Rational. Statemate

Software Code Generator Interface Manual

Rational Statemate
Software Code Generator Interface
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 IBM® Rational® Statemate® 4.6 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Software Code Generator Overview

Rational Statemate enables you to extend a Rational Statemate model by adding handwritten code to the generated code. You may want to use this feature to:

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

This manual documents the pre-Rational Statemate 1.2 method for supplementing user code to the generated code. This method adds code to the generated C or Ada code by modifying the `user_activities` files. This manual also explains how to use the callback mechanism to communicate with external code.

Although Rational Statemate continues to support this method, you should use the new methods for supplementing code. For information on these methods, refer to "Adding User Written Code" in the *Code Generator Reference Manual*.

Code Architecture and the Generator

To obtain a working prototype of the system, you can extend the Rational Statemate-generated code by implementing those elements and aspects of the system's behavior that have not been explicitly defined by the controlling statecharts and mini-specs.

The Code Generator does not implement primitive activities whose behavior is not described by a statechart or mini-spec. The supplemental code that you write can be interfaced with the Rational Statemate code. This interface describes when and how these primitive activities "accept" synchronization actions applied to them (start, stop, suspend, resume). It also describes when and how they produce and consume items that flow between them and the rest of the system.

Similarly, the user can implement the interface between the prototype components of the system and its environment since the Code Generator has no information about the structure of the environment's activities.

The Code Generator supports several structures that help you extend the model, such as the following:

- Templates needed to implement the primitive activities
- The callback mechanism - sensing any change in the model
- A set of standard procedures that provide all the necessary flows of events, conditions and data-items between the environment and primitive activities and the rest of the system
- Routines to synchronize the primitive activities with the rest of the system

User Supplemented Files

The user-activities file includes all the stubs generated for the basic activities according to the compilation profile.

C Code

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.

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

Ada Code

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)
```

Supplementing the C Code

When supplementing the generated code with user additions, it is important to add the additional compilation statements to `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.

```
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. If you add any libraries, add them to the `user_libs` file.

Stubs: Procedures or Tasks

Code Generator enables you to create stubs in the generated C or Ada. A stub serves as a "placeholder" where you can insert handwritten or vendor-supplied code into the profile.

You can define which **primitive activities** will be stubs, and also choose how the stubs are modeled, as Procedures or Tasks.

Note

Rational Statemate uses the signal and port assignments that were in place before you made the element a stub.

Make PROCEDURE

Activities implemented as Procedures are *not* expected to run in parallel with other activities (including control activities). Such an activity can be characterized as:

- Short living
- Not interruptible (does not interact with others)
- Not stopped, suspended, or resumed from outside (self-terminating).

A Procedure is similar to a Rational Statemate action and is usually of a transformational nature, rather than reactive. If started at some step, it finishes at the same step. Activation of such an activity is very similar to calling a routine—until it stops, others cannot advance or be activated.

To Make a Procedure, complete the following steps:

1. Select **View > Show Scope as Tree**.
2. Select the **Activity-chart** that contains the activity box that you want to make a procedure.
3. Select **View > Show Boxes**.
4. Select the **primitive activity** you want.

5. Click **Procedure** or select **Edit > Make Procedure.**

   The Code Generator labels the activity with a **P** and a graphic as shown in the VIBRATION activity example in the following figure.

Make Task

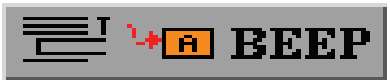
Activities implemented as Tasks are expected to run in parallel with other activities, particularly in parallel with the advance of the Statecharts. Tasks may interact with other activities and be synchronized with them. You can also apply action to them i.e., stop, suspend, or resume.

To Make a Task, complete the following steps:

1. Select **View > Show Scope as Tree**.
2. Select the **Activity-chart** that contains the activity box that you want to make a task.
3. Select **View > Show Boxes**.
4. Select the **primitive activity** you want.

5. Click on the **Task** icon in the toolbar or select **Edit > Make Task**.

   The Code Generator labels the activity with a `T` and a graphic as shown in the BEEP activity example in the following figure.



Note

For information on integrating handwritten **C**, refer to Supplementing Generated C.

Note

For information on integrating handwritten **Ada**, refer to Supplementing Generated Ada.

Hooks and Callbacks

The Code Generator also provides a powerful mechanism that allows you to hook user-actions or procedures to any change in the specification during execution. This mechanism is very useful when you wish to tie your external environment to the behavior represented by the generated code.

Unlike stubs, which simply serve as placeholders for external code, hooks generate callback functions and actually communicate with external code. They become activated when there is a change in the selected element such as an event, condition, or state.

Hooks in Generated Code

To create hooks, select **Edit > Hooks** and then refer to either the Ada or C section below.

Hooks in C

In C, all Rational Statemate elements can be incorporated into a callback routine. Normally, if you are using elements in callbacks, it is not necessary to create hooks. However, for states, you need to tell Rational Statemate in advance which states are going to be used in a callback, so an appropriate ID can be generated.

For more information on how to use callback routines in the user code to track model changes, refer to **Callbacks for Compound Elements.**

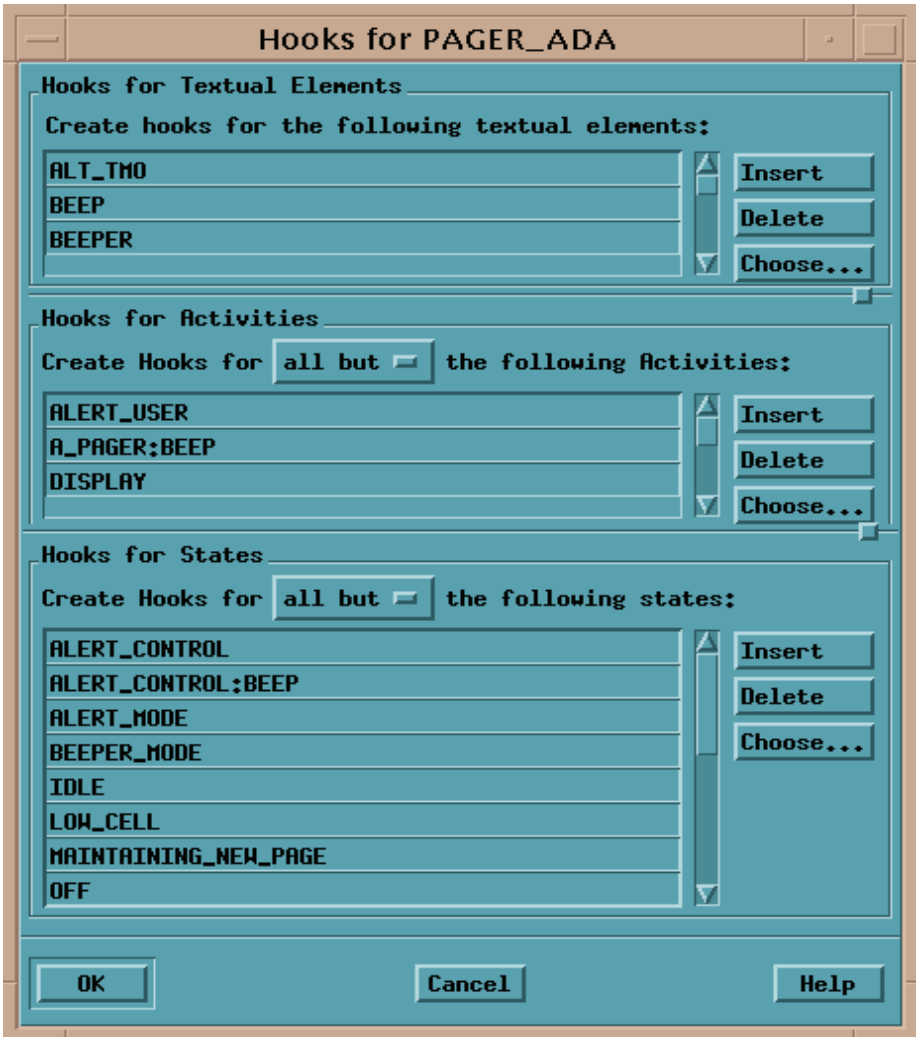
Hooks in Ada

In Ada, if you are going to use any Rational Statemate elements in a callback routine, you must first create hooks for them. This is done through the Hooks Data Editing screen shown in the following figure.

For more information on how to use callback routines in the user code, refer to **Interfacing With the Rational Statemate Model.**

Ada allows you to create hooks for the following:

- Textual elements
- Activities
- StatesTextual Elements


Hooks for PAGER_ADA
Hooks for Textual Elements
Create hooks for the following textual elements:
ALT_TMO
BEEP
BEEPER
Insert
Delete
Choose...
Hooks for Activities
Create Hooks for
all but
the following Activities:
ALERT_USER
A_PAGER:BEEP
DISPLAY
Insert
Delete
Choose...
Hooks for States
Create Hooks for
all but
the following states:
ALERT_CONTROL
ALERT_CONTROL:BEEP
ALERT_MODE
BEEPER_MODE
IDLE
LOW_CELL
MAINTAINING_NEW_PAGE
OFF
Insert
Delete
Choose...
OK
Cancel
Help

Hooks for Textual Elements

To create hooks for textual elements, complete the following steps:

1. Select **Choose** in the Textual Elements window as shown in the next figure.

   The **Element Selection for Textual Hooks** dialog box opens.

2. Use the **Type**, **Sub-Type,** and other buttons to locate the textual elements you want, and then click **Filter** to display them in the Name field.

3. Select one textual element at a time by highlighting it or multiple elements by pressing **Ctrl** while selecting. Select all the elements in the current list by clicking **Select All**.

4. Press **OK**. The selected elements appear in the list of hooks.

Activities

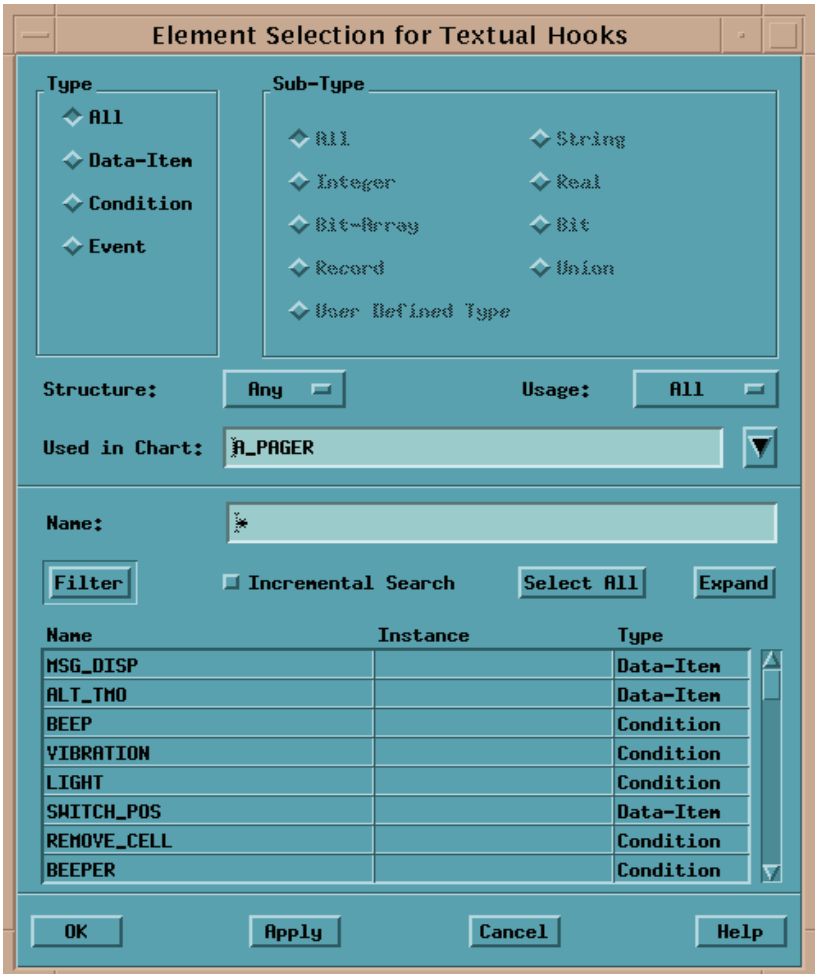
To create hooks for activities, complete the following steps:

1. Select **Choose** in the Activities window. The **Selection of Activities** dialog box opens.

2. Use the **Defined in Chart** and **Name Pattern** buttons to locate the activities you want, and then press **Filter** to display them in the Name field.

3. Select one activity at a time by highlighting it or multiple activities by pressing **Ctrl** while selecting. Select all the activities in the current list by clicking **Select All** .

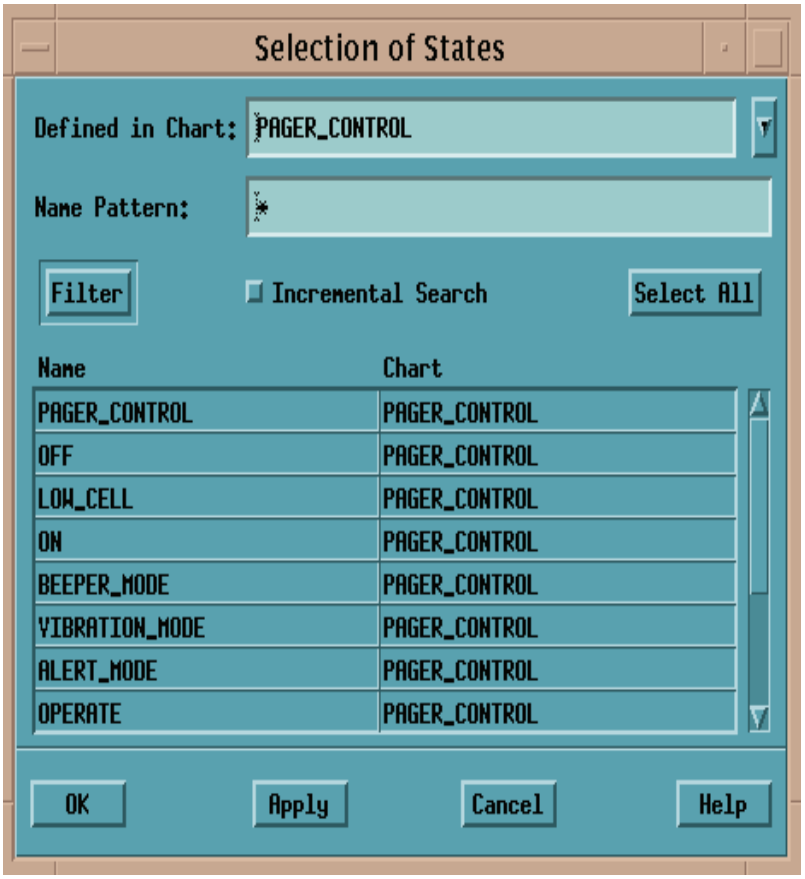
4. Click **OK**. The selected activities appear in the list of hooks


Element Selection for Textual Hooks
Type
All
Data-Item
Condition
Event
Sub-Type
All
String
Integer
Real
Bit-Array
Bit
Record
Union
User Defined Type
Structure:
Any
Usage:
All
Used in Chart:
A_PAGER
Name:
*
Filter
Incremental Search
Select All
Expand
Name
Instance
Type
MSG_DISP
Data-Item
ALT_TMO
Data-Item
BEEP
Condition
VIBRATION
Condition
LIGHT
Condition
SWITCH_POS
Data-Item
REMOVE_CELL
Condition
BEEPER
Condition
OK
Apply
Cancel
Help

States

To create hooks for states, complete the following steps:

1. Select Choose in the **States** window. The **Selection of States** dialog box opens.



2. Use **Defined in Chart** and **Name Pattern** to locate the states you want, and then press **Filter** to display them in the Name field.

3. Select one state at a time by highlighting it or multiple states by pressing **Ctrl** while selecting. Select all the states in the current list by pressing **Select All**.

4. Click **OK**. The selected states appear in the list of hooks.

Supplementing Generated C

This section explains how to supplement the Rational Statemate-generated code with handwritten code.

Implementation of Primitive Activities

The file `user_activities.c` contains templates for every primitive activity that is to be implemented in the compilation profile's activity-stub options. Each selected primitive activity contains a stub with the following format:

```
void user_code_for_<activity_name>()
{
/*      Parameters :
        Output <output data-elements>;
        Input <input data-elements>;

        Input/Output <Inout data-elements>;
*/
}
```

The parameters list describes the interface of the activity to the rest of the model. Note that these are not parameters in the programming-language sense. The parameters list is actually a reference list that shows the context of the activity to be implemented in the model.

```
void user_code_for_FFT()
{
/* Parameters

--      Input double sonar_datal;
--      Input double sonar_data2;
--      Output double processed_data;
/*
}
```

The previous example shows a primitive activity that represents an FFT filter. The data-items `sonar_data1, sonar_data2` are flowing into FFT, and the `processed_data` is flowing outside. This is actually the interface of the FFT activity with the rest of the model. Mathematical processing functions such as an FFT filter, are typical cases where something is implemented as a primitive activity, and the algorithm could be taken from an existing library.

Once the `user_activities.c` file is generated, it is not overwritten when the code is regenerated. In subsequent generations of the code, a `user_activities.c_temp` file is generated. If new templates are generated, they should be merged from `user_activities.c_temp` into `user_activities.c`.

Note

Empty stubs stop right after activation and the **sp** (activity) event is generated in the next step.

User Init and User Quit

The file `user_activities.c` contains the following procedure template:

```
void user_init ( )
{
}
```

The code calls this procedure before the very first step is taken in the translated model. Therefore, you can use it for many types of initializations.

For example, you can add an actual piece of code to initialize various global structures in the code supplied for primitive activities, to open windows, etc.

- Another important option is to initialize specification elements. Recall that all events, conditions and data-items used in the specification have the following default values:
- events - *not active*
- conditions - *false*
- integer, real and bit-array - *zero*
- textual data-items - *blank string*

The default value is used when there is no explicit initialization of an element before it is referenced in an expression. However, you might wish to intentionally leave an element uninitialized in the specification because you do not know the precise initial value. In such a case, you want to be able to run the same prototype code with different initial values of the element and to choose an appropriate one in a “trial and error” process.

Once you choose an initial value, you can add it to the specification. In other words, you tune the system specification by working with the prototype derived from it. For example, if you want to assign an initial value of true to the condition *FAULT*, and a value of 50 to the integer data-item *LOW_BOUND* which both belong to a chart, you transform the template into the following procedure:

```
void user_init()
{
       setc(&FAULT,true);
       seti(&LOW_BOUND,50);
}
```

Execution of the code may come to a point where all activities of the prototyped system become non-active and thus the system must finish its work. This may be caused by various reasons: self-termination of activities, explicit or implied actions stop or the command **quit** entered when running the Code Generator Debugger.

In all cases where the system stops, the code performs a call to the procedure `user_quit`, intended to support a graceful termination of the user extensions. The template of this procedure resides in the `user_activity.c` file:

```
void user_quit()
{
   ..
}
```

Consider an example in which the prototype code is connected to a graphical mock-up of the operator display. Suppose that among the user's extensions there is a task responsible for I/O interface between the code and the display. When a soft button is "pushed" on the display, the task accepts an interrupt from the mouse and translates it into generation of an event sent to the prototype code. When the system stops, this task must terminate. To achieve this goal, place an abort statement for this task in the template `user_quit`.

Synchronization of Primitive Activities

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

User-written procedures are called when the system starts the corresponding activity (for example, `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. Therefore, the Code Generator provides two types of user activities: simple procedures and tasks.

Procedures

A procedure-activity is executed in a *one-shot* - it is not preempted until it returns. Therefore, you should use this mechanism for instantaneous activities (activities that execute for a short period of time). Typically, these activities perform short calculations or non-blocking I/O operations, like displaying data or drawing graphics. If the procedure mechanism is used for continuous calculations or delayed I/O, it blocks the rest of the prototype from reacting properly to incoming events. Since a procedure-activity is not being preempted, the suspend, stop and resume actions do not have any effect on them. When a procedure-activity returns, the sp!( ) event is sent to the controlling code.

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 the **Scheduler Package** for more details).

The scheduler package allows you to define C functions as concurrent routines or co-routines. An activity which you choose to implement as a task is invoked 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:

- `sched_wait_for_event(event)`
- `sched_delay(delay_time)`
- `scheduler()`

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

The `sched_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) {
    sched_wait_for_event(&gevSENSE);
    /* here you are supposed to check status.*/
   printf("Time generated\n");
  }
} /* end sense_start */
```

The `sched_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 . . .
              }
              sched_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 allows other activities to proceed and the calling activity resumes 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 invoked as co-routines. The statechart code itself is a task, which runs concurrently with the other invoked tasks.

Controlling all those tasks is the responsibility of statecharts which issue different actions to the different activities (for example, 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 sections.

Creating a Task

In order to create a new task, call

```
task_entry *sched_create_task(proc, param, stoproc, stoparam, model_context,
inst_context)
   void_funcp proc;/* proc(param) is activated as task*/


   unsigned long param;

   void_funcp stoproc;/* stoproc(&stoparam) is called
                  when the task terminates*/


   char *stoparam;


   void *model_context;
/* when several separately generated models are integrated with user-written
main(), this parameter identifies the model to which the task belongs*/


   void *inst_context;
/* when task is connected to activity in a generic activity-chart, this
parameter identifies the currently executed instance*/
```

This routine initializes a task and returns to its caller a descriptor, which is to be used in further references to this task.

The task is initialized with a 40 KB stack. In case of stack overflow, a message is printed and the task is aborted.

Controlling a Task

Calling `sched_create_task` puts the new task in stopped status. To make it ready, call:

```
sched_start(te)
```

where *te* is a pointer to the descriptor returned by `sched_create_task` for this task. In a similar manner, there are routines which take the other Rational Statemate actions on such tasks:

```
sched_stop(te)
sched_suspend(te)
sched_resume(te)
```

Aborting a Task

The following routine aborts the task identified by the descriptor `te`. It de-allocates its stack and its control blocks:

```
sched_abort_task(te)
int te;
```

The `sched_abort_task` function should be called only if the task is not invoked again. If this is not the case, use the `sched_stop` function.

Special Services

The following routine causes the calling task to be delayed for the specified number of seconds. It also causes rescheduling so that other tasks may advance while this task is delayed.

```
sched_delay(time_amount)
double time_amount ;
```

Each task can request a time slice which is the maximal time period for which the task may hold the CPU without rescheduling another task. The request is made by calling the following routine:

```
sched_slice(time_slice)
double time_slice ;
```

To disable the time-slicing for some task, it should call this routine with a time slice of 0.0. This is also the default with which each task is initialized.

To disable and enable the time slicing mechanism for all the tasks, the following routines may be called:

```
sched_enable()
sched_disable()
```

Note that when time slicing is disabled, delays, timeouts and scheduled actions initialized by the scheduler (either before or after `sched_disable` has been called) cannot end until `sched_enable` is called. We recommend disabling the time slicing when performing print statements. Unexpected results may occur.

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 `sched_delay`. The calling task then becomes delayed.
- When a task calls a `wait_for_event` service. The calling task then becomes delayed.

  ```
  sched_wait_for_event(EVENT)
  event *EVENT;
  ```

- If a task enables the time slice option, which invokes the scheduler implicitly after a time period.
- After the task function performs a return, it stops.

Operations like `sched_create_task`, `sched_start`, `sched_stop`, `sched_suspend` and `sched_resume` do not cause rescheduling.

Restrictions

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

Interfacing With the Rational Statemate Model

The model elements are abstract data types that can be accessed by procedures produced by the Code Generator.

There are two ways to interface with the Rational Statemate model:

- Procedures to modify values of events, conditions and data-items. You have to call them in your code whenever you wish to perform the manipulations on these elements. These procedures are discussed in the following subsections.
- Set callback functions to respond to changes in the system. The code guarantees that such a callback is called whenever the corresponding change occurs. This can be, for example, displaying a message on the screen or assignment of an appropriate value to a variable used in the user code.

Referencing Model Elements

Communication between the user-defined 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:

- Events and conditions are represented as bytes
- Data-items are represented as integers, reals, strings or unsigned
- User-defined types are derived from basic data-types

The following table shows the mapping between the Rational Statemate basic types and the corresponding C types:

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

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, i.e., 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 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 ;
.
.
```

Element Names in the Output Code

The element name in the object code is the same as in the Rational Statemate model. If a user-defined element name is not unique, or if it conflicts with a reserved word in the target language, it is changed in the code to contain a prefix that denotes its type and scope. Since the Rational Statemate scopes are different from the modules in the output code, the names are not identical. This avoids any ambiguities that might result from name duplications. The naming convention is shown below:

```
prefix<STATEMATE_NAME>
```

Where prefix is determined as follows:

1. **The type**:

   ev - event

   co - condition

   di - data-item

   For activities the notation is:

   ```
   acy_<ACTIVITY_NAME>.
   ```

2. **To resolve ambiguities**:

   If two elements have the same name in a module, a number is added to the prefix to resolve the ambiguity. If an ambiguity occurs, refer to the cross-reference table in the info-file to determine which is the correct element.

   Example of two data-items with the same Rational Statemate name (Z):

   ```
   di1Z, di2Z
   ```

   In this case, you should look in the cross reference table to identify which one belongs to which Rational Statemate scope.

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 ;
```

Generating Events

Events are primitive elements and are special in the sense that software languages do not support them directly. An event is active, or "high," for only one step unless it is regenerated. The intrinsics library supports this behavior via the "`gen`" function.

Once an event is generated via "`gen`," the intrinsics runtime module is set and resets the event at the right time. An active event signifies a value of "1" in the byte that represents that event.

Example:

```
gen (&event1);
```

Note

The function expects an address of an event element. Direct setting of an event, i.e., `event1` = true, causes the code to behave incorrectly since the intrinsics module does not handle this situation.

Assigning Values to Rational Statemate Elements

Since model elements follow Rational Statemate semantics, their assignments should be synchronized to the beginning of the next step (cycle). A direct assignment such as:

```
X = Y + 1;
```

might result in racing condition especially when the data/condition element is shared between two concurrent activities. The synchronized assignments are implemented via a set of service calls supported by the intrinsics library. The following is the synchronized assignment call for the above assignments

```
seti(&X, Y + 1);
```

There are cases where using direct or deferred assignments do not make a difference, however, it is always recommended that you avoid using direct assignments.

The intrinsics library offers a set of procedures that apply deferred assignments to the different types of Rational Statemate data-items. The assignment interface calls for each Rational Statemate type are listed below:

1. Bit:

```
   void setbit(ba, val)
bit *ba;
bit val;
```

Example: set?(&FAULT,true);

2. Condition:

```
void setc(c_p, val)
       boolean *c_p;
       boolean val;
```

Example: setc(&FAULT,true);

3. Real data-item:

```
void setd(d_p, val)
       real *d_p;
       real val;
```

Example: set?(&FAULT,true);

4. Integer Data-item:

```
void seti(i_p, val)
       int *i_p;
       int val;
```

Example: seti(&LOW_BOUND,50);

5. String Data-item:

```
void sets(s_p, val)
       char *s_p;
       char *val;
```

Example: sets(&FAULT,true);

The following APIs can be used to set arrays or slices of arrays. They all take source and destination arrays, and length. For example, in case of a slice assignment such as a1(3..5):=a2(1..3), the following call does:

```
set_array_<type>(&a1[3], &a2[1], 3);
```

Arrays:

```
void gen_array (ev_p, len)
   event *ev_p;
   int len;

void set_array_condition(trg, src, l)
   condition *trg;
   condition *src;
   int l;

void set_array_int(trg, src, l)
   int *trg;
   int *src;
   int l;

void set_array_real(trg, src, l)
   real *trg;
   real *scr;
   int l;

void set_array_string(trg, src, l,l_trg_str,l_scr_str)
   char *trg;
   char *src;
   int l; /* length */
   int l_trg_str;
   /* string length of target */
   int l_src_str;
   /* string length of source  */

void set_array_ba(trg, len, trg_ba_l, src, len_src,
        src_ba_l)
   bit_array *trg;
   int len;
   /* length of target */
   int trg_ba_l;
   /* bitarray length of target */
   bit_array *src;
   int len_src;
   /* length of source */
   int src_ba_l;
   /* bitarray length of source */
```

Example

The following is a supplemented basic activity that processes X,Y and generates two events according to the result: `PROCESS_OK` and `PROCESS_ERROR`.

