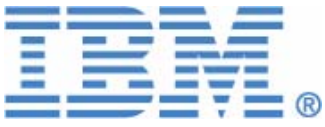
Rational. Statemate

MicroC Programming Style Guide

Rational Statemate
MicroC Programming Style Guide



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

MicroC Overview

MicroC is a graphical software design and implementation tool that supports the development of embedded real-time software for micro-controllers. The focus of the tool is to support the process of developing software pieces while targeting small micro-controllers. The support to design-level debugging, testing – both interactively and in batch mode and analysis of runs is implemented through various instrumentation of the generated code. The output of the tool is a compact, readable ANSI C code, with support to local extensions of the standard C, as well as automatically generated design documentation. MicroC uses an *Operating System Implementation (OSI)* definition to describe the implementation of the software and hardware target environment for a given design. Any one OSI might support only a subset of the design concepts referred to above. As a general rule, the tool tries to make use of any such design aspect/concept it encounters in the model. If the given OSI has no support for that design aspect/concept, an error message is produced.

Code is generated directly within MicroC based on a graphical model that represents the full functionality of the application being designed. There are four basic graphical tools used to define the application. These include:

- Statecharts
- Activity Charts
- Flowcharts
- Truth Tables

Each graphical tool has an associated graphical design language that allows the designer to be very precise in defining the functional role of each graphical element. The graphical elements can be supplemented by linking in user supplied C and/or Assembly Language code.

All of the graphical elements are stored in an internal database that contains associated data about each element. The Data Dictionary tool is used to define and manage the various data elements as well as various other properties of both the textual and the graphical entities in the model.

Properties can be applied to data or to tasks. Data properties are typically defined as Exact Type, although integer types can be BYTE-defined for as appropriate for specific system architectures. Task properties are defined with a Task Priority in the model.

MicroC also includes a Check Model utility that serves as a model checker (somewhat like a precompiler) to detect and warn of incomplete definitions as well as common design pitfalls to help reduce development time and increase the quality of the generated code.

Scope of this Guide

Before the inherent aspects/concepts supported by the tool are described, it is important that you understand the scope and limitations of this material. The recommendations given throughout this document are intended to serve as design guidelines for advanced programmers concerned with details of OSI definition and use. By no mean do they guarantee the safety, correctness or any other property of the application developed using MicroC. This is the sole responsibility of the designer.

The functional details presented here do not imply, by any means, any limitation on the developed features of the Languages Supported by MicroC application code. Because C and Assembly language functions are a part of a Micro model, as well as any existing (i.e. legacy) libraries and sources, everything that can be done in those languages can be done within the MicroC model.

Languages Supported by MicroC

The languages used in MicