE,FALSE);

  init_int(&X_OUT1,0);
```

```
  init_report(&rep_X_OUT1,"A2:X_OUT1","",&X_OUT1,
    el_enumer,0,trace_f,FALSE,
    &card_card_1_desc,"0x0013","2",1,NULL,
    NULL,output_mapf,FALSE,FALSE);

  init_int(&X_OUT2,0);

  init_report(&rep_X_OUT2,"A2:X_OUT2","",&X_OUT2,
    el_enumer,0,NULL,FALSE,&card_card_2_desc,
    "0x0013","2", 1,NULL,NULL,output_mapf,
    FALSE,FALSE);

  init_activity(&A1,activ,FALSE,0,0,0,0,0,"A1",FALSE);

}
```

Step Execution

The `pr_make_step( )` function does the following:

- At the beginning of each step, it determines whether tracing should be enabled.
- The function that goes through the list of report elements that were changed during the current step and produces the output mapping and trace messages is called.
- At the end of each step, the stop trace condition is checked; if it holds, tracing is disabled.

The generated code is as follows:

```
r2main.c:
boolean  pr_make_step()
{
  boolean step_has_changes = FALSE;
  incr_stepN();
  if (X_OUT1 > 0)
    enable_trace();
  lo_main();
  step_has_changes = update();
  garbage_collect() ;
  if (!step_has_changes && (!deb_was_update()))
    return TRUE;
  if (call_cbks_p)
    (*call_cbks_p)(FALSE);
```

```
 update();
  do_report();
  if (X_OUT2 < 0)
    disable_trace();
  return FALSE;
}
```

Input Mapping

You must build a tree-like structure of functions to perform input mapping. In the example, there is only one module other than main, so it looks like this:

```
r2main.c:
void do_inputs()
{
  m1_do_inputs();
}

m1.c:
void m1_do_inputs()
{
  input_mapf(&rep_X_IN1);
  input_mapf(&rep_X_IN2);
}
```

The `input_mapf()` function does the following:

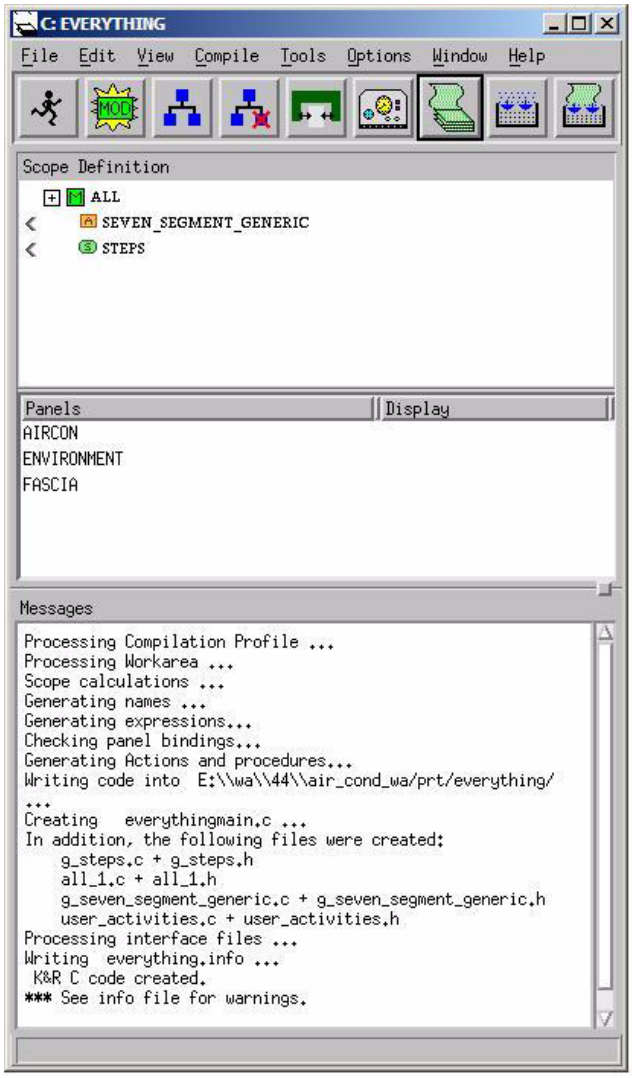
- Prepares a place in memory for the value to be read.
- Calls to the card driver function to read the new value of the data item from the card.
- According to the type of data item and double buffer settings within the report element, assigns the read value to the corresponding model element.

Starting Code Generation

Code is generated from the profile editor main screen. From an open profile, select the **Compile > Check Profile** menu item to confirm that the profile is complete and ready to generate code. If an error is reported, it must be reconciled before code can be generated.

To generate code, select the **Compile > Generate Code** menu item or click on the Generate Code speed button. This initiates the code generator and causes a text box to appear. It displays messages about the progress of code generation.

Once code is successfully generated, it is ready for compilation.



Compiling Generated Code

The compilation and linking of code is totally dependant on the specified target. This means that the user must supply the suitable compiler/linker for the desired target.

Compilation and Linkage.

The environment variables `WIND_BASE`, `WIND_HOST_TYPE`, and `PATH` must be set properly before running Rational Statemate.

The compile/link process can be started indirectly from within the profile editor. This is done by using a `make` file, which is initiated from the **Compile > Make Code** menu item or by clicking Compile Generated Code  .

This method allows the make file to be modified to accommodate the unique requirements of a specific compile/link process, including the downloading of compiled code into a target embedded prototyping system.

Downloading and Execution

After the target is booted and the generated code is compiled, you can run the executable. If you intend to run the model directly on the embedded prototyping system target, you should start the target server on the host machine, load the executable, and start it on the target.

The target server can be started from the command line using the following command:

```
tgtsvr  -V <target name>
```

Load the executable using the following command:

```
ld 1, 0, “<executable name>”
```

Run the executable using the following command:

```
vxmain
```

If you intend to run the model using the `WindSh` remote shell, you must complete the following:

1. Start the target server.
2. Run `WindSh`.
3. Redirect the standard I/O files to the virtual console window.
4. Download and execute the model on the target.

These actions can be done using the `run_windsh` batch file, whose full name (including path) should be printed into the `#Run script name` paragraph of the `<target>.rtrg` file. It looks something like the following:

```
#Run script name:"%STM_ROOT%\bin\run_windsh <target name>",
```

In the command, `<target name>` is the name or TCP/IP address of the target machine.

Note

The files `run_windsh.bat` and `run_windsh.csh` are located in the encrypted VxWorks distribution file.

The `run_windsh` batch file has two input parameters (*target name* and *executable name*) and looks like the following:

For Windows host platforms:

```
#Creating of the file for model executable download and execution

echo ioGlobalStdSet(0,vf0)   >>  run_model.bat- reopening of the virtual I/O
channel 0

echo ioGlobalStdSet(1,vf0)  >>  run_model.bat- designation of standard input
file

echo ioGlobalStdSet(2,vf0)  >>  run_model.bat - designation of standard error
file

echo logFdAdd(vf0)"            >>  run_model.bat- sending logging output to
the virtual channel 0

echo ld 1,0, "%2"                 >>  run_model.bat- download of the
executable

echo vxmain                           >>  run_model.bat- model execution
starting

start tgtsvr   %1 –C –c%WIND_BASE%/target/config/pc486/vxworks  - target
server starting

#WindSh running

windsh -n -s run_model.bat  %1 > null
```

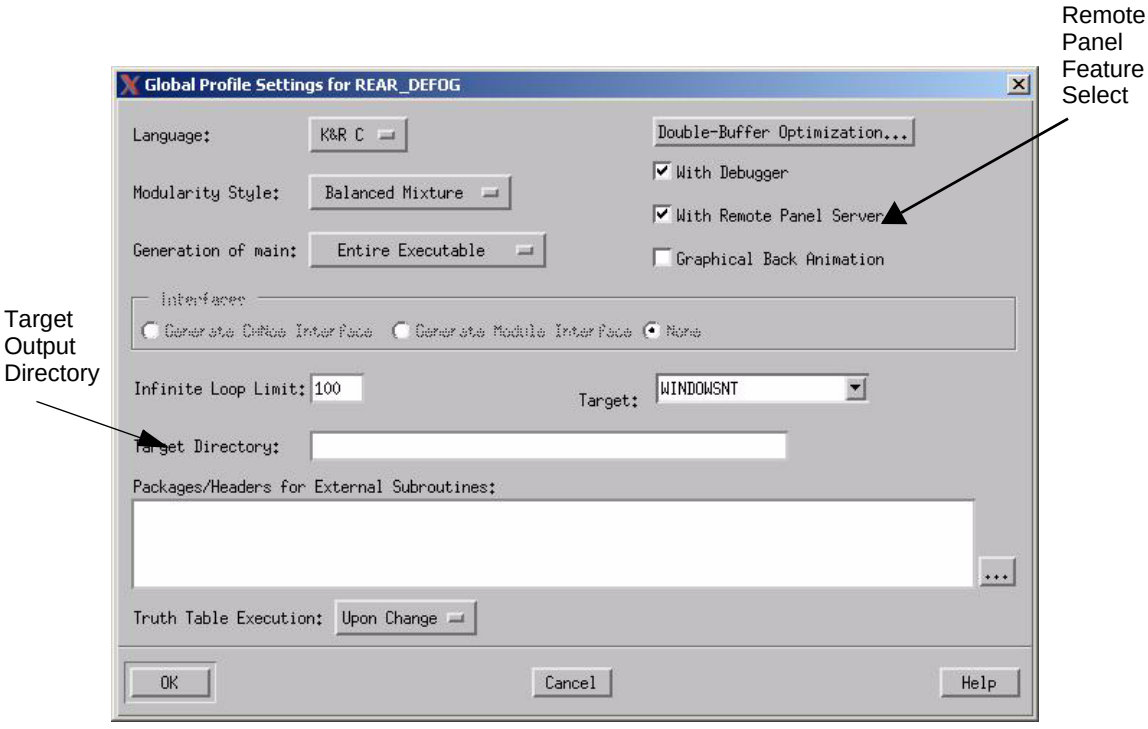
For UNIX platforms (Solaris):

```
#Creating of the file for model executable download and execution
#! /bin/csh -f
echo 'vf0=open("/vio/0",2,0)' >   run_model.csh
echo "ioGlobalStdSet(0,vf0)" >>  run_model.csh
echo "ioGlobalStdSet(1,vf0)" >>  run_model.csh
echo "ioGlobalStdSet(2,vf0)" >>  run_model.csh
echo "logFdAdd(vf0)"         >>  run_model.csh
echo 'ld 1,0, "'$2'"'              >>  run_model.csh
echo "vxmain"                     >>  run_model.csh
windsh -n -s run_model.csh $1
```

Now the model system is ready for debugging.

Remote Panel

The following figure shows the remote panel dialog.



To open this panel, click **Options** > **Global Profile Settings**.

If the mode is set, the code generator creates a proper makefile and additional elements in the generated code. In addition, when you run generated and linked code using the **Compile > RunCode** menu item, the profile editor sends the request to the main of Rational Statemate in order to run the *Remote Panel/Trace Server*. If the **Target** [output] **Directory** field is not empty, the Profile Editor copies the `rcomm.cfg` file (created by the Remote Server) into the target output directory where the code was generated. This configuration file contains the host name, the input port and output port addresses, and the debug level number.

The *target output directory* is defined by the appropriate entry in the Global Profile settings form. It should be the same directory defined in the file selection box during code generation, but in terms of the target file system. For example, the host output directory has the following path:

```
d:\tmp\io\rprt\io
```

The target output directory would be:

```
/tmp/io/rprt/io
```

If the target directory field is empty, the executable looks for the `rcomm.cfg` file in the workarea directory. It should normally be seen from the target exactly with the same name as in the host machine.

Next, run the generated code using the remote execution script whose name is defined in the `<target OS>.rtrg` file.

GBA

The Graphic Back Animation *(GBA)* mechanism's configuration functions are similar to those of the remote panel feature. It uses the configuration file `gba.cfg`, which is created when the GBA server runs. This file contains the following information:

- Host name
- Port address
- Debug level number (used for debugging purposes only)

After the configuration file is created by the GBA server, the Profile Editor copies it, and then deletes it from the workarea. This file is needed only in the first running of the generated code. If you subsequently run the same configuration of the host and target, there is no need to recopy this file. Generated code can use the existing file, so you can run it manually using the **RunCode** command from the Profile Editor.

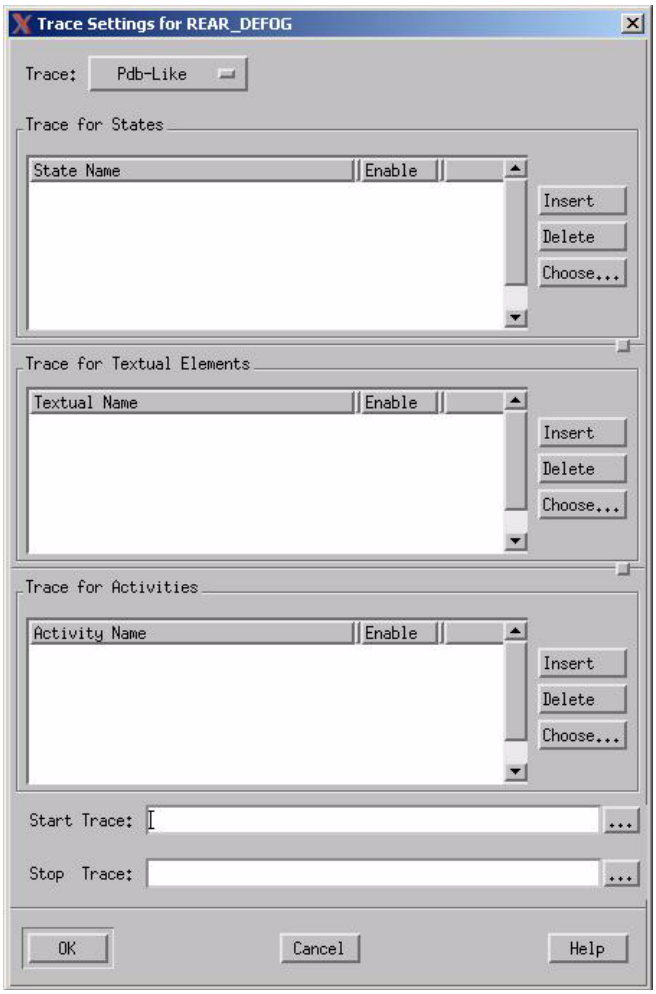
To start the GBA server, use the **Tools > Open GBA** menu item of the Profile Editor.

Trace Facility

If some variables (states, data items, conditions, or activities) are selected to be traced, the trace task is created automatically when generated code executes. This trace task uses the Remote Server to output the trace message data into the `<profile name>.trc` file, located in the target output directory. Its reporting mechanism is the same as that used for the Remote Panel feature. The basic mechanism allows the trace message data to be sent to the trace buffer, whose length is user-definable in the dialog called by the **Options** > **Trace Settings** menu item.

You can also define the format of the trace in this dialog. The format information is written into the `.trc` file.

If you select the **User Supplied** function name, every time the variable value is changed, this function is called instead of the standard trace function normally supplied by the `libintrinsics` library.



Required User-written Code

User-written code is the term applied to any custom code added to the generated code and included in the subsequent compilation. It is also called handwriiten code. This code may include modifications to the generated code, pretested and prequalified code modules, or special test or use case routines.

The interface between the I/O cards and the generated code is the responsibility of the user. This means that the user must supply at least three functional modules/routines:

- I/O card initialization routines
- I/O hardware driver routines
- I/O card shut-down/closure routines

The card element data structure contains a descriptive paragraph with the details of the API that generated code provides to its user. The card is a pointer to these three functions.

Card Initialization

The card initialization function should have one parameter, which is a pointer to the card element structure. When this function is called from the generated code, all internal data structures of the generated code are initialized properly.

Card Driver

The card driver function should have two parameters. The first is a pointer to the report element structure. The second is an integer that defines whether to perform input or output mapping. It is recommended that you use the constants `STM_IN_MAP` and `STM_OUT_MAP` (defined in the file `types.h` of the Intrinsics library) for this purpose. When you perform input mapping, the driver should put the received value into the `received_val` field of its first argument. Note that information about the corresponding card is available from the `card_p` field of the report element.

Card Closure

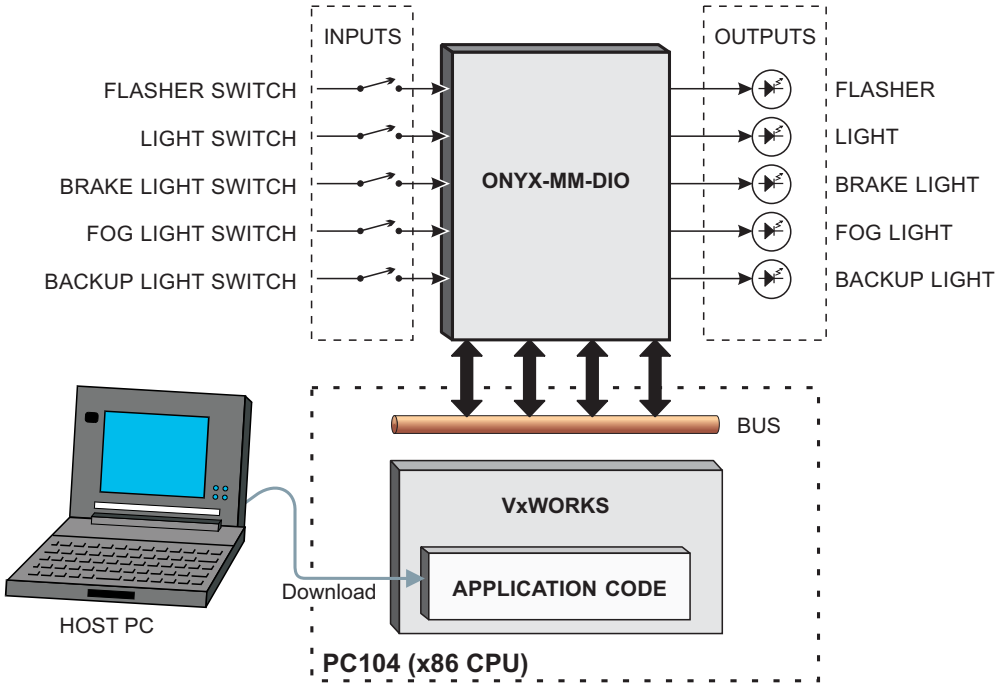
The card closure function has one parameter—a pointer to the card element structure. Its purpose is to perform the necessary actions upon finishing the work with the card. Refer to **Ada Code Generation**for an example.

Simple Embedded Code Example

Because of the highly hardware dependant nature of embedded code, the following example is not provided within the software. Nevertheless, this section examines a simple example application in terms of the hardware target and procedures necessary for setting up the system so that code can be generated and downloaded into the target.

Use Case

The use case example that is the subject of the sample code presented here is a simple tail light controller for an automobile. It is uses an x86 CPU based prototyping development system equipped with a PC104 bus and one digital I/O card (i.e. ONYX-MM-DIO). The development system is equipped with an ethernet controller which is used as the serial link to the host system. The RTOS is VxWorks and the compiler/linker/debugger is integrated into the Tornado development environment. Rational Statemate is hosted on a Pentium laptop where all model and code development is performed, as well as all remote terminal functions. The following figure illustrates the basic system configuration, including I/O functions.



I/O Driver Functions

The following code sample is taken from a working driver file implemented to support the ONYX-MM-DIO I/O card from Diamond Systems Corp.

Example Code:

```
#include <vxWorks.h>
#include <stdio.h>
#include <stdlib.h>
#include <syslib.h>
#include "types.h"
#include "symbols.h"
#include "string.h"
```

```
#define DIO_1A    0
#define DIO_1B    1

#define DIO_1C    2
#define DIO_1CR   3

#define DIO_2A    4
#define DIO_2B    5
#define DIO_2C    6
#define DIO_2CR   7

int onyx_base_addr;  /* The card base address , converted into int format */

/* -- generic card initializer for both input and output mapping */
void  onyx_init(card_desc_p card_p)
{ int addr;
sscanf(card_p->base_addr,"%x",&onyx_base_addr);
printf ("base_addr = 0x%x\n",onyx_base_addr);
sysOutByte(onyx_base_addr+ DIO_2A,0x00); /* Reset of output registers before
setting of 2A port
                to be output  */
sysDelay();
addr = onyx_base_addr+DIO_1CR;/* control register 1CR address
*/
                 /*  Port 1A , 1B and 1C set to OUTPUT/MODE 0  */
sysOutByte(addr, 0x80);
printf ("Ports 1A , 1B and 1C set to OUTPUT/MODE 0\n");
sysDelay();

addr = onyx_base_addr + DIO_2CR;  /* control register 2CR address
*/
sysOutByte(addr,0x9B); /*   Port 2A , 2B and 2C set to INPUT/MODE 0   */
printf ("Ports 2A , 2B and 2C set to INPUT/MODE 0\n");

}

/* -- generic card driver for both input and output mapping */
```

```
void  onyx_in(report_link elem)
{
  genptr  new_value = (genptr)elem->received_val;
  int     b, offset;

  sscanf(elem->pin_offset,"%d",&offset);
  b = sysInByte(onyx_base_addr+offset);   /* input from 2A port */

 switch(elem->elem_type) {
   case el_integer:
   case el_enumer:
   case el_bit_array:
          *(int*)new_value = b;
          break;
   case el_condition:
   case el_event:
          *(char *)new_value = b;
          break;
   case el_real:
          *(double*)new_value = b;
          break;
   default:
          *(int*)new_value = 0;
          break;
  }
}

#define OUTPUT_BUF_SIZE   16
double  output_buf;

void onyx_out(report_link elem)
{
  void *actual_val = (void*) &output_buf;
  int offset;
  sscanf(elem->pin_offset,"%d",&offset);
```

```
switch(elem->elem_type) {
  case el_integer:
  case el_enumer:
  case el_bit:
  case el_bit_array:
   if (elem->pin_inverse)
         (*(int*)actual_val) = ~(*(int*)elem->elem_value);
    else
         (*(int*)actual_val) = (*(int*)elem->elem_value);
    sysOutByte(onyx_base_addr+ offset, *(char*)actual_val);   /*  a=> output
to 1A port */
    break;
  case el_real:
    (*(double*)actual_val) = (*(double*)elem->elem_value);
    …
    break;
  case el_condition:
  case el_event:
    if (elem->pin_inverse)
         (*(char*)actual_val) = ~(*(char*)elem->elem_value);
    else
         (*(char*)actual_val) = (*(char*)elem->elem_value);
    sysOutByte(onyx_base_addr+ offset,*(char*)actual_val);
         break;
  default:
    break;
  }
   }

void onyx_driver(report_link elem, int map_mode)
{
  if (map_mode == STM_OUT_MAP) {
    onyx_out(elem);
  }
  else if (map_mode == STM_IN_MAP) {
    onyx_in(elem);
```

```
  }
}

/* -- generic card closing function */

void  onyx_close(card_desc_p  card_p)
{
   /* Some RESET calls for I/O card */
    printf("Card has been closed\n");
}
```

Target Description File

This section describes details of the target description file. Some of the key words are actually strings that are basically copied to the `Makefile` that is created along with the generated code. For clarity, we call their type: **makefile string**. To use this feature, you should have certain basic knowledge about `Makefile` language and syntax. Examples of possible values for important key words are included in the sample code listing.

In particular, if you want to use `$STM_ROOT` as part of the value of a string or a makefile string key word, it is necessary to use a duplicate $ sign to correctly express the variable type. For example: “$$STM_ROOT”

Note that some of the keywords contain OS path information. You should be aware of the correct directory separator character to use for the target operating system.

The internal double quote character shown here should be replaced by the single quote character. For example, the line **“ /D “PRT” “** should be **“ /D ‘PRT’ “**.

Example Code:

The following example is taken from the `vxworks.rtrg` file for the target OS: VxWorks.