```
void user_code_for_filter()
{
    /* Parameters :
    Input int X ;
    Input int Y ;
    Output event PROCESS_OK ;
    Output event PROCESS_ERROR ;
    */
}
```

The supplemented procedure is shown below:

```
void user_code_for_filter()
{
    /* Parameters :
    Input int X ;
    Input int Y ;
    Output event PROCESS_OK ;
    Output event PROCESS_ERROR ;
    */
    apply_filter(X, Y, Z);
    if (in_range(Z))
    gen(&PROCESS_OK);
    else
    gen(&PROCESS_ERROR);
}
```

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; The following table lists the available structures. 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]

Note

In $STM_ROOT/etc/prt/c/types.h you find the statement: type def 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;
```

Structured Elements

For complex *Data-Items* in the Rational Statemate model (e.g., a *Data-Item* record) a type is defined for the *Data-Item*. This happens even if the *Data-Item* is not defined as a user type. The type declaration (typedef statement) is placed in the same *.h* file as the external declaration for the *Data-Item*. These implicitly defined types are treated the same way as ordinary User-Defined Type*s*.

Rational Statemate defines structured elements and user-defined types in a file called `<profile name>_type_utils.c`, and assigns names to the types based on the name of the *Data-Item* and the characters *ty* as a suffix. For example,

- RECORDty RECA;
- UNIONty UNIONA;
- USER-DEFINEDUSER[1];

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 are 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.

Enumerated Types

An Enumerated Type is a user-defined type with a finite number of values.You cannot directly define a data item as an enumerated type. First, define the data item as a user-defined type, and then define the user-defined type as an enumerated type. You define the values for the enumeration in the “Definition” field of the Properties by listing the values in brackets separated by commas. For example,

```
{SUN,MON,TUE,WED}
```

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.

Note

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. These are summarized in the Constant Operators Table and the General Operators Table.

Constant Operators	Notes*
en_first(T)	First enumerated value of T
en_last(T)	First enumerated value of T
* Parameters to these constant operators are user-defined types that were defined as enumerated types.	

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

User-Defined Type Functions

There are a number of functions provided for manipulating Rational Statemate model variables that should be used when augmenting the Rational Statemate generated code.

Note

Rational Statemate variable values may be read by checking the correct variable name. Value changes, however, should not be made directly to the same variable. All value changes are made through a list of variables to be updated. This list is affected through a variety of functions created by the code generator.

The call

```
seti (&variable_name, value);
```

is used for setting any (primitive) integer variable to a desired value. Other similar calls provide the ability to set conditions, strings, real numbers, etc.

In addition to these general functions, the Code Generator creates similar functions which are specific to each UDT.

Every User-Defined Type has the following functions defined for it:

```
void set_<type>(A,B)
<type> *A,*B;
```

It uses the update list to assign A:=B. The user-code should not make direct assignments to Rational Statemate variables. Use the following functions to test for equality:

```
boolean eq_<type>(A,B)
<type>  A,B;
```

Returns TRUE if the elements A and B are equal.

For every type that has a `corresponding_event` declaration, the following functions are defined:

```
boolean all_<type>(A)
<type>_event  A;
```

These functions test to see if all the events that form A are currently generated. This example only applies to `RD<element>` and `WR<element>`.

```
void gen_<type>(A)
<type>_event  *A;
```

These functions generate all the events in A. This only applies to `RD<element>` and `WR<element>`.These functions create and initialize complex functions.

```
void init_<type>(A)
```

Queue Functions

Queues are implemented as linked lists in the generated code. Each node in the linked list contains a pointer to an element. Access the lists by using the access functions described in the following table.

Queue Function	Description
q_length	This function returns the length of the queue Q.
	`int q_length( (QUEUE) Q)`
q_get	This function removes an element from the head of the queue and puts it into element. The status is TRUE if there was an element to get. If the status is passed as NULL, no status value is returned.
	`q_get( (QUEUE *) Q, (<type> *) element,(boolean *) STATUS)`
q_peek	This function is the same as q_get, except the TOP queue element is not removed from the queue it is only copied.
	`q_peek( (QUEUE *) Q, (<type> *) ELEMENT, (boolean *) STATUS)`
q_initialize	If you want to create your own queues the queue should be initialized before use. The queue starts empty.
	`q_initialize( (QUEUE *) Q, (int) sizeof(ELEMENT))`
q_put	This function puts element at the end of Q.
	`q_put((QUEUE*)Q,(<type>*)ELEMENT,(int)sizeof(ELEMENT))`
q_uput	This function puts element at the head of Q.
	`q_uput((QUEUE*)Q,(<type>*)ELEMENT,(int)sizeof(ELEMENT))`
q_flush	This function deletes all the elements in Q.
	`q_flush( (QUEUE*) Q)`

All the queue manipulation is done using memory allocation for elements added to the queue and freeing the memory when the elements are removed or deleted from the queue. The memory allocation and freeing is done automatically so there is no need to preserve the value of element after the `q_put()` or `q_uput()` function is called.

The queue manipulation functions work through the update list, so if some elements are put into a queue, they are not in the queue until the end of the step.

String Functions

```
char *string_extract(str,index,length)
       char *str;
       int index;
       int length;

int string_index(str,offset,sub_string)
       char *str;
       int offset;
       char *sub_string;

char *string_concat(str_dest, str_src)
       char *str_dest;
       char *str_src;

int char_to_ascii(str)
       char *str;

char *ascii_to_char(int_val)
       int int_val;

char *int_to_string(int_val)
       int int_val;

int string_to_int(str)
       char *str;

int string_length(str)
       char *str;
```

READ, WRITTEN, CHANGED, TRUE, and FALSE in Complex Data Types

Sensing how the *rd() , wr()* , and *ch()* events, as well as *tr()* and *fs(),* are related to complex *Data-Types* requires the Code Generator to create some additional C *Data-Types* and variables. These type declarations are put in the same header file as the type declaration for the complex type itself. (Remember that the names and number of header files varies, based on whether the **Separate File per Statechart** option was selected in the profile under **Options > Module Settings.**)

Additional types are required for *wr()* and *rd()* as well as additional variable declarations. These additional types have similar structures to the complex types themselves. The type have fields with the same name but the types of the fields are EVENTS. These are used to store the event of WRITTEN or READ for each of the fields of the complex type.

A separate variable is defined for *rd()* and *wr()*. Each is defined as a type COMPLEX_VARIABLE_NAME_event. The name of the variables is the same as the complex type itself, with either *wr* or *rd* prefixed, as appropriate. One or more fields in the *wr* or *rd* variable is set when these fields are READ or WRITTEN. This is found in the procedure for the action that sets or uses the referenced field values.

Detecting Changes in Value

If the event expression *ch()*, *tr()* or *fs()* is applied to an expression that uses a *Data-Item* or condition, it is necessary to preserve the previous value of the element. This is done by maintaining a duplicate copy of the element that is updated at the end of every step. (This makes detecting *ch()* on a large array or complex record a very slow process, because it takes time on every step.) The duplicate copy of the variable is named *prev_<element>*.The following table lists the affected element types.

Element Type	Variable Name	Purpose
User-Defined Type	<type>	Define base *Data-Item*
User-Defined Type	<type>_event	Define rd() and wr()
Data-Item	<element>	Holds value of *Data-Item*
Data-Item	WR<element>	Holds wr() event for *Data-Item*
Data-Item	RD<element>	Holds rd() event for *Data-Item*
Data-Item	prev_<element>	Holds old value of the *Data-Item*
Element—is the name of the element as listed in the .info file. This is normally the same as the Element Name in the Properties, unless there are duplicate Element Names.		
Type—is the name of the User-Defined Type. It is also listed in the info file.		

Callbacks to Track Model Changes

The Code Generator provides a powerful mechanism that allows you to hook user-actions or procedures to any change in the specification during execution. This mechanism is very useful when you wish to tie your external environment to the behavior represented by the generated code.

For each type of Rational Statemate element, there is a callback routine. For instance, `set_state_cbk` or `set_event_cbk`.

Callbacks are called when the element changes. In general:

```
    set_<element>_cbk(g_addr,el_p,callback_routine,
            callback_param)
genptr g_addr;
genptr el_p;
void(*callback_routine) ();
int callback_param;
```

Data-items

Callback functions are available for all primitive types:

- Event
- Condition
- Bit
- Integer
- Real
- String
- Bit-Arrays

With reference to the callback routine:

```
set_<element>_cbk (g_addr,el_p,callback_routine,callback_param)
```

where each element is identified by two parameters: `g_addr`, the instance in which it is defined, and `el_p`, the address of the element itself.

For `g_addr`, set the value to 0 in most cases. However, when referring to elements in generic charts, set the value to 1. The `el_p` indicates the address of the element. Make sure you insert the correct address of the data item.

Note

Check in <module name>.h for the correct reference to element names.

`callback_routine` is the address of a C function.

`callback_param` is a parameter used by the callback routine when called.

The callback function has the following interface:

```
callback_routine void (value, callback_param)
```

where `value` holds the new value of the element.

Note

The type of the value parameter depends on the element with which the callback is associated: for data-items of type string, it returns a pointer to a string (char *), for real data-items, value is of type double. If the element is an event, the value parameter does not exist.

The `callback_param` is the same value that was installed when setting the callback.

Note

Use the `callback_param` when you want to associate one procedure to a number of elements (instead of writing a separate procedure for each element). The returned value allows the callback procedure to perform differently on each element.

- For states. the value is 1-entered, 0-exited
- For activities, the value is 0-nonactive, 1-active, 2-hanging

In the following interfaces, these conventions are used:

```
public void set_event_cbk(g_addr, el_p, func_p, param)
genptr g_addr;
genptr el_p;
funcp  func_p;
int param;

public void set_cond_cbk(g_addr, el_p, func_p, param)
genptr g_addr;
genptr el_p;
funcp  func_p;
int param;

public void set_int_cbk(g_addr, el_p, func_p, param)
genptr g_addr;
genptr el_p;
funcp func_p;
int  param;

public void set_real_cbk(g_addr, el_p, func_p, param)
genptr g_addr;
genptr el_p;
funcp func_p;
int param;

public void set_str_cbk(g_addr, el_p, func_p, param)
genptr g_addr;
genptr el_p;
```

```
funcp  func_p;
int param;

public void set_ba_cbk(g_addr, el_addr, len, func_p,
                 param)
genptr g_addr;
genptr el_addr;
int             len;
funcp func_p;
int             param;

public void set_bit_cbk(g_addr, el_p, func_p, param)
genptr g_addr;
genptr el_p;
funcp func_p;
int  param;
```

Assume that conditions *c1*, *c2* are in your specification and you want to monitor them. The callback routine looks like:

```
void show_conditions(c_val, c_num)
int c_val;
int c_num;
{
      if (c_num == 1)
            printf("Condition C1 was set to
            %s\n",
            (c_val==0) ? "true" :
            "false");
      else
            printf(Condition C2 was set to

            %s\n",
            (c_val==0) ? "true" :
            "false");
      }
```

The callbacks should be set during the initialization. The most logical place is within the `user_init` procedure (see `user_activities.c`).

```
void user_init()
{
      set_cond_cbk(0,&c1, show_conditions, 1);
      set_cond_cbk(0,&c2, show_conditions, 2);
}
```

Note that each element can be associated with a number of callback routines. This is why the callback setting functions is called `prt_add_cbk`, since it adds callback functions.

States

```
public void
set_state_cbk(g_addr,state_id,func_p,instance_name)
       genprt g_addr;
       int state_id;
       funcp func_p;
       int param;
```

Note

If you are using state callbacks, it is necessary to tell Rational Statemate in advance which states are going to be used in callbacks, i.e., “hook them out” so an appropriate ID can be assigned. This is not necessary for other element callbacks.

Activities

```
public void set_act_cbk(g_addr, el_p, func_p, param)
       genprt g_addr;
       genptr el_p;
       funcp func_p;
       int param;
```

Callbacks for Compound Elements

```
public void set_compevent_cbk(g_addr, el_p, func_p,
                  param)
          genptr g_addr;
          genptr el_p;
          funcp func_p;
          int    param;

public void set_compcond_cbk(g_addr, el_p, func_p,
                  param)
          genptr g_addr;
          genptr el_p;
          funcp func_p;
          int     param;

   public void
   set_compint_cbk(g_addr, el_p, func_p, param)
          genptr g_addr;
          genptr el_p;
          funcp func_p;
          int    param;


   public void
   set_compreal_cbk(g_addr, el_p, func_p, param)
          genptr g_addr;
          genptr el_p;
          funcp func_p;
          int    param;
```

```
public void
set_compba_cbk(g_addr, el_addr, len, func_p, param)
       genptr g_addr;
       genptr el_addr;
       int    len;
       funcp func_p;
       int param;


public void set_compbit_cbk(g_addr, el_p, func_p,
               param)
       genptr g_addr;
       genptr el_p;
       funcp func_p;
       int    param;


public void set_compstr_cbk(g_addr, el_p, func_p,
               param)
       genptr g_addr;
       genptr el_p;
       funcp func_p;
       int param;
```

Callback Example

The following example illustrates the Rational Statemate **callback** and **set** commands.

Initially the system time and date are sensed from the operating system and then passed to the Rational Statemate model by using the **seti** command.

The use of **condition callbacks** monitor the SOUND_FLAG conditions inside the Rational Statemate model and then call their respective function routine to pass a desired sound file to the sound device driver.

```
#ifndef m_generic


#include “types.h”
#include “all.h”
#include <time.h>
#include <stdio.h>


void ext_code_task_a(c_val,c_num)
int c_val;
int c_num;


{
```

```
if (SOUND_FLAG_A)
        {
        system("cat sound_a.au > /dev/audio &");
        }
}

void ext_code_task_b(c_val,c_num)
int c_val;
int c_num;


{
if (SOUND_FLAG_B)
        {
        system("cat sound_b.au > /dev/audio &");
        }
}

void ext_code_task_c(c_val,c_num)
int c_val;
int c_num;


{
if (SOUND_FLAG_C)
        {
        system("cat sound_c.au > /dev/audio &");
        }
}

void ext_code_task_d(c_val,c_num)
int c_val;
int c_num;


{
```

```
if (SOUND_FLAG_D)
         {
         system("cat sound_d.au > /dev/audio &");
         }
}

void ext_code_task_e(c_val,c_num)
int c_val;
int c_num;



{
if (SOUND_FLAG_E)
         {
         system("cat sound_e.au > /dev/audio &");
         }
}



void user_init()


{


  struct tm locTime, UTCtime;
   time_t ltime;


   time( &ltime );      /* get system time */


   locTime = *localtime( &ltime );/* convert to struct
   tm in local time */
   UTCtime = *gmtime( &ltime );/* convert to struct
   tm in UTC/GMT */



/* pass time and date back to statemate model */


seti(&BUR_TIME_MINS,locTime.tm_min);
```

```
seti(&BUR_TIME_HRS,locTime.tm_hour);
seti(&BUR_TIME_MTH,locTime.tm_mon);
seti(&BUR_TIME_YR,locTime.tm_year);
seti(&BUR_TIME_DATE,locTime.tm_mday);

/* monitor sound flag conditions in model */
set_cond_cbk(0,&SOUND_FLAG_A,ext_code_task_a,1);
set_cond_cbk(0,&SOUND_FLAG_B,ext_code_task_b,1);
set_cond_cbk(0,&SOUND_FLAG_C,ext_code_task_c,1);
set_cond_cbk(0,&SOUND_FLAG_D,ext_code_task_d,1);
set_cond_cbk(0,&SOUND_FLAG_E,ext_code_task_e,1);

}
void user_quit()
{
}

#endif

#ifdef m_g_enter_pin

static void g_enter_pin_user_init()
{
}
#endif
```

Supplementing Generated Ada

Implementing Primitive Activities

The file `user_activities.a` contains templates for every primitive activity that is to be implemented in the compilation profile's activity-stub options. Each selected primitive activity contains a stub with the following format:

```
procedure user_code_for<activity_name> IS
--      Parameter :

--      Input <input data-elements>;
--      Output <output data-elements>;
--      Input/Output <Inout data-elements>;
--
        begin
        null ;
        end;
```

The parameters list describes the interface of the activity to the rest of the model. Note that these are not parameters in the programming-language sense. The parameters list is actually a reference list that shows the context of the activity to be implemented in the model.

```
procedure user_code_for FFT IS
--      Input double sonar_datal;
--      Input double sonar_date2;
--      Output double processed_data;
--
begin
        null ;
end ;
```

The previous example shows a primitive activity that represents an FFT filter. The data-items `sonar_data1`, `sonar_data2` are flowing into FFT, and the `processed_data` is flowing outside. This is actually the interface of the FFT activity with the rest of the model. Mathematical processing functions such as an FFT filter, are typical cases where something is implemented as a primitive activity, and the algorithm could be taken from an existing library.

Once the `user_activities.a` file is generated, it is not overwritten when the code is regenerated. In subsequent generations of the code, a `user_activities.a_temp` file is generated. If new templates are generated, they should be merged from `user_activities.a_temp` into `user_activities.a.`

Note

Empty stubs stop right after activation and the **sp** (activity) event is generated in the next step.