```
{
    #UNIX-like target OS:yes
    #Link command:"LINK = $(LD)"
    #System libraries:""
    #Library extension:".a"
    #Executable extension:""
    #Output file keyword:"-o "
```

```
    #Intrinsics library:"$(STM_ROOT)/lib/VxWorks/libintrinsics$(CPU).a"
    #Scheduler library:"$(STM_ROOT)/lib/VxWorks /libscheduler$(CPU).a"
    #Simulated scheduler library:"$(STM_ROOT)/lib/VxWorks /
libsim_scheduler$(CPU).a"
    #Debugger library:"$(STM_ROOT)/lib/VxWorks /libdbg$(CPU).a"
    #GBA library:"$(STM_ROOT)/lib/VxWorks /libgba$(CPU).a"
    #Panel library:"$(STM_ROOT)/lib/VxWorks /libpgertl$(CPU).a"
    #Remote panel library:"$(STM_ROOT)/lib/VxWorks /librpgertl$(CPU).a"
    #Additional libraries:""
    #Object extension:".o"
    #Archiv command:"$(AR) $(ARFLAGS) "
    #File existing command:""
    #File deleting command:"$(RM) "
    #Make command:"$(MAKE) -f"
    #Main file directory:"$(STM_ROOT)/lib/VxWorks/"
    #CPU name:"CPU = I80486"
    #Ranlib command:""
    #K&R C compiler name:""
    #K&R C compiler flags:
"STM_CFLAGS = -O -I$( STM_ROOT )/etc/prt/c -I$( STM_ROOT )/etc/sched -DPRT -
DVxWorks"
"TOOL = gnu"
"include $(WIND_BASE)/target/h/make/defs.bsp"
"include $(WIND_BASE)/target/h/make/make.$(CPU)$(TOOL)"
"include $(WIND_BASE)/target/h/make/defs.$(WIND_HOST_TYPE)"
"include $(WIND_BASE)/target/h/make/rules.bsp"
"INCLUDE_QUALIFIER=-I"
"CC_OPTIM += $(STM_CFLAGS)"
    #ANSI C compiler name:""
    #ANSI C compiler flags:
"STM_CFLAGS= -O -I$( STM_ROOT )/etc/prt/ansic -I$( STM_ROOT )/etc/ansisched -
DPRT -  VxWorks"
"TOOL = gnu"
"include $(WIND_BASE)/target/h/make/defs.bsp"
"include $(WIND_BASE)/target/h/make/make.$(CPU)$(`TOOL)"
"include $(WIND_BASE)/target/h/make/defs.$(WIND_HOST_TYPE)"
"include $(WIND_BASE)/target/h/make/rules.bsp"
```

```
"INCLUDE_QUALIFIER=-I"
"CC_OPTIM += $(STM_CFLAGS)"
    #Link flags:"-r "
    #Make script name:""
    #Run script  name:"%STM_ROOT%\misc\VxWorks\run_windsh mary"
  }
```

dSPACE Support

The Rational Statemate Embedded Rapid Prototyper (ERP) supports dSPACE models DS1102 and DS1103. The dSPACE DS110* models are single-board solutions—the processor and I/O are located on the same card.

The dSPACE interface enables you to do the following:

- Generate C code from the Rational Statemate model, compile the code, and download it to the dSPACE machine with a single click.
- Map model elements to the board I/Os.
- Automatically generate dSPACE TRC files for use with dSPACE ControlDesk layouts.

This section describes the required driver configuration and sample usage. The topics are as follows:

The dSPACE Package

The dSPACE package includes the following:

- Run-time library source code and batch files (which must be compiled on your machine). The libraries are linked with the generated code and the resulting executable is downloaded to the target. Refer to **Before You Begin**for more information.
- I/O driver library source code and batch file. The library performs the I/O calls according to the ERP I/O mapping definitions. Refer to **I/O Driver Configuration Settings** for more information.
- Automatic generation of TRC files, which are used for variable binding in the ControlDesk. Refer to **Generating TRC Files** for more information.

Unsupported Rational Statemate Functionality

Currently, the following Rational Statemate functionality is not supported on the dSPACE hardware:

- Graphical Back Animation (GBA) and the remote panel
- User tasks

  When you define tasks, keep the following in mind:

  - There is a special format for writing user tasks for dSPACE, which is different from the one used for other targets. Therefore, you should modify existing models in order to execute them on the dSPACE machine.
  - The main benefit of a task (running concurrently to the model) is effectively lost on the dSPACE hardware—the behavior is no different from a normal Rational Statemate subroutine defined as a procedure or function.

    For more information, refer to **Implementing User Tasks**.
- Continuous diagrams (VISSIM)
- Simulated synchronous and simulated asynchronous time model
- Double polling rate

Unsupported I/O Signals

The following I/O signals are not supported on the dSPACE model DS1103 hardware:

- PWM3 generation (synchronized 3 PWM signals)
- PWMSV generation
- CAN
- Synchronized reading of A/D converter (ADC) signals
- Serial interface
- Slave ADC

Refer to **Signals** for the list of supported signals.

Before You Begin

Before you begin using the dSPACE interface, perform the following tasks:

1. Edit the `run_stmm.bat` file.
2. Compile the run-time libraries.

Editing the Batch File

To use ERP on dSPACE hardware, you must edit the batch file `%STM_ROOT%\bin\run_stmm.bat`. Refer to the dSPACE section of the file for the list of necessary changes.

Compiling the Run-Time Libraries

The following sections describe how to compile the run-time libraries for the dSPACE boards.

- **DS1102**
- **DS1103**

DS1102

To compile the run-time libraries for the DS1102 board, complete the following:

1. Open a DOS shell.
2. Set the environment variable `STM_ROOT`.
3. Execute the following file:

```
%STM_ROOT%\etc\prt\c\create_DS1102_intrinsics.bat
```

4. Execute the following file:

```
%STM_ROOT%\etc\sched\create_DS1102_sched.bat
```

DS1103

To compile the run-time libraries for the DS1103 board, complete the following:

1. Open a DOS shell.
2. Execute the following batch file:

```
%STM_ROOT%\etc\rapid\build_DS1103_libs.bat
<Statemate installation dir>
```

For example:

```
%STM_ROOT\etc\rapid\build_DS1103_libs.bat C:\IBM Rational\stmm\4.6
```

When you execute the batch file, the following libraries and object files are created:

```
%STM_ROOT%\lib\dspace\DS<xxxx>\libintrinsics.lib
%STM_ROOT%\lib\dspace\DS<xxxx>\libscheduler.lib
%STM_ROOT%\etc\rapid\ds<xxxx>\obj\stm_ds1103.lib
%STM_ROOT%\lib\dspace\DS<xxxx>\real_main.obj
%STM_ROOT%\lib\dspace\DS<xxxx>\sync_main.obj
%STM_ROOT%\lib\dspace\DS<xxxx>\async_main.obj
```

To compile the remote debugger run-time libraries for the DS1103 board, complete the following:

1. Open a DOS shell.
2. Execute the following batch file:

```
%STM_ROOT%\etc\rapid\build_DS1103_libs_dbg.bat <Statemate installation dir>
```

For example:

```
build_DS1103_libs_dbg.bat C:\IBM Rational\stmm\4.6
```

When you execute the batch file, the following libraries and object files are created:

```
%STM_ROOT%\lib\dspace\DS<xxxx>\libintrinsics_dbg.lib
%STM_ROOT%\lib\dspace\DS<xxxx>\libscheduler_dbg.lib
%STM_ROOT%\etc\rapid\ds<xxxx>\obj\libdbg.lib
%STM_ROOT%\lib\dspace\DS<xxxx>\real_main_dbg.obj
```

Using the dSPACE Interface

The following sections describe how to use both the normal dSPACE interface and remote debugger mode.

Normal Use

To use the dSPACE interface, complete the following steps:

1. Open Rational Statemate and click the **Embedded Rapid Prototyper** icon.
2. Select **Options > Global Profile Settings.** The Global Profile Settings window is displayed.
3. Enter `stm_dspace.h` in the **Packages/Headers for External Subroutines** field.
4. Unselect the options **With Debugger**, **With Remote Panel Server**, and **Graphical Back Animation**.
5. Select the **target** in the **Target** field. For example, **DS1103**.
6. Click **OK** to dismiss the window.
7. Click **Options > Time Settings >Time model** in the Profile Editor.
8. Select **Real Time**.

9. Use the I/O mapping tool (refer to **Describing Signal Mapping to I/O Cards**) to map model elements to I/O ports using the DS1103 card.

   Refer to **Setting the I/O Polling Rate** for more information on I/O.

10. Generate, make, and run (load) the code. Two scripts are called in the process to make the generate code and to load the executable.

Remote Debugger Mode

Remote debugger mode provides model-level debugging. The ERP interacts with the target using a terminal program running on the host and communicating with the target via serial communication.

Note the following restrictions:

- I/O is not supported in remote debugger mode.
- Remote debugger mode is not supported on the DS1102 board.

To use remote debugger mode, complete the following:

1. Open Rational Statemate and click the Embedded Rapid Prototyper icon.
2. Select **Options > Global Profile Settings.**The Global Profile Settings window is displayed.
3. In the field **Packages/Headers for External Subroutines**, type `stm_dspace.h`.
4. Select the option **With Debugger**, but unselect **With Remote Panel Server** and **Graphical Back Animation**.
5. In the **Target** field, select **DS1103_DBG**.
6. Click **OK** to dismiss the window.
7. In the Profile Editor, click **Options > Time Settings> Time model**. Select **Real Time**.
8. Connect the PC COM port to the DS1103 panel Slave RS232 connector. Use a “simple null modem without handshaking” cable.

9. Use a terminal program (such as the Freeware `console.exe`) with the following settings:
   - **COM port**—As actually connected
   - **Baud rate**—9600 baud
   - **Parity**—None
   - **Databits**—8
   - **Stopbits**—1
   - **Echo**—No
10. Generate, make, and run (load) the code.

Generating TRC Files

The TRC file defines the variables that can be read to or written from the hardware by the dSPACE ControlDesk at run time. The variables that are visible to the host are those defined in the C code as global variables. (In the Rational Statemate generated code, all the model elements are actually global variables.)

The variables are arranged in the TRC file in a hierarchy (groups) according to the model charts' hierarchy. This arrangement enables you to easily navigate in the ControlDesk variables browser.

TRC file generation is enabled when the corresponding field in the target file (`.rtrg`) is set to yes, as follows:

```
#Generate dSPACE TRC file:yes
```

For ease of use, the TRC file has a separate group for each chart. All the defined variables in a chart are part of the same group in the TRC file.

In the Rational Statemate generated code, some variables are named differently from the model name in order to solve a uniqueness problem. For those variables, the entry in the TRC file is the Rational Statemate model name (defined as an alias to the "code name").

I/O Driver Configuration Settings

Some information regarding the hardware configuration (such as the frequency or the resolution for the PWM port, or the range of the value read from the ADC port) is available to the driver at run time. Until the information exists in the card file, it is hardcoded in a part of the driver source code.

A C structure that holds all the required information is instantiated in the file `stm_ds<xxxx>_config.c`, where `<xxxx>` is the model number of the board. The structure is initialized to labels defined in the corresponding header file `stm_ds<xxxx>_config.h`.

You must change the header file according to the actual card configuration, then compile the file with the rest of the driver libraries. Instructions for modifying the header file are included in the `stm_ds<xxxx>_config.h` file.

Setting the Timer Frequency

The timer interrupt frequency (timer resolution) is defined in the constant `DEFAULT_TIMER_RESOLUTION_MS` in the file `os_include.h`. The minimum value for this integer variable is 1; the default value is 10 milliseconds.

If you change this value, you must rebuild the run-time libraries to have your changes take effect.

Setting the I/O Polling Rate

You specify the I/O polling rate using the command **Files > I/O Card Management > Polling Rate**. A polling rate value 100 means that the I/O ports will be polled 100 times per second.

For more information on polling rates, refer to **Target Requirements.**

Driver Tasks

The driver performs different tasks during initialization and model execution.

Initialization Tasks

During initialization, the driver performs the following tasks:

- Processes signal mapping information, which is read from the ERP data structure
- Configures I/O ports
- Calls the initialization functions

Model Execution Tasks for the Driver

When reading from an input port, the driver performs the following tasks:

- Calls the input function
- Normalizes the read value (to match the model value range)
- Writes the new value to the ERP data structure

When writing to an output port, the driver performs the following tasks:

- Reads a new value from the ERP data structure
- Determines whether the value is in-range (according to the driver configuration)
- Normalizes the value (to match the I/O port range)
- Calls the output function

Signals

The following sections list the supported signal types, and the mapping combinations of Rational Statemate variables to signals on the dSPACE hardware.

Signal Types

The following tables lists the dSPACE signal types supported by the ERP driver. The signal type names are used in the Rational Statemate ERP I/O mapping table. For example, if you map a Rational Statemate variable to “ADC 1”, it is mapped to the first A/D converter pin on the dSPACE board.

In the tables, “ADC” stands for A/D converter, “DAC” stands for D/A converter, and “IOP” stands for input output port.

Note that the card file includes all the supported signals. You should change the `in_port` or `out_port` for IOP signals only, because they can be configured as either digital input or digital output.

Alternatively, you could make this change without editing the card file by following these steps:

1. Map an IOP signal.
2. Select In or Out in the I/O column of the I/O mapping table.

Port Names

The port names in the card file represent the port number in the dSPACE hardware.

For example, to map a Rational Statemate condition `COND1` to port “IOP 1” as output, configure the second digital port on the hardware to be an output port and map `COND1` to it.

Similarly, to map a Rational Statemate bit array `BITARR1` to port “IOP 1-3” as input, configure the second to fourth digital ports on the hardware to be input ports, and map each element of `BITARR1` to the corresponding ports on the hardware.

Note

Digital ports can be either input or output ports. You should configure the ports before using them.

The driver issues a warning message whenever information might be lost. For example, if a Rational Statemate integer is mapped to an A/D converter (real), the driver issues a warning.

Port Type	Port Name	Channel
Digital I/O	IOP	0 to 15
ADC 16-bit	ADC	1 to 2
ADC 12-bit	ADC	3 to 4
DAC 12-bit	DAC	1 to 4
Encoder	ENC	1 to 2
Encoder index	ENCIDX	1 to 2
PWM	PWM	1 to 6

The following table lists the DS1103 signal types.

Port Type	Port Name	Channel
ADC 16-bit	ADC	1 to 16
ADC 12-bit	ADC	17 to 20
DAC 14-bit	DAC	1 to 8
Digital I/O	IOP	0 to 31
Encoder position reading	ENC_POS	1 to 7
Encoder position delta reading	ENC_POSD	1 to 7
Encoder position writing	ENC_POSW	1 to 7
Encoder counter reading	ENC_CNT	1 to 7
Encoder counter fine reading	ENC_FINE_CNT	7
Encoder counter writing	ENC_CNTW	1 to 7
Encoder counter clearing	ENC_CNTCL	1 to 7
Encoder index reading	ENC_IDX	1 to 7
Slave PWM generation	SLV_PWM	1 to 4
Slave PWM measuring period	SLV_PWMD_PD	1 to 4
Slave PWM measuring duty	SLV_PWMD_DT	1 to 4
Slave Digital I/O	SLV_IOP	0 to 19
Slave frequency generation	SLV_DF	1 to 4
Slave frequency measuring	SLV_FD	1 to 4

Mapping Rational Statemate Variables to dSPACE Signals

The following table lists the mapping combinations of Rational Statemate variables (integer, real, and so on) to signals on the DS1103 board. The table uses the following abbreviations for the dSPACE I/O types:

- **ADC**—A/D converter
- **DAC**—D/A converter
- **IOP**—Input output port
- **PWM**—Pulse width modulation
- **ENC**—Encoder
- **AI** and **AO**—Analog input and output
- **DI** and **DO**—Digital input and output

Note

As noted in the table, a warning is issued by the driver when information might be lost.

dSPACE I/O Type	STMM Integer	STMM Real	STMM Bit	STMM Condition	STMM Bit Array
ADC (AI)	Yes (warning)	Yes	No	No	No
DAC (AO)	Yes	Yes	No	No	No
IOP (DI)	Yes	Yes	Yes	Yes	Yes
IOP (DO)	Yes (warning)	No	Yes	Yes	No
ENC_POS	Yes (warning)	Yes	No	No	No
ENC_POSD	Yes (warning)	Yes	No	No	No
ENC_POSW	Yes	Yes	No	No	No
ENC_CNT	Yes	Yes	No	No	No
ENC_CNTW	Yes	No	No	No	No
ENC_CNTCL	Yes	No	Yes	Yes	No
ENC_IDX	Yes	No	Yes	Yes	No
SLV_PWM	Yes	Yes	No	No	No

SLV_PWMD_PD	Yes (warning)	Yes	No	No	No
SLV_PWMD_DT	Yes (warning)	Yes	No	No	No
SLV_IOP (DI)	Yes	Yes	Yes	Yes	Yes
SLV_IOP (DO)	Yes (warning)	No	Yes	Yes	No
SLV_DF	Yes	Yes	No	No	No
SLV_FD	Yes (warning)	Yes	No	No	No

Implementing User Tasks

User tasks originate from two sources—model subroutines that are defined as tasks, and activities that are defined as tasks in the ERP Profile editor.

Normally, Rational Statemate user tasks can include some synchronization calls, including the following:

- `wait_for_event(<event>)`
- `task_delay(<delay_time>)`
- `scheduler()`

The synchronization calls suspend task execution and call the Rational Statemate scheduler. The task resumes execution when a specific event occurs; execution is resumed from the point at which it was stopped.

Note the following when using tasks on dSPACE hardware:

- Do not use an endless `while` loop to wrap the task code.
- You should call the `scheduler()` routine as the last command within the task. Do not call it anywhere else.
- Do not use calls to `wait_for_event()` and `task_delay()` in your task.

ERP CANoe Interface

The Rational Statemate ERP CANoe® interface uses a CANoe API (available in CANoe 3.0 build 43) and an enhanced version of the Rational Statemate Embedded Rapid Prototyper (ERP).

It is based on the ability to describe a CANoe node behavior by an external DLL. The code generated by Rational Statemate is compiled and packed in a DLL, which is then executed by CANoe.

The Rational Statemate model is generated in an enhanced Module Procedures Only mode. In this mode, the model is wrapped within one function, and extra code is generated. The extra code maps the model elements to CANoe environment variables. The extra code that is generated is called **module interface code**.

This section describes how to use Rational Statemate with the CANoe environment. The topics are as follows:

Specifying Profile Settings

Before generating the code, you must define the module interface (which model elements are read to, or written from, the environment), and the mapping between a Rational Statemate model element and the CANoe environment variable name.

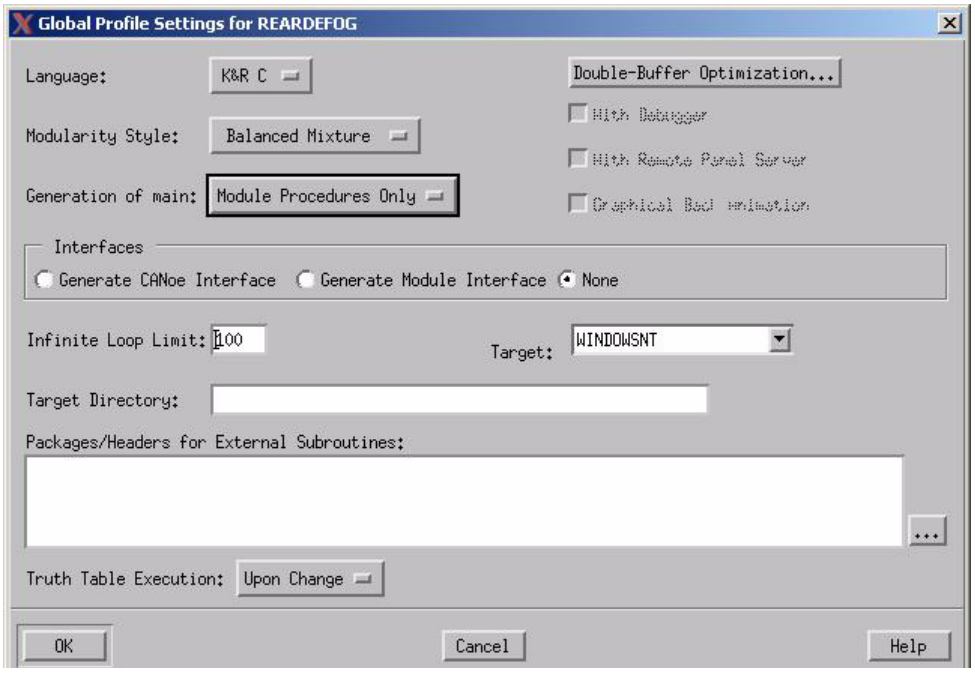
Complete the following steps:

1. In the `run_stmm.bat` file, uncomment the following line:

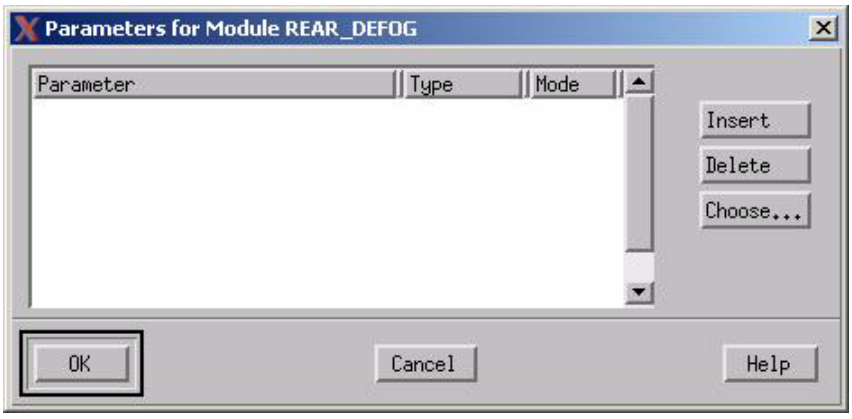
   ```
   set GEN_CANOE_IF_CODE=ON
   ```

2. Open Rational Statemate, and click the **Embedded Rapid Prototyper** icon.
3. Create a new profile (refer to Invoking the Profile Editor).

4. Select **Options > Global Profile Settings**. The following window displays:



5. Select **Module Procedures Only from the Generation of main:** drop-down list.
6. Select **Generate CANoe Interface**.
7. Disable the following options:
   - **With Debugger**
   - **With Remote Panel Server**
   - **Graphical Back Animation**
8. Specify a module and scope. Do **not** include a panel.
9. Click **OK**.
10. Select a module in the profile, then select **Options > Module Settings > Parameter Setting**. The Parameters for Module <Name> window is displayed, as shown in the following figure.



11. Select the **Type** [Data-Item (integer, real, string, or binary), Event, or Condition], and **Mode** (In or Out, but not In/Out) information for each parameter.

By default, if you have not specified values for the external symbols, Rational Statemate uses the values of the attribute `CANOE_ENV_VAR`.

If you want to select additional parameters, click **Choose**.

12. Set the mapped CANoe environment variable name as the value of the `CANOE_ENV_VAR` attribute in the Prioperties entry of the element (using the Attributes mechanism).

Note: Currently, the interface supports only one module. Repeat Step 10 for every element defined as part of the module interface.

13. Generate the code.

Code Generation

Currently, the external mapped symbol is read from the `CANOE_ENV_VAR` attribute.

The ERP CRD file includes a new message list section and a general port attribute list section. The CRD template is as follows:

```
{
  #card name:""
  #card polling rate:
  #card number of ports:
  #card base address:""
  #card init function:""
  #card driver function:""
  #card closure function:""
  #port list:
  {
    #port name:""
  #port inverse logic:
  #port default mode:
  #port default buffer:
  #port offset:""
     #port attribute list:
{
  #key:””
  #value:””
}
   }
   #message list:
   {
     #message name:””
  #message id: “”
  #message period:
  #message size:
  #signal list:
  {
    #signal name:””
```

```
      #signal byte number:
      #signal starts at bit:
      #signal number of bits:
      #signal attribute list:
         {
           #key:””
           #value:””
         }
    }
     }
  }
```

The message list is stored in the `card_desc` data structure, whereas the port attribute list is stored in the `report_elem` data structure.

Module Interface Code

When you use the CANoe environment, Rational Statemate generates module interface code in the module file. The C macros used are defined in the `stmm.h` header file. The macro calls create an array that defines the mapping between a Rational Statemate model element and the CANoe environment, which is accessible by CANoe at run time. Note that the `init_module()` and `exec_module()` functions are called by CANoe.

The module interface code is as follows:

```
/******* CANoe interface code *******/
#include "private\stmm.h"
sw_module_ptr      MODULE_HANDLE  = 0;
condition          stm_BREAK_PADDLE;
real               stm_SPEED;


CN_ENVIRONMENT_MAP_BEGIN()
CN_ENVIRONMENT_ENTRY(&stm_BREAK_PADDLE,
  el_condition, STM_OUT_MAP, "EnvBreakActive")
CN_ENVIRONMENT_ENTRY(&stm_SPEED, el_real,
  STM_IN_MAP, "EnvDashboardEngSpeedDsp_")
CN_ENVIRONMENT_MAP_END()


void init_module()
{
```

```
  speed_init_module(&MODULE_HANDLE,
   &stm_BREAK_PADDLE, &stm_SPEED);
}


sw_module_status exec_module()
{
   return speed_exec(MODULE_HANDLE, 1);
}
```

Using the Generated Code

The generated module files (`<module>.c` and `<module>.h`) should be part of a Microsoft® Developer Studio® project, which includes source files and settings provided by Vector-Informatik GmbH®.

When you link these modules, use the run-time library `libschedulercn.lib` instead of `libscheduler.lib`. To generate this library, complete the following:

1. Set the `STM_ROOT` environment variable as it is set in the `run_stmm.bat` file before running the batch file in the next step.
2. Run the `\etc\sched\create_sched_cn.bat` batch file.

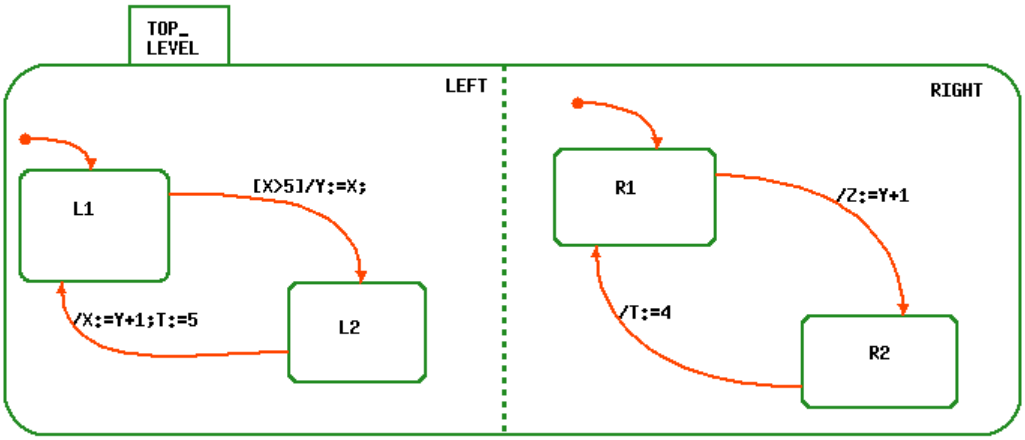
For more information on using the CANoe interface, go to the Vector-Informatik Web site (http://www.vector-informatik.de).

Double Buffering

This section shows examples where removing double buffers increases efficiency and other cases where it may introduce errors.

Double-Buffered Statechart

The following Statechart helps to illustrate how the Double Buffer Analysis program works.



When you run the Double Buffer Analysis program on the above Statechart, it reports that you only have to double buffer data items Y and T. There is no need to double buffer X and Z.

The Double Buffer Analysis program uses the following rules to arrive at its recommendations:

In general, you need double buffering in two cases:

- When assignment and accessing a data value occur concurrently (read-write racing).
- When two assignments to a data value occur concurrently (write-write racing).

In both cases, the obtained result may depend on the order of execution of concurrent components.

For example, suppose that upon default the Statechart enters the states L1 and R1 (X = 6 and Y = 10). Then, the execution of a step may produce these results:

- If orthogonal component LEFT is executed before component RIGHT, then at the end of the step Y = 6 and Z = 7.
- If RIGHT is executed before LEFT, then at the end of the step Y = 6 and Z = 11.

Similarly, after performing a step that involves transition from L2 to L1 and transition from R2 to R1, data-item T have value 5 or 4 depending on the order of the components execution.

With double buffering, values assigned during a step are deferred until the end of the step so that all actions in the step are executed with values the data had at the beginning of the step. In this case, at the end of the step Y = 6 and Z = 11.

Because of double-buffering, assignments are implemented in generated code through calls to special services:

```
seti(&Y, X);
seti(&Z, Y+1);
```

rather than by direct C assignments:

```
Y = X;
Z = Y + 1;
```

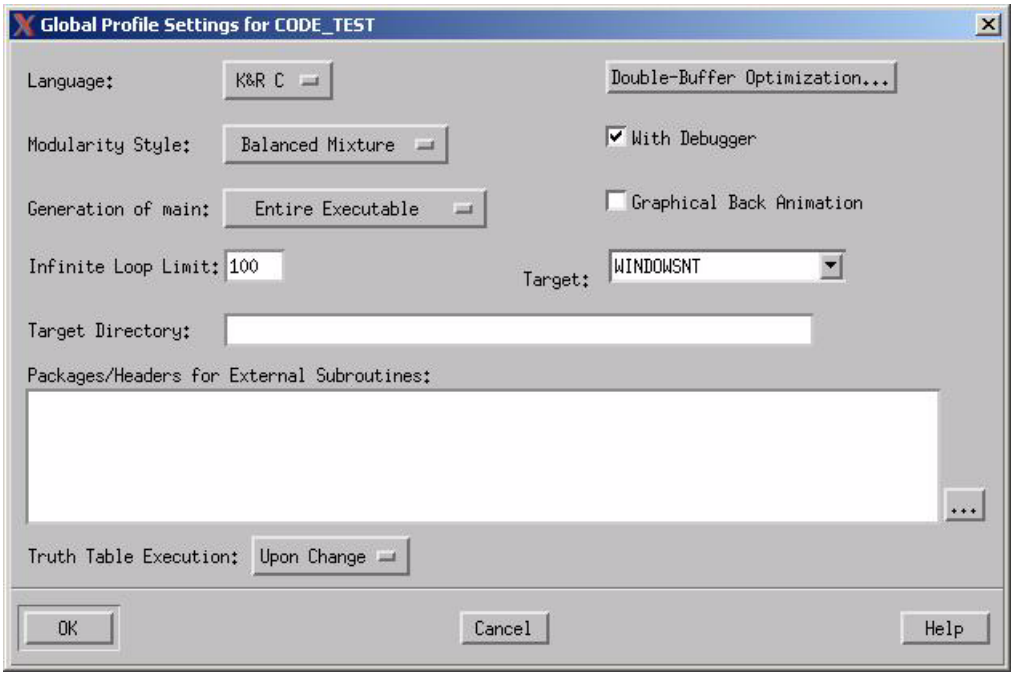
The Double Buffer Analysis program also finds elements in the model for which double buffering can be safely removed. For example, in the sample Statechart, the following elements do not need double buffering:

- Data-item Z is an output only, i.e., it is only assigned a value in the model, but never used in it. Therefore, there is no read-write racing. Since there is only one assignment to Z, there is also no write-write racing.
- Data-item X has an assignment and use of X, but they both belong to the same component and are never executed at the same step. Therefore, X does not need double buffering.

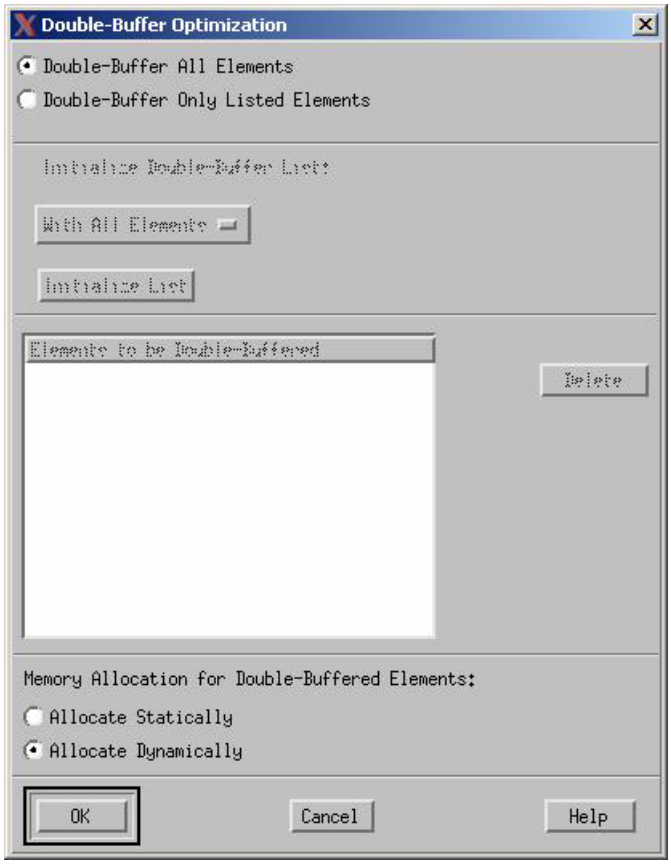
Optimizing Double Buffers

After running the Analyzer, complete the following steps to modify the compilation profile:

1. Select **Options > Global Profile Settings** in the Code Generator's main window. The Global Profile Settings window opens.



2. Click **Double-Buffer Optimization**. The Double Buffer Optimization window opens.



3. Select **Double Buffer Only Listed Elements**. This enables you to choose the elements to double buffer.

   The Initialize Double-Buffer List fields become enabled.

4. Change **With All Elements** to From Saved List.

   An drop-down list displays. You can select any list of elements that you saved including the lists you created with the Analyzer.

5. Select the list that the Analyzer created when you selected With Double Buffering List Name.

6. Click **Initialize List** and the element names appear in the Elements to be Double Buffered list.

   All assignments in the generated code are done in accordance with the selected list:

   - All elements in this list are double-buffered.

     For example, Rational Statemate assignment to an integer data-item such as X := 5 is translated into the following call:

     ```
     seti(&X,5);
     ```

   - All elements **not** in this list use direct assignments in the natural C style; for the above assignment this would be:

     ```
     X = 5;
     ```

7. Select a memory allocation option for elements that are double buffered.

   - **Allocate Statically** - Generally faster if there is a small number of elements.
   - **Allocate Dynamically** - Generally faster if there is a large number of elements. This is the default setting, which you can change in **Options > Preferences Management.**

Ada Code Generation

This section describes the architecture of the generated Ada code including how the Code Generator structures the modules.

The Rational Statemate Code Generator generates fully functional code, based on the Statecharts and Activity-charts in the Rational Statemate model. The generated modules are partitioned according to a compilation profile, which allows you to generate code for a complete Rational Statemate model or just a subsection of the 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).

The generated code uses runtime modules for timing and scheduling. Requests are generated to the timing module for timeouts and scheduled events, and to the scheduler module to control handwritten tasks that are connected to basic activities. In addition, the data elements are double buffered, so data assignments are synchronized to prevent racing conditions among the “concurrent’ behavioral components.

Note

Ada is supported for regular code generation only, not ERP.

Code Libraries

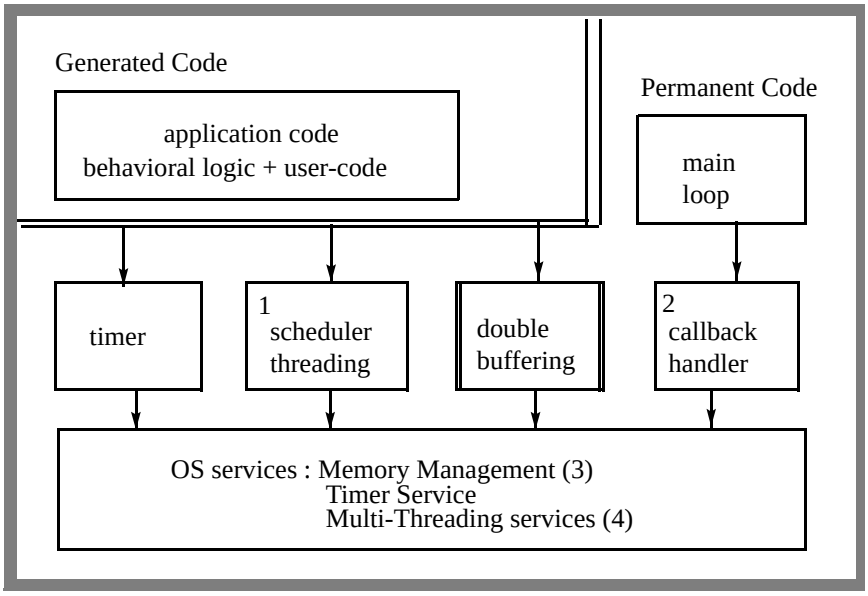
All the runtime modules are actually a set of compiled libraries. The libraries can be reused for other projects as they are supplied in source code form. The runtime modules actually comprise an interface between the generated behavioral logic and the underlying operating system.

Porting the generated behavioral components to a particular environment primarily means to tailor the runtime library to use the specific services provided by the operating system/real-time kernel, or in cases where none of them exist, to provide an alternative functionality.

Note

Tailoring the runtime libraries is a one-time effort. Once completed, the generated components can be compiled and linked without being modified.

The following diagram shows the layers of software components in the embedded application. The final executable image is built from some permanent modules, in addition to the generated modules that are dependent on the application.



- **The scheduler component is optional** - It is needed only if the user specifies that basic activities should be implemented as tasks or desires to link a graphic panel into the executable.
- **Callback handler** - Is used only if the user selects to attach callback routines to behavioral logic components.
- **Memory management** - The runtime modules timer, double-buffering and callback handlers utilize dynamic memory allocations. Under certain assumptions it is possible to tailor them to use only static allocation, if a memory management package is not available or memory resources are limited.
- **Multi-threading (tasking) support** - This support provides a mechanism for creating task threads and switching between them. This service is needed only if the user wishes to implement environment tasks or basic activities as tasks. This issue is discussed in greater detail in this document.

Tasks View of the Code

One of the major issues that confuse many users is how “concurrent” activities and states are actually translated into a sequential language. 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.

Writing an embedded application as a single thread or multi-threaded is actually a design decision. Since the underlying architecture is sequential, a multi-threaded program is actually a set of sequential pieces managed by a sequential scheduler.

Module Execution

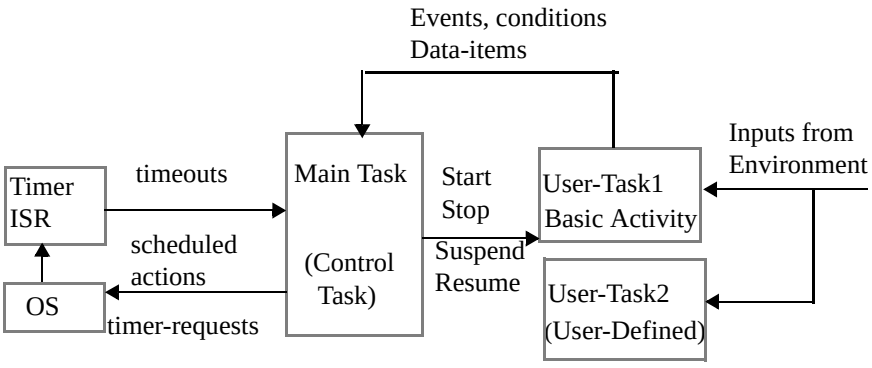
The modules of the generated code are sequential. They are executed cyclically with each iteration evaluating the next step of processing. In terms of simulation, executing the code is equivalent to executing a “go-step” repeatedly, while changing the environment asynchronously. The main difference is that the clock is incremented in real time, so timeouts happen according to the time taken to execute the code.

Multi-Threading

So why is multi-threading needed at all? Multi-threading is used to allow the user to implement basic activities as independent processes, without having to comply with the “one cycle at a time’ method. It also allows writing additional environment processes outside the system model, to process inputs, to drive outputs or for simulating the environment. Therefore, a multi-threading capability is needed only if the user wishes to add threads that run “concurrently’ with the generated modules that execute as a single thread, denoted as the “main task’.

Asynchronous Timer

Another component in the process view of the code is the asynchronous timer. The main task issues timer requests to be notified about timeouts and scheduled actions. The timer module asynchronously notifies the main task when timeout events are occurring



- In some applications there are no basic activities implemented as tasks. In those cases, the only processes that exist are the main task and the asynchronous timer. If basic-activity tasks exist, the main task issues tasking control calls such as start, suspend, etc.
- There are cases where the user implements environment tasks, but none of the basic-activities are implemented as a task. In these cases, the generated-code (the main task) does not use any tasking services. The code does not need a multi-threading adaptor unless the user connects a panel to the executable.

Using Simulated Time Model

The generated code uses the *real-time* model, by default. In this model, timeouts and scheduled actions are treated very similarly to other inputs. The system clock keeps time and generates interrupts which are processed along with the other inputs.

When using this time model, it is possible for the code to miss a timeout or scheduled action due to heavy loading of the processor or an extremely small request for a timeout. In such a situation, the generated code may actually behave slightly different than a simulation of the same model.

Rational Statemate also provides a *simulated-time* model. The purpose of the *simulated-time* model is to force the generated code to behave in the same manner as the simulated model. It does this at the cost of the *real-time* nature of the generated code.

In order to meet all timeouts regardless of duration and CPU loading, the code would be required to run at an arbitrarily fast speed. Since this is not possible, code which is compiled using the *simulated-time* model, does not adhere to the system clock. Rather, it keeps its own artificial time, much the same as a simulator. An internal counter is kept. The code executes model steps until it reaches an idle status. It then advances the internal clock to the necessary value to execute the next timeout or scheduled action.