User Init and User Quit

The file `user_activities.a` contains `USER_INIT` procedure templates for the main module and for every generic module.
For example:

```
separate (g_ACST_G)
procedure g_acst_g_USER_INIT is
begin
   null;
end g_acst_g_USER_INIT;
with ac_mongo; use ac_mongo;
package body user_activities is

   procedure USER_INIT is
   begin
      null;
   end USER_INIT;

   procedure USER_QUIT is
   begin
   null;
   end USER_QUIT;

   end user_activities
```

You can use the generic USER_INIT function to initialize local variables into generic module.

The code calls this procedure before the very first step is taken in the translated statecharts. Therefore, you can use it for many types of initializations.

For example, you can add an actual piece of code to initialize various global structures in the code supplied for primitive activities, to open windows, etc.

Another important option is to initialize specification elements. Recall that all events, conditions and data-items used in the specification have the following default values:

- events - *not active*
- conditions - *false*
- integer, real and bit-array data-items - *zero*
- textual data-items - *blank string*

The default value is used when there is no explicit initialization of an element before it is referenced in an expression. However, you might wish to intentionally leave an element uninitialized in the specification because you do not know the precise initial value. In such a case, you want to be able to run the same prototype code with different initial values of the element and to choose an appropriate one in a "trial and error" process. Once you choose an initial value, you can add it to the specification. In other words, you tune the system specification by working with the prototype derived from it.

For example, if you want to assign an initial value of true to the condition *FAULT*, and a value of 50 to the integer data-item *LOW_BOUND* which both belong to the chart *EWS*, you transform the template into the following `USER_INIT` procedure:

```
procedure USER_INIT is
begin
       setc (FAULT' address, true);
       seti (LOW_BOUND' address,50)
end ;
```

Execution of the code may come to a point where all activities of the prototyped system become non-active and thus the system must finish its work. This may be caused by various reasons: self-termination of activities, explicit or implied actions stop or the command **QUIT** entered when running the prototype Debugger.

In all cases where the system stops, the code performs a call to the procedure `user_quit`, intended to support a graceful termination of the user extensions. The template of this procedure `USER_QUIT` resides in the `user_activity.a` file:

```
procedure USER_QUIT is
begin
       null ;
end USER_QUIT ;
```

Consider an example in which the prototype code is connected to a graphical mock-up of the operator display. Suppose that among the user's extensions there is a task responsible for I/O interface between the code and the display. When a soft button is "pushed" on the display, the task accepts an interrupt from the mouse and translates it into generation of an event sent to the prototype code. When the system stops, this task must terminate. To achieve this goal, place an abort statement for this task in the template `USER_QUIT`.

Synchronization of Primitive Activities

This portion 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. Therefore, the Code Generator provides two types of user activities:

- Simple procedures
- Tasks

Procedures

A procedure-activity is executed in a *one-shot* - it is not preempted until it returns. Therefore, you should use this mechanism for instantaneous activities (activities that execute for a short period of time). Typically, these activities perform short calculations or non-blocking I/O operations, like displaying data or drawing graphics. If the procedure mechanism is used for continuous calculations or delayed I/O, it blocks the rest of the prototype from reacting properly to incoming events. Since a procedure-activity is not being preempted, the suspend, stop and resume actions do not have any effect on them. When a procedure-activity returns, the sp!( ) event is sent to the controlling code.

Tasks

The task mechanism allows you to integrate continuous or synchronized code into the primitive activity. You do not have to define the Ada task yourself, it is defined by the Code Generator. You specify the body of the task within the procedure template, which is generated in the `user_activities` file.

A task activity is an Ada task that executes concurrently with the rest of the system. It can delay itself, wait for events, and perform continuous calculations. When the task is executed, however, the rest of the code is suspended. To remedy this situation, synchronization points may be defined to allow the rescheduling of other tasks or the control code (main task) to proceed and the actions stop and suspend to take effect. If your procedure returns, the `stop_activity` event is generated.

Synchronization

There are three types of synchronization calls:

- `synchronize (activity_address);`

  The activity address is the address of the activity status variable defined in the package where the activity is referenced.
- `wait_for_event(event)`
- Ada **DELAY** statement

Each one of these calls suspends the activity and reschedule another activity or the `main_task` (statechart) on a round-robin policy.

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.

For example:

```
procedure sense_start IS
    while true loop
        wait_for_event (start'address);
        -- check the status here
        put_line("start generated") ;
    end loop ;
end ;
```

The Ada **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.

For example:

```
procedure poll_input IS
begin
    while true loop
        mouse_input := get_input_from_mouse ;
        if mouse_input /= null  . . . . .
        DELAY 0.1 ;
    end loop ;
end ;
```

Note

The **DELAY** statement allows other activities to run, but stop and suspend do not take effect. If you wish to stop or suspend the activity by other activities, add a synchronize call after the `DELAY` statement.

The `activity_synchronize` is used when you have a lengthy calculation which is too long to be executed without interruption. 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 `activity_synchronize` call allows other activities to proceed, and the calling activity resumes execution in the next available time slot unless a stop or suspend command is 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.

```
procedure multiply IS
begin
   for i in 1 . . 10000 loop
   for s in 1 . . 10000 loop
   -- internal loop
   -- the internal loop is short enough to -- complete.
   end loop
   activity_synchronize(
      acy_MULTIPLY'address);
   end loop ;
end ;
```

Note

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

Tasks in Ada Code Belong to One of the Following Groups:

1. Basic activities that you asked (in the profile) to implement as tasks generated by the tool
2. Tasks that you manually add in the external code (in `user_activities.a`)
3. Tasks in run-time libraries:
   - Intrinsics library:
     SEMAPHORE, COLLECTOR, DELAYS_TASK
   - Debugger library: TRACE_TASK
   - Pge_Interface package: task responsible for accepting inputs from panels:
     PANEL_DISPATCH

Creation and Start

All these are usual Ada tasks supported by the Ada language. In particular, the very declaration of an Ada task does both creation and activation.

Aborting Tasks

Perform abort by using the Ada `abort` statement. Abort is done as follows:

1. User aborts his user-written tasks: where needed in the code he adds abort statements.
2. In addition, if code is generated with debug facilities, then to enable a proper termination, command QUIT causes calls to procedures:
   - FINISHING (found in file main_dbg__.a) aborts all tasks in Run Time Libraries.
   - USER_QUIT (found in file user_activities.a) aborts all extra tasks.

Note

The PANEL_DISPATCH, and basic activities-tasks abort statements are generated automatically. Abort statements for user written tasks should be added manually. The only way to stop code generated without the debug facilities is by **Ctrl-C**. In this case, all involved tasks also become terminated.

Interfacing With the Rational Statemate Model

The Code Generator produces procedures that can access model elements, which are abstract data types.

There are three ways to interface with the Rational Statemate model:

- Procedures to modify values of events, conditions, and data-items. You have to call them in your code whenever you wish to perform the manipulations of the elements. These procedures are discussed in the following subsections.
- Set callback functions to respond to changes in the system. The code guarantees that such a callback is called whenever the corresponding change occurs. This can be, for example, displaying a message on the screen or assignment of an appropriate value to a variable used in the user code.
- Ada does not support pointers to functions, so callbacks are supported using a switch/case command with entries to each of the required elements.

Referencing Model Elements

Communication between the user-defined 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 Ada target language:

- Events and conditions are represented as bytes
- Data-items are represented as integers, reals, strings or unsigned
- User-defined types are derived from basic data-types

The following table shows the mapping between the Rational Statemate basic types and the corresponding Ada types:

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

Where Elements are Defined

An element can be local or global to a module. The element is globally defined if it is referenced by more than one module, i.e., defined in the top-level module. Each module "exports' all its local elements as externals in its package specification file. This allows other user modules to access them. If you want to reference an element you must refer to its scope by WITHing the appropriate package. An example is shown below.

Example:

If you want to reference an element BAUD_RATE defined in module display, you should `WITH` the package display to make the element visible.

```
-- my module
WITH display; USE display;
package body my_package IS
.
br :=diBAUD_RATE ;
.
.
```

Element Names in the Output Code

The element name in the object code is the same as in the Rational Statemate model. If a user-defined element name is not unique, or if it conflicts with a reserved word in the target language, it is changed in the code to contain a prefix that denotes its type and scope. Since the Rational Statemate scopes are different from the modules in the output code, the names are not identical. This avoids any ambiguities that might result from name duplications. The naming convention is shown below:

```
prefix<STATEMATE_NAME>
```

Where prefix is determined as follows:

1. The type:

   ev - event

   co - condition

   di - data-item

   For activities the notation is:

   ```
   acy_<ACTIVITY_NAME>.
   ```

2. To resolve ambiguities:

   If two elements have the same name in a module, a number is added to the prefix to resolve the ambiguity. If an ambiguity occurs, refer to the cross-reference table in the info-file to determine which is the correct element.

Example of two data-items with the same Rational Statemate name (Z):

```
di1Z, di2Z
```

In this case, you should look in the cross reference table to identify which one belongs to which Rational Statemate scope.

Accessing an Element Value

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

Example:

```
if(my_data = X + Y)then
                ...
```

Generating Events

Events are primitive elements and are special in the sense that software languages do not support them directly. An event is active, or “high’, for only one step unless it is regenerated. The intrinsics library supports this behavior via the “gen’ function. Once an event is generated via “gen’, the intrinsics runtime module sets and resets the event at the right time. An active event signifies a value of “1’ in the byte that represents that event.

```
gen (event:address);
```

Example:

```
gen(E1’address);
```

Note that the function expects an address of an event element. Direct setting of an event, i.e., `e1 := true` causes the code to behave incorrectly since the intrinsics module does not handle this situation.

Assigning Values to Rational Statemate Elements

Since model elements follow Rational Statemate semantics, their assignments should be synchronized to the beginning of the next step (cycle). A direct assignment such as

```
X := Y + 1 ;
```

might result in racing condition especially when the data/condition element is shared between two concurrent activities. The synchronized assignments are implemented via a set of service calls supported by the intrinsics library. The following is the synchronized assignment call for the above assignment.

```
seti(X’address, Y+1) ;
```

There are cases where using direct or deferred assignments do not make a difference, however, you should avoid using direct assignments.

Arrays

The intrinsics library offers a set of procedures that apply deferred assignments to the different types of Rational Statemate data-items. The assignment interface calls for each Rational Statemate type are listed in the following table:

Data Item Type	Procedures
Bit:	procedure seti(i: address; val: integer32);
Condition:	procedure setc(cond: address; new_value : boolean);
Real data-item:	procedure setf(r: address; val:float);
Integer Data-item:	procedure seti(i: address; val: integer32);
String Data-item:	procedure sets (s: address; val: string);
Bit-arrays:	procedure SETBA (BA_TRG : ADDRESS; TRG_L,TRG_FROM, TRG_TO ; INTERGER; BA_SRC : ADDRESS; SRC_L,SRC_FROM, SRC_TO : INTEGER):

The following APIs can be used to set arrays or slices of arrays. They all take source and destination arrays, and length. In case of slice assignment such as a1(3..5):=a2(1..3), the following call does:

```
set_array_<type>(a1(3..5)'address,a2(1..3)'address,3);
```

Arrays of Bit-arrays:

For assignments between arrays and bit-arrays

```
procedure SET_ARRAY_BA(TRG_ADR : ADDRESS;
     LEN_TRG, TRG_BITS: INTEGER32;
     SRC_VAL_ADR :ADDRESS;
     LEN_SRC, SRC_BITS : INTEGER32)
```

Array of Events:

```
procedure GEN_ARRAY(E : ADDRESS;LEN:INTEGER);
```

Array of Queues:

```
procedure SET_ARRAY_QUEUE(C:ADDRESS; VAL : QUEUE_ARR);
```

Array of Reals (assigning integer values):

```
procedure SET_I2R_ARRAY (C : ADDRESS; VAL : INT_ARR);
```

Array of Integers (assigning real values):

```
procedure SET_R2I_ARRAY (C : ADDRESS; VAL : REAL_ARR);
```

Array of Integers:

```
procedure SET_I_ARRAY (C : ADDRESS; VAL : INT_ARR);
```

Array of Reals:

```
procedure SET_R_ARRAY (C : ADDRESS; VAL : REAL_ARR);
```

Array of Conditions:

```
procedure SET_C_ARRAY (C : ADDRESS; VAL : COND_ARR)
```

Array of Strings:

```
procedure SET_S_ARRAY(C : ADDRESS; VAL_ADR: ADDRESS ; NUM_ELEM,
STR_LEN_SRC, STR_LEN_TRG : INTERGER);
```

Example:

The following is a supplemented basic activity that processes X,Y and generates two events according to the result : `PROCESS_OK` and `PROCESS_ERROR`.

```
procedure user_code_for_filter IS
-- Parameters:
-- Input int X;
-- Input int Y;
-- Output event PROCESS_OK;
-- Output event PROCESS_ERROR;
BEGIN
   NULL;
END;
```

The supplemented procedure is shown below:

```
procedure user_code_for_filter IS --Parameters; --Input int X; --
Input int Y;
-- Output event PROCESS OK;
BEGIN
   apply_filter(X,Y,Z);
   -- function,operates on X,Y
   If in_range(Z)THEN
      gen(PROCESS_OK'address);
   ELSE
      gen(PROCESS_ERROR'address);
   END IF;
END;
```

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.

The examples below are of Rational Statemate elements and the resulting definitions found in the generated code and in the Rational Statemate Ada libraries.

Rational Statemate Elements:

```
BA1 is Array 1 to 10 of Bit-array 31 downto 0
BA2 is Array 1 to 10 of Bit-array 48 downto 0
BA3 is Array 1 to 10 of Bit-array 3 down to 0
```

In the Generated Code:

```
BA1 :  u_ba32_arr(1..10) ;
BA2 :  u_ba49_arr(1..10) ;
BA3 :  u_ba4_arr(1..10) ;
```

```
type u_ba32_arr is array(natural rand <>) of u_ba32;
subtype u_ba32 is BITS_ARRAYS (1..1);
type u_ba49_arr is array (natural range <>) of u_ba49;
subtype u_ba49 is BIT_ARRAY (1..2);
type u_ba4 is array(natural range<>) of u_ba4;
subtype u_ba4 is BITS_ARRAY (1..1);
```

`BITS_ARRAY` is defined in the intrinsics package as follows:

```
type BITS_ARRAY is array(NATURAL range<>) of
INTERGER32;
```

Bit Array Functions

```
function AND_B(BA1 : ADDRESS;
               L_BA1, FROM1, TO1 : INTEGER32;

               BA2 : ADDRESS;
               L_BA2, FROM2, TO2 : INTEGER32)
               return ADDRESS;


function NAND(BA1 : ADDRESS;
                   L_BA1, FROM1, TO1 : INTEGER32;
               BA2 : ADDRESS;
                   L_BA2, FROM2, TO2 : INTEGER32)
                       return ADDRESS;


function NOT_B(BA : ADDRESS;
               L_BA, FROM, TO : INTEGER32)
                return ADDRESS;


function OR_B(BA1 : ADDRESS;
               L_BA1, FROM1, TO1 : INTEGER32;
                   BA2 : ADDRESS;
                   L_BA2, FROM2, TO2 : INTEGER32)
                       return ADDRESS;


function XOR_B(BA1 : ADDRESS;
               L_BA1, FROM1, TO1 : INTEGER32;
                   BA2 : ADDRESS;
                   LBA2, FROM2, TO2 : INTEGER32)
                      return ADDRESS;


function NOR(BA1 : ADDRESS;
               L_BA1, FROM1, TO1 : INTEGER32;
                   BA2 : ADDRESS;
                   L_BA2, FROM2, TO2 : INTEGER32)
                      return ADDRESS;


function NXOR(BA1 : ADDRESS;
                   L_BA1, FROM1, TO1 : INTEGER32;
                   BA2 : ADDRESS;
                   L_BA2, FROM2, TO2 : INTEGER32)
                       return ADDRESS;


function LSHL(BA : ADDRESS;
                   LEN_BA,FROM,TO,SHIFT : INTEGER32)
                       return ADDRESS;
```

```
function LSHR(BA : ADDRESS;
                LEN_BA,FROM,TO,SHIFT : INTEGER32)
                     return ADDRESS;


function ASHL(BA : ADDRESS;
              LEN_BA,FROM,TO,SHIFT : INTEGER32)
              return ADDRESS;


function ASHR(BA : ADDRESS;
              LEN_BA,FROM, TO, SHIFT : INTEGER32)
               return ADDRESS;


function CONCAT_BA(BA1 : ADDRESS;
              LEN_BA1, FROM1, TO1 : INTEGER32;
              BA2 : ADDRESS;
              LEN_BA2, FROM2, TO2 : INTEGER32)
              return ADDRESS;


function SIGNED_B(BA : ADDRESS;
              LEN_BA, FROM, TO : INTEGER32)
              return INTEGER;


function BITS_OF(I_VAL : INTEGER32;
              FROM, TO : INTEGER32)
                       return ADDRESS;


function MUX(BA1 : ADDRESS;
                  L_BA1, FROM1, TO1 : INTEGER32;
                  BA2 : ADDRESS;
                  L_BA2,FROM2,TO2,SEL : INTEGER32)

              return ADDRESS;


function EQBA(BA1: ADDRESS;
              AR1_LENGTH, FROM1, TO1 : INTEGER32;
                  BA2 : ADDRESS;
                  AR2_LENGTH, FROM2, TO2 : INTEGER32)
              return BOOLEAN;


function EQ_ARRAY_BA(BA1 : ADDRESS;
              AR_LENGTH : INTEGER32;
              L_BITS1 : INTEGER32;
              BA2 : ADDRESS;
              L_BITS2 : INTEGER32) return BOOLEAN;


function I2BA(VALUE : INTEGER32)
              return BITS_ARRAY;
```

```
function I2BA(VALUE : INTEGER32) return ADDRESS;


function BA2INT(BA : ADDRESS;
             LEN_BA, FROM, TO : INTEGER32)
             return INTEGER32;


function EXPAND_BIT(BIT_VAL : INTEGER32;
               LEN_BA : INTEGER32) return ADDRESS;


function MINUS(BA : ADDRESS;
               LEN_BA, FROM, TO : INTEGER32)
             return ADDRESS;


Procedure ASSIGN_BA(TRG_BA : ADDRESS;
             TRG_LEN_BA: INTEGER32;
             BA : ADDRESS;
             LEN_BA, FROM, TO : INTEGER32) ;


function GET_SLICE(BA : ADDRESS;
             LEN_BA, FROM, TO : INTEGER32)
             return ADDRESS;
```

Structured Elements

For complex *Data-Items* in the Rational Statemate model (e.g. , a *Data-Item* record) a type is defined for the *Data-Item*. This happens even if the *Data-Item* is not defined as a user type. The type declaration is placed in the same package specification as the declaration for the *Data-Item*. These implicitly defined types are treated the same way as ordinary User-Defined Type*s*.

Rational Statemate defines structured elements and user-defined types in a file called `<profile name>_type_utils.a`, and assigns names to the types based on the name of the *Data-Item* and the characters *ty* as a suffix. For example,

- RECORDty RECA;
- UNIONty UNIONA;
- USER-DEFINED USER[1];

Records and Unions

Rational Statemate records and unions are translated into Ada records. For example, a record INVOICE_TYPE is translated into:

```
type INVOICE_TYPE is record
  NAME : string(1..81);
  ITEM : string(1..81);
  AMOUNT : float64;
end record;
```

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 Ada code names is modified to make all the names unique. This name mapping information is listed in the *.info* file.

Enumerated Types

An Enumerated Type is a user-defined type with a finite number of values.

You cannot directly define a data item as an enumerated type. First, define the data item as a user-defined type, and then define the user-defined type as an enumerated type. You define the values for the enumeration in the “Definition’ field of the Properties by listing the values in brackets separated by commas. For example, `{SUN,MON,TUE,WED}`

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.

Note

Ada provides a way to define a subtype of an enumerated type. This subtype element usually can hold a subrange of enumeration values of the enumerated type. These types and subtypes can be related and used together in expressions. Run-time errors are issued when “out of range’ values are assigned to a subtype.

The definition of a subtype is only allowed for user-defined types, not for data-items.

There are two constant operators and five general operators for enumerated types:

Constant Operators	Description
T’FIRST	First enumerated value of T
T’LAST	Last enumerated value of T

Parameters to these constant operators are user-defined types that were defined as enumerated types.

General Operators	Description
T’SUCC	Successor enumerated value of T
T’PRED	Predecessor enumerated value of T
T’ORD	Ordinal position of VAL in T
T’VAL	Value of the i’th element in T
T’IMAGE	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.

User-Defined Type Functions

There are a number of functions provided for manipulating Rational Statemate model variables that should be used when augmenting the Rational Statemate generated code.

Note

Rational Statemate variable values may be read by checking the correct variable name. Value changes, however, should not be made directly to the same variable. All value changes are made through a list of variables to be updated. This list is affected through a variety of functions created by the code generator.

Use the call `seti(variable_name'address, value);` to set any (primitive) integer variable to a desired value. Other similar calls provide the ability to set conditions, strings, real numbers, etc. In addition to these general functions, the Code Generator creates similar functions that are specific to each User-Defined Type (UDT).

Every UDT has the following functions defined for it:

```
procedure SET_<type> (A : address ; B : address);
```

It uses the update list to assign A:=B. The user-code should not make direct assignments to Rational Statemate variables. Use the following set functions to test for equality:

```
function EQ_<type> (A : address ; B : address):
```

Returns TRUE if the elements A and B are equal.

For every type that has a corresponding *_event* declaration, the following functions are defined:

```
function ALL_<type>
(A : <type>_event) return boolean;
```

These functions test to see if all the events that form A are currently generated. This example only applies to `RD<element>` and `WR<element>`.

```
procedure GEN_<type>
(A : <type> (A : <type>_event);
```

These functions generate all the events in A. This only applies to `RD<element>` and `WR<element>`.

These functions create and initialize complex data-types.

```
procedure INIT_<type> (A : address);
```

Queue Functions

Queues are implemented as linked lists in the generated code. Each node in the linked list contains a pointer to an element. Access the lists using the access functions described in the following table:

Function	Description
SETQ	To return the length of the queue Q, use the following function.
	`procedure SETQ(Q : ADDRESS; ELEMENT : ADDRESS; Q_EL_SIZE : INTEGER);`
Q_GET	To remove an element from the head of the queue and put it into an element, use the following function. (The status is TRUE if there was an element to get. If the status is passed as NULL, no status value is returned.)
	`procedure Q_GET(Q : ADDRESS; ELEMENT :ADDRESS; STATUS : ADDRESS);`
Q_PEEK	To copy an element from the Q, use the following function. This is the same as Q_GET, except the TOP queue element is not removed from the queue it is only copied.
	`procedure Q_PEEK(Q: ADDRESS; ELEMENT : ADDRESS; STATUS : ADDRESS);`
Q_INITIALIZE	To create your own queues, use the following function to initialize the queue before use. The queue starts empty.
	`procedure Q_INITIALIZE(Q:ADDRESS ;  Q_EL_SIZE :INTEGER);`
Q_PUT	To put an element at the end of Q, use the following function.
	`procedure Q_PUT(Q:ADDRESS; ELEMENT : ADDRESS; Q_EL_SIZE : INTEGER):`
Q_UPUT	To put an element at the head of Q, use the following function.
	`procedure Q_UPUT(Q:ADDRESS; ELELMENT : ADDRESS; Q_EL_SIZE : INTEGER);`
Q_FLUSH	To delete all the elements in Q, use the following function.
	`procedure Q_FLUSH(Q : ADDRESS);`

All queue manipulation is done using memory allocation for elements added to the queue and freeing the memory when the elements are removed or deleted from the queue. The memory allocation and freeing is done automatically so there is no need to preserve the value of element after the `Q_PUT()` or `Q_UPUT()` procedure is called.

The queue manipulation functions work through the update list, so if some elements are put into a queue, they are not in the queue until the end of the step.

String Functions

```
function CHAR_TO_ASCII(STR: string) return integer;


function ASCII_TO_CHAR(INT_VAL:integer) return string;


function STRING_EXTRACT( STR:string ;
INDEX : integer;  LENGTH : integer) return string;


function STRING_INDEX ( STR:string ;
    OFFSET: integer;SUBSTR : string) return integer;


function INT_TO_STRING(INT_VAL:integer) return string;


function STRING_TO_INT(STR : string) return integer;


function STRING_LENGTH(STR :string ) return integer;


function STRING_CONCAT(STR_DEST : string;
    STR_SRC : string) return string ;
```

READ, WRITTEN, CHANGED, TRUE, and FALSE in Complex Data Types

Sensing how the *rd() , wr()* , and *ch()* events, as well as *tr()* and *fs(),* are related to complex *Data-Types* requires the Code Generator to create some additional *Data-Types* and variables. These type declarations are put in the same header file as the type declaration for the complex type itself. Remember that the names and number of header files varies, based on whether the *Separate File per Statechart* option was selected in the profile under **Options > Module Settings**.

Additional types are required for *wr()* and *rd()* as well as additional variable declarations. These additional types have similar structures to the complex types themselves. The type have fields with the same name but the types of the fields are EVENTS. These are used to store the event of WRITTEN or READ for each of the fields of the complex type.

A separate variable is defined for *rd()* and *wr()*. Each is defined as a type `COMPLEX_VARIABLE_NAME_event`. The name of the variables is the same as the complex type itself, with either *wr* or *rd* prefixed, as appropriate. One or more fields in the *wr* or *rd* variable is set when these fields are READ or WRITTEN. This is found in the procedure for the action that sets or uses the referenced field values.

If one or more of statements *ch() , tr() ,* or *fs()* are used in the model, an additional variable is needed that saves the previous value of the referenced model variable. This variable has the same type as the complex variable, and the same name as the complex variable with the letters *prev_* prefixed.

This is the same scheme that is used for primitive variables in the code generator. It has been extended to work with complex variables as well.

Note

The prev_ variable is updated every step and using `wr` and `rd` has an impact on the performance of the generated code. You should use it discriminatively. Sometimes, it may be more efficient to create your own `X_CHANGED` flag that is set only when needed.

Detecting Changes in Value

If the event expression *ch(), tr()* or *fs()* is applied to an expression that uses a *Data-Item* or condition, it is necessary to preserve the previous value of the element. This is done by maintaining a duplicate copy of the element that is updated at the end of every step. (This makes detecting *ch()* on a large array or complex record a very slow process, because it takes time on *every* step.) The duplicate copy of the variable is named prev_<element>.

The following table lists the affected element types:

Element Type	Variable Name	Purpose
User-Defined Type	<type>	define base *Data-Item*
User-Defined Type	<type>_event	define rd() and wr()
Data-Item	<element>	holds value of *Data-Item*
Data-Item	WR<element>	holds wr() event for *Data-Item*
Data-Item	RD<element>	holds rd() event for *Data-Item*
Data-Item	prev_<element>	holds old value of the *Data-Item*

Note

- **Element**—is the name of the element as listed in the .info file. This is normally the same as the Element Name in the Properties, unless there are duplicate Element Names.
- **Type**—is the name of the User-Defined Type. It is also listed in the info file.

Implementing a Function to Get External Inputs

You should create a separate task, using the tasking functions described in “Synchronization of Primitive Activities” of this section. The task can be initiated in the `user_init()` function in the `user_activities.a` module.

Use the input task 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.

Index