```
-- The main loop, loops forever
procedure main is
       . . .
begin
       INIT_FUNCTIONS_. . .
       . . .
       USER_ACTIVITIES.USER_INIT;
. . .
       loop
              -- Execute a step --
              -- Advance internal time keeper to next
              relevant step --
              -- Apply timeouts and scheduled actions. -
       end loop;
exception
       . . .
end MAIN;
```

Main Task—Partition and Flow Control for Ada

In this section we describe 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 basic activity tasks are created. In addition the `user_init` procedure is called.

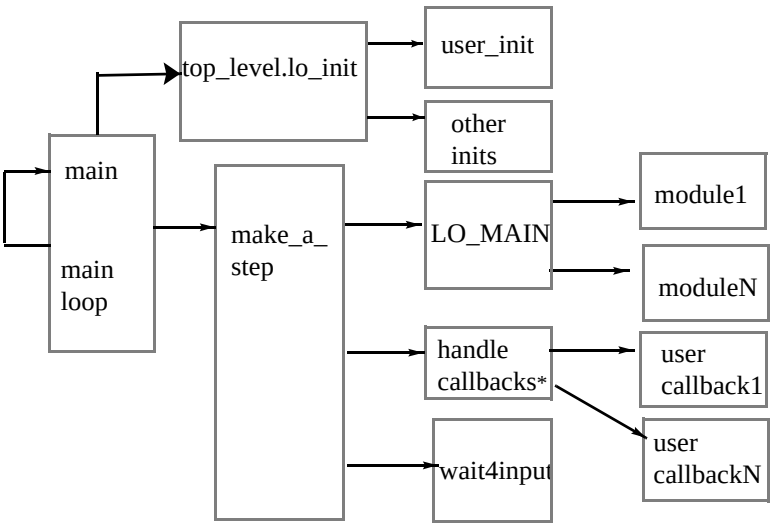
The `user_init` procedure resides in a file called `user_activities.a`. When you generate code, the Code Generator automatically creates the `user_activities.a` file and the `user_init` procedure. Prior to executing the model, you may initialize values in the `user_init` procedure.

After the 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.

This is how the Main procedure program looks:

```
begin
      PR_INITIALIZE;
   loop
            PR_MAKE_STEP;
   end loop;

       exception

       when NUMERIC_ERROR | CONSTRAINT_ERROR =>

             REPORT_error("NUMERIC_ERROR or CONSTRAINT_ERROR
             exception raised");
             FINISHING;

       when others =>

             REPORT_error ("Fatal error: exception raised");
             FINISHING;

end MAIN;
```

The following diagram shows the calling sequence within the main task:



Executing a Single Step

```
procedure MAKE_A_STEP is
  NEED_GO : BOOLEAN;
begin
  INCREMENT_STEPN;
    -- advance steps counter
  if IS_SYNC_TIME = true then
    -- synchronous (simulated) time model is used
    if get_stepN >1 then
         INCR_TIME;
          -- in synchronous time model,
          -- time is advanced at each step
          -- except for the first step when model
          -- enters its default states
    end if;
    WAS_UPDATE := true;
        -- this means that each step
        -- involves some update
    WAIT4INPUT;
  end if;
  TOP_LEVEL.LO_MAIN;
    -- execution of step; all changes are
    -- buffered till the end of the step
    SEMAPHORE.LOCK;
       -- to process the last step's changes,
       -- block the arrival of new external
       -- changes to the main task
    UPDATE_VALUES(NEED_GO);
       -- assign elements their new values,
       -- according to changes in the step
    WAS_UPDATE := WAS_UPDATE or DEB_WAS_UPDATE;
       -- true if there were changes in the
       -- previous step, or
       -- an element was changed between
       -- steps using the Debugger's SET command
    if not NEED_GO and then (not WAS_UPDATE) then
       -- check whether to enter the wait mode;
       -- do it if there were no updates in
       -- 2 last steps
      SLEEPING := TRUE;
                 -- mark the main task as going
                 -- to enter the wait mode
      SEMAPHORE.RELEASE;
                 -- and now allow arrival of new
                 -- external changes to the main task
      WAIT4INPUT;
                 -- if meanwhile an external input was
```

```
                -- generated then accept it
                -- otherwise enter the wait mode
    else
      SEMAPHORE.RELEASE;
                -- allow arrival of new external changes
    end if;
    WAS_UPDATE := NEED_GO;
    TRANSMITTER.EVAL_CALLBACKS;
          -- at the end of step, evaluate all
          -- callbacks  - according to hooks
          -- requested in Compilation Profile,
          -- and according to panel bindings
    GARBAGE_COLLECT;
          -- free memory which was temporarily allocated

          -- during step execution
  end MAKE_A_STEP;
```

Activating the Generated Modules (the “State Machines”)

`LO_MAIN` is a GENERATED procedure, that “glues’ together all the specific modules as partitioned by the compilation-profile. Since `LO_MAIN` refers to specific procedures, it differs between different models:

```
procedure LO_MAIN is
begin
   <module1>_EXEC_ALL;
   <module2>_EXEC_ALL;
   .........
   <moduleN>_EXEC_ALL;
end
```

Note

The `LO_MAIN` is actually the scheduler of the generated components. It applies a fair non-prioritized round-robin scheduling policy, similar to the interpretive simulator. However, it is possible to introduce priority scheduling by modifying this module.

Updating Double Buffer Assignments

The procedure `UPDATE_VALUES` executes all the deferred assignments into the actual data objects, based on the update list. As a by-product, the procedure 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 system executed an idle step. The `step_has_changes` flag indicates whether the step has ongoing processing, or the previous execution cycle was actually an idle step.

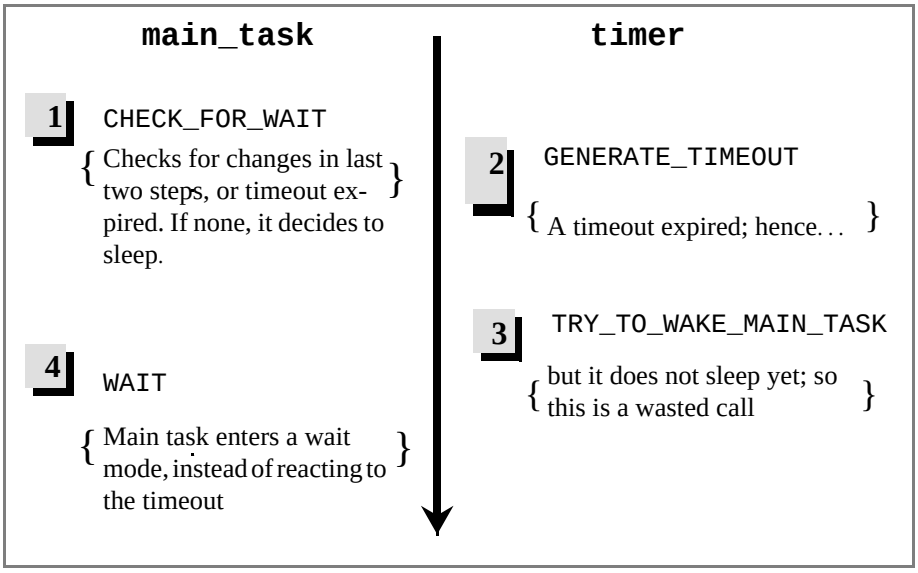
Evaluating the Callback List

If you sets hook for callback procedures, they are checked by EVAL_CALLBACKS. In case that one of the callback hooks is “active’, the callback procedure is called.

Entering the Wait State

If the system has executed two consecutive idle steps, it is in a stationary condition. The reason for executing two idle steps is that the negation of events might yield an active trigger after a single idle state. If the system is idled for two steps, no negative event triggers can take place. At this point, the main task releases the CPU by calling to a system service that blocks it from running until some external stimulus occurs. The external stimulus can be either an event/data change, or a timeout.

The decision whether to enter a wait state or not should be handled carefully, since once the main task blocks itself, only external input wakes it.



This scenario leads to a deadlock condition. Since the timeout is ignored by the system, the main task has already “decided” to hibernate itself but has not yet done so and the “wake-up” call is lost. The use of the SEMAPHORE task allows for virtual exclusion test-and-block and therefore avoids the deadlock.

Structure of a Behavioral Module

Each behavior module is translated into a package that contains objects and subprograms implementing that module in Ada. Two files are generated for such a package: one for the package specification and another for the package body.

The structure of the two files is explained for the module called `RAIL_CROSS`; the files are:

- `rail_cross_.a`package specification
- `rail_cross.a` package body

Package Specification

Context Clauses

Basic type definitions and services for conditions, events, data-items, etc.

```
with SYSTEM;          use SYSTEM;
with INTRINSICS;      use INTRINSICS ;
```

Services for timeouts and scheduled actions

```
with TIMEOUTS;        use TIMEOUTS ;
```

Rational Statemate queues operations

```
with QUEUES;          use QUEUES ;
```

Rational Statemate string operations

```
with STM_STRINGS;     use STM_STRINGS ;
```

Rational Statemate operations for bit-arrays

```
with BIT_OPERATIONS; use BIT_OPERATIONS ;
```

Rational Statemate predefined functions

```
with STD_FUNCS;       use STD_FUNCS ;
```

GDS containing the model's global types and constants

```
with UNCHECKED_CONVERSION;
with RAIL_DICTIONARY; use RAIL_DICTIONARY;
```

Generic behavior modules instantiated in module RAIL_CROSS with TOP_LEVEL; use TOP_LEVEL; elements shared by modules but not defined in GDS's

```
with g_GEN_BARRIER;
with g_GEN_ROAD_CROSS;
with g_GEN_TRAIN;
```

Interface Section Documents Inputs and Outputs

```
--Inouts
TIME1 :        FLOAT64 ;
END_MEASURE1 : event ;
```

Definitions of Data and Control Elements of the Module

In this section all primitive (non-compound) local data-items, events, and conditions are defined. “Local’ means elements not used outside the module scope.

```
ACK_BAR2_DOWN : event ;
IN_CRIT2 : condition ;
```

Definition of Fictive Events

```
ENMOVE_DOWN : event ;
tmDOWN_FINISHED:  event ;
```

The fictive events are events not explicitly defined in the model. They are essentially timeout events, enter/exit state events and in-state conditions. They are generated only when necessary; i.e. only if there is a use of en(STATE) the ENSTATE event appears in the code. The above two events appear because the model contains timeout event DOWN_FINISHED defined as tm(en(MOVE_DOWN), 5).

Definition of Activities

Each activity in the module is represented by a data structure that contains information on the activity’s current status, and on its hierarchical relations with other activities.

```
CHECK_TIME2 : ACTIVITY := dummy_act;
```

In addition, if in the Compilation Profile the user asks to implement a basic activity as task, then an appropriate task declaration appears:

```
task CHECK_TIME1_TASK is
      entry START;
end CHECK_TIME1_TASK ;
```

Generic Instances in the Module

Instances of generic charts in the module are translated into instances of the corresponding generic packages:

```
package Inst_BARRIER2 is new
       g_GEN_BARRIER(BAR2_DOWN,BAR2_UP,
              ACK_BAR2_UP, ACK_BAR2_DOWN,
              REPAIR_CLOSE2,REPAIR_OPEN2,
              FAIL2_TRANSIENT,FAIL2_PERMANENT) ;
```

Definition of Compound Elements

Compound elements are translated into functions of the appropriate type. For example, for compound condition:

```
function CAN_OPEN_CROSS return boolean;
```

Procedures for Initialization and Execution of the Module

Procedures for initialization of the module and execution of single step by all charts in the module.

```
procedure rail_cross_INIT;
procedure old_rail_EXEC_ALL;
```

Package Body

For each behavioral module, its package body contains implementation of those items that are declared in the package specification.

Definitions of State Status Types and Variables

Every non-basic OR-state has a status variable that indicates what substate is currently active. The status type is an enumeration type that actually lists all substates of the OR-state. Status variable gets value notaS (which is the default) when state S is not active.

```
type tpNORMAL_states is
      (notaNORMAL, OPEN, CLOSED, MOVE_UP, MOVE_DOWN);
NORMAL_isin : tpNORMAL_states := not NORMAL;
```

Schedule Timeouts Procedure

This procedure executes every execution cycle, and evaluates what timeouts should be triggered in the particular module. All timeout triggers are evaluated and the necessary timeouts are scheduled using the service SC_TMO (from the run-time library package TIMEOUTS).

```
procedure SCHEDULE_TIMEOUTS is
begin
   if ENMOVE_UP then
      SC_TMO(tmUP_FINISHED'address,
      FLOAT64(5) * SEC);
   end if ;
end SCHEDULE_TIMEOUTS ;
```

Body Stubs for Basic Activities

For each basic activity selected to be stubbed out with the Profile Editor's *Make Procedure* option, a procedure is created:

```
   procedure user_code_for_check_time1 is separate;
   procedure user_code_for_check_time2 is separate;
```

```
A separate procedure body is then placed in file user_activities.a
where you can add your code to implement the activity.
```

Functions Implementing the Compound Elements

The function CAN_OPEN_CROSS return BOOLEAN is

```
begin
   return(EXIT_CROSS1 or EXIT_CROSS2) and
      (not (IN_CRIT1 or IN_CRIT2));
end CAN_OPEN_CROSS;
```

Action Procedures

In some cases (depending on the modularity style), actions are translated into procedures:

```
procedure EXEC_GET_TRAIN_SPEED is
begin
   seti(TRAIN_SPEED'address,160);
end EXEC_GET_TRAIN_SPEED ;
```

State Enter/Exit Procedures

Depending on modularity style, the enter/exit (including history enter) sequences are grouped into procedures:

```
procedure entdef_OPEN is
begin
   NORMAL_isin := OPEN;
   gen(ACK_BAR_UP'address);
end entdef_OPEN ;
```

This example shows the default entering sequence (i.e. entering via a transition that goes to the edge of the state) for the OPEN state:

- Change parent status variable to OPEN
- Generate event ACK_BAR_UP; this reflects the static reaction
- Define in the model for state OPEN

```
entering/ACK_BAR_UP
```

State Execution Procedures

The EXEC procedure is actually the heart of the behavioral logic as described in the statecharts. Every non-basic state has an EXEC procedure that activates all the state-logic within a single execution cycle. The EXEC procedure takes care of in state transition, static reactions, and activation of substate EXEC procedures. The traversal is done hierarchically, starting at the very top state in the module and going down towards the basic states. In case of an AND-state, the orthogonal components are traversed sequentially one after the other.

```
procedure EXEC_BARRIER is
begin
   case BARRIER_isin is
      when NORMAL =>
      if FAIL_PERMANENT then
      exit_NORMAL;
      BARRIER_isin := DAMAGED;
       else
      EXEC_NORMAL;
      end if;
      when DAMAGED =>
      . . .
   end case ;
end EXEC_BARRIER ;
```

Module Initialization Procedure

The module “init” procedure is called once the executable is started, before running through any execution cycle. It performs various initializations, as shown in the example:

```
procedure RAIL_CROSS_init is
begin
   INIT_ACTIVITY(CHECK_TIME1’address,NONACTIVE,
      FALSE,ZNIL,ZNIL,ZNIL);
— initialization of the data structure
     — for an activity in the module
   Inst_BARRIER2.g_GEN_BARRIER_init;
— hierarchical call for initialization
      — of a generic instance in the module
...
end RAIL_CROSS;
```

The “init” procedure is exported to the module (TOP_LEVEL) where it is called by procedure LO_INIT that is responsible for initialization of the entire model.

Module Execution Procedure

This procedure activates a single execution cycle (step), once being called by LO_MAIN in the main module TOP_LEVEL. It activates the SCHEDULE_TIMEOUTS procedure to schedule potential timeouts, the user-written code for basic activities, and most importantly, it activates the hierarchical traversal of the state *EXEC* procedures by activating the EXEC procedures for all top-level statecharts that belong to the module.

```
procedure rail_cross_EXEC_ALL is
begin
   SCHEDULE_TIMEOUTS;
   if STARTED(CHECK_TIME1) then
      CHECK_TIME1_TASK.START;
   end if;
   EXEC_Chart_BARRIER;
  EXEC_Chart_RAIL_ROAD_CROSS;
  EXEC_Chart_TRAIN_MOVE;
end rail_cross_EXEC_all ;
```

File Structure In Ada: Control Files

Behavioral Modules

The behavioral modules are the heart of the code and implement the state/transitions logic as described by the statecharts. The specification is partitioned into behavioral modules in the compilation profile. For each specified module, two files are generated based on the user-defined module name.

The following module specification file exports all the specification objects defined in the module (to use by other modules), and the module execution procedure.

```
<module_name>__.a
```

The following module body defines all the local objects (events, conditions, data-items), and the procedures that implement the logic of the statecharts.

```
<module_name>.a
```

Top Level Module

The top-level module “wraps” all the behavioral modules into a single behavioral unit. It also defines all the global elements, i.e., those elements used by more than one module. It defines two procedures:

- **LO_INIT** - initialization of all the participating modules.
- **LO_MAIN** - execution of a single step of all modules.

The specification of the top-level module is identified below. It exports the global elements, the initialization and the execution procedures.

```
<profile_name>main__.a
```

Implementation of these procedures is found in the module’s body:

```
<profile_name>main.a
```

Main Procedure

This is the main scheduler that activates the behavioral modules. It consists of the main unit that instantiates all the other modules. In many cases where the generated code is not the backbone of the application, you might want to replace the supplied main procedure with your own application scheduler.

`main.a (main_dbg.a` in debugging mode`)`

User Supplemented Files

These files include all the stubs generated for the basic activities according to the compilation profile. Once the user-activities stubs file exists in the output directory, it is not overwritten, and a file `user_activities.a_tmp` is generated.

```
user_activities_.a (user_activities_.a_temp)
user_activities.a (user_activities.a.temp)
```

Transmitter Template

This file contains the hooks for the elements specified in the Profile with the “Hooks” option. Since you can modify the file, it is not overwritten. Instead, the file `user_transmitter.temp` is generated.

Interface Modules

- The *Symbol_table* file is generated only when the debug option is enabled in the Profile Editor (**Options > Global Profile Settings**). It includes symbolic information about the original model that is used by the debugger.

  `<profile_name>.dbg`

- The *PGE Interface* file is generated only if your code uses PGE to build mockup panels. This is actually the code that glues the panel to the behavioral modules:

  panel_transmitter.a

Info File

```
<profile_name>.info
```

The info file contains information about the translation process, the relevant portion of the model and the generated modules.

The info file contains the following information:

- **Compilation profile parameters**
- **Errors and warnings**
- **Cross reference table**—This table contains all the elements in the code and the names of the original elements they represent. This information is useful when supplementing the generated code. In cases where the same name is used in different charts, this cross-reference table is the only way to identify which code-element maps to the spec-element.
- **Interface report**—The interface report is a graphical diagram that shows the flow of information and control among the behavioral modules, and among the environment and the rest of the model.

dSPACE DS1103 ERP I/O Driver

This section explains how the DS1103 I/O driver is designed. It is intended for advanced users who want to enhance the DS1103 driver, or to port the driver to other dSPACE hardware. The topics are as follows:

Note

The Rational Statemate Embedded Rapid Prototyper (ERP) also supports dSPACE model DS1102.

The Rational Statemate Embedded Rapid Prototyper (ERP) supports dSPACE models DS1102 and DS1103. The dSPACE DS110 models are single-board solutions meaning the processor and I/O are located on the same cards.

The dSPACE interface enables you to:

- Generate C code from the Rational Statemate model, compile the code, and download it to the dSPACE system with a single click.
- Map model elements to the board I/Os.
- Generate dSPACE TRC files for use with dSPACE ControlDesk layouts.

Implementing the Driver

The driver consists of two parts. The first part describes the general infrastructure that can be used with other boards besides model DS1103. This infrastructure is defined in the file `stm_dspace.c`. It reads the signal mapping and selects the correct I/O function to be called by the driver function.

The second part, defined in `stm_ds1103.c`, deals with items specific to the dSPACE DS1103 board. The following sections describe each of these parts in detail.

General Driver File

To support a new card `DS<xxxx>` (where `<xxxx>` is the model number of the card), you must modify some sections of the `stm_dspace.c` file. Specifically, you must change the sections that use preprocessor directives in order to switch between the different cards (`"#ifdef DS<xxxx> …"`). In these sections, add calls to the functions `stm_ds<xxxx>_global_initialize(), stm_ds<xxxx>_get_driver_func(), and stm_ds<xxxx>_close_connection()`.

Driver Interface Functions

The dSPACE driver interface functions are declared in the file `stm_dspace.h`. The driver includes three functions:

- `stm_dspace_init()` - This function is called only once. It configures and initializes the signals defined in the ERP I/O mapping table. If mapping is invalid, a warning or an error message is generated (and sent to the dSPACE ControlDesk).
- `stm_dspace_driver()` - This function is called whenever data is read from, or written to, an I/O port. Therefore, this function is called very often and affects overall efficiency.
- `stm_dspace_close()` - This function should be called before closing the connection with the hardware.

The stm_dspace_init() Function

The `stm_dspace_init()` function parses the pin names and splits them into tokens. For example, “IOP 1-3” is a valid reference for an array of pins on the DS1103 board. The name is interpreted in the following way:

- **IOP** - The type of the signal
- **“1” and “3”** - The boundaries of the bit array

After pin name processing is performed, the function `stm_dspace_global_initialize()` is called. This function calls the initialization function of the special part of the driver.

The `stm_dspace_get_driver_func()` function selects the appropriate I/O access function. The function pointer is saved in the Rational Statemate element data structure and is used by the driver function.

The stm_dspace_driver() Function

The `stm_dspace_driver()` function is called with an element data structure as an argument. The function executes the function pointed to by the `drv_func` field of the element data structure.

The stm_dspace_close() Function

The `stm_dspace_close()` function calls `stm_dspace_close_connection()`, which in turn calls the specific driver function.

Driver-Specific Files

In addition to the `stm_ds<xxxx>.c` file, there are four other files that contain driver-specific information:

- `stm_ds<xxxx>_types.h` - Contains types definitions
- `stm_ds<xxxx>_conf.h` - Contains the driver configuration information
- `stm_ds<xxxx>_conf.c` - Contains the card configuration data structure
- `stm_ds<xxxx>_msg.h` -Contains error message handling and warning information

Handling I/O Signals

As an example, this section describes how to handle an A/D converter (ADC) signal on a DS1103 board. The following functions are used to handle I/O signals:

- `stm_ds1103_global_initialize()`
- `stm_ds1103_init_ADC()`
- `stm_ds1103_get_driver_func()`
- `stm_ds1103_drv_ADC()`

These functions are declared in the file `stm_dspace_1103.c`.

The stm_ds1103_global_initialize() Function

The `stm_ds1103_global_initialize()` function “attaches” some data structure to the `user_data` field of the Rational Statemate `report_elem` data structure.

The global variable `stm_ds1103_conf_var` is of type `stm_ds1103_conf`, which is defined as follows:

```
typedef struct stm_ds1103_conf {
  stm_ds1103_ADC_type          adc_t;
  stm_ds1103_ADC_signal        adc[20];
  stm_ds1103_DAC_signal        dac[8];
  stm_ds1103_SLAVE_PWM_type    slave_pwm_t;
  stm_ds1103_SLAVE_PWM_signal  slave_pwm[8];
  stm_ds1103_SLAVE_DF_type     slave_df_t;
  stm_ds1103_SLAVE_DF_signal   slave_df[4];
  stm_ds1103_SLAVE_FD_signal   slave_fd[4];
  stm_ds1103_SLAVE_PWMD_signal slave_pwmd[4];
  stm_ds1103_SLAVE_IOP_type    slave_iop_t;
} stm_ds1103_conf;
```

This data structure, defined in the file `stm_ds1103_types.h`, stores the user-defined card configuration. To change the card configuration, edit the file `stm_ds1103_conf.h` (as documented within the file itself).

As you can see in the `stm_ds1103_conf` structure declaration, there are 20 separately configured channels in an ADC signal. Therefore, the data structure includes an array of 20 elements of type `stm_ds1103_ADC_signal`.

The `stm_ds1103_ADC_signal` data structure is defined as follows:

```
typedef struct stm_ds1103_ADC_signal {
  stm_ds1103_ADC_type  *type;
  double               minimum;
  double               maximum;
  double               norm_const_A;
  double               norm_const_B;
} stm_ds1103_ADC_signal;
```

The fields of the `stm_ds1103_ADC_signal` structure are as follows:

- `type` - Points to a common part for all ADC signals. This part is used by the driver function for storing information regarding the current status of the ADC.

  The relevant structure declaration is as follows:

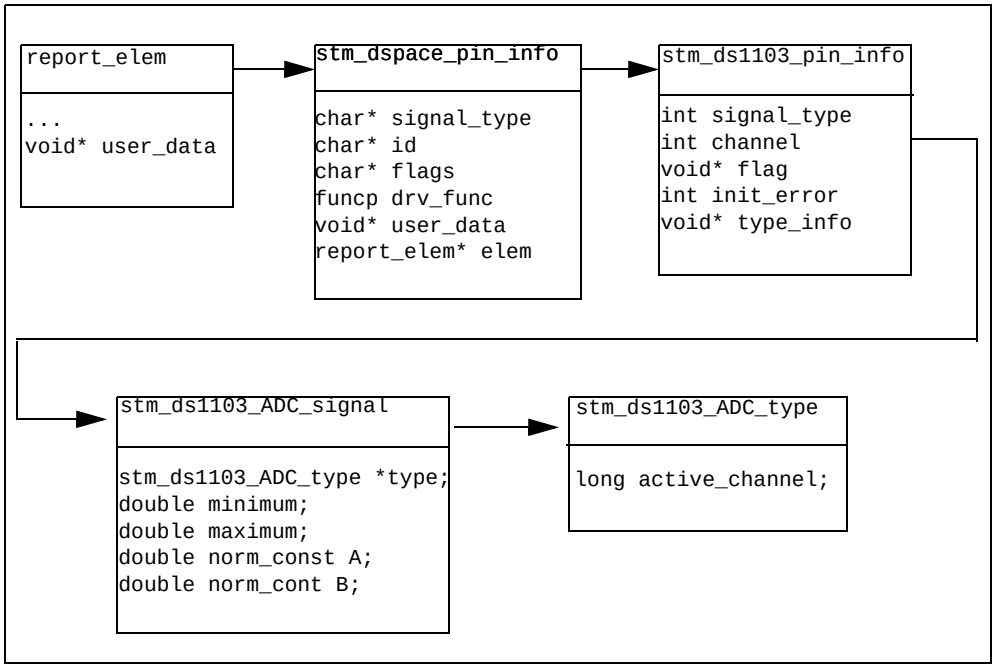
  ```
  typedef struct stm_ds1103_ADC_type {
    long active_channel;
  } stm_ds1103_ADC_type;
  ```

- `minimum` **and** `maximum` - Specify the range of the receiving signals.
- `norm_const_A` **and** `norm_const_B` - Used for normalization of the signal. This is necessary because the card returns values between –1.0f and 1.0f only.

The global variable `conf` stores lists of pointers to the I/O- mapped Rational Statemate elements. Each list stores the elements, which are mapped to the same signal type. It is used for configuration check and initialization.

After pin processing, the driver calls the initialization routines. The ADC signals are initialized by the `stm_ds1103_init_ADC()` function.

The following figure shows the result of the data structure allocation and attachment process performed by the initialization function.



The stm_ds1103_init_ADC() Function

The `stm_ds1103_init_ADC()` function performs the following tasks:

- Checks the validity of the port number
- Checks whether the signal is mapped as input in Rational Statemate
- Checks the Rational Statemate variable type against the port type
- Manages the MUX mask
- Calls the initialization function

The ADC channels are numbered from 1 to 20. If there is an invalid channel number, mapping is cancelled and the driver generates an error message. For example, in the Rational Statemate I/O mapping table, an element is specified as an input or output signal. Because the ADC signal is an input port, if there is a contradiction with the definition in Rational Statemate, an error message is generated and mapping is cancelled.

Model elements that are mapped to an ADC signal should be of type real or integer. In every other case, the driver generates an error message. Refer to **Signal Types** for information about Rational Statemate to dSPACE mapping.

Note that the first 16 channels are organized in 4 multiplexers. Every multiplexer has to be initialized with a selected channel. Multiplexer number 1 handles channels 1 to 4, multiplexer number 2 handles channels 5 to 8, and so on.

The stm_ds1103_get_driver_func() Function

The `stm_ds1103_get_driver_func()` function returns an I/O access function, which is used by the driver function. For example, for an ADC port, `stm_ds1103_get_driver_func()` returns the function `stm_ds1103_drv_ADC()`.

If the pin signal type is not supported by the driver, the function returns a `NULL` value. If an error occurs during initialization, the function returns the dummy function `stm_ds1103_drv_error()`.

The stm_ds1103_drv_ADC() Function

The `stm_ds1103_drv_ADC()` function is used by the model for I/O access. It is called at a defined polling rate. The implementation of this function should be very efficient.

The `stm_ds1103_drv_ADC()` function performs the following tasks:

- Changes the multiplexer channel (if necessary)
- Reads the value from the hardware
- Normalizes the value according to the user-defined variable range
- Stores the value in the Rational Statemate element structure

The first 16 ADC pins of the DS1103 hardware are handled by 4 multiplexers. The function changes the multiplexer active channels as necessary. The multiplexers' active channels are stored in the variable `active_channel`.

After setting the multiplexer, the function reads the value of the specified ADC channel. The read value, which is always between –1.0f and 1.0f, is translated to the user-defined range (specified in the file `stm_ds1103_conf.c`) using two previously calculated constants. The result is stored in the Rational Statemate element data structure.

Reserved C Words

This section lists the reserved words for the C programming language. If you use any of these words as names of elements, Rational Statemate renames the elements during code generation. For example, if you name a condition “AFTER,” when you generate the code, Rational Statemate automatically renames the element (“AFTER_”) so there is no conflict. The following table lists the reserved words.

ABS	ABSOLUTE	ACOS	ACOSD
AFTER	ALIAS	ALL	AND
ARCHITECTURE	ARCSIN	ARCTAN	ARCCOS
ARRAY	ASCII_TO_CHAR	ASIN	ASHL
ASHR	ASIND	ATAN	ATAND
ATAN2	ATAN2D	AT	ATOM
BA_LEN	BASETYPES	BEGIN	BITS_OF
BOOL	BOOLEAN	BUFFER	BUS
BYTE	CALLBACK	CASE	CHAR
CHAR_TO_ASCII	COMPOUND	CONCAT_BA	CONFIGURATION
CONST	COS	COSINE	COSH
COSD	COUNTER	DECIMAL	DELAY
DELETE	DELTA	DIGITS	DISCONNECT
DO	DOMAIN	DUPLICATE	ELSE
ELSIF	END	ENTERED	ENTITY
ENTRY	ERROR	EXCEPTION	EXIT
EXITED	EX_ENTERED	EXP	EXPORT
EXPAND_BIT	FALSE	FROZEN	FILE
FLOAT	FOR	FS	GENERATE
GOTO	GUARDED	HANGING	HEX
HR	IF	IN	INFINITE
INOUT	INT	INTERFACE	INTERNAL_RESET

INT_MAX	INT_MIN	IS	ISR
LABEL	LIBRARY	LIMITED	LINKAGE
LONG	LONG_MAX	LONG_MIN	LOOP
LOG	LOG2	LOG10	MAP
MAX_INT	MAX	MAXINDEX	MAXLONG
MAX_DIGITS	MEMORY_SIZE	MIN	MINUTE
MINUS	MIN_INT	MOD	MS
MUX	NAME	NOTA	NONACTIVE
NAND	NEW	NEXT	NOR
NXOR	NAND	NOT	NS
NULL	OPERATING	OF	ON
OPEN	OR	OTHERS	OVERFLOW
OUT	PANEL_ TRANSMITTER	PS	PORT
PRAGMA	PRIORITY	PRIVATE	PROC
PROCESS	QACTIVE	OUT	Q_PUT
Q_UPUT	OVERFLOW	RAISE	RANGE
RECORD	REGISTER	REM	RENAMES
REPORT	RETURN	REVERSE	ROUND
SEC	SELECT	SEVERITY	SHORT
SIN	SIND	SINE	SING
SINH	SIGNAL	SIGNED	SQRT
STORAGE_UNIT	STRING	STRING_INDEX	STRING_EXTRACT
STRING_LENGTH	STRING_TO_INT	SUBTYPE	SUCCESS
SYSTEM_NAME	TANGENT	TANH	TAND
TAN2D	TASK	TERMINATE	TEXT
TICK	TRUE	TRUNC	TRY_AGAIN
THEN	TO	TRANSPORT	TRY
TYPE	TYPES	UNITS	UNTIL

UNION	USE	USER_ACTIVITIES	US
VARIABLE	VOID	WAIT	WAIT_TIMEOUT
WHEN	WHILE	WINDOWS	WITH
WORD	XOR		

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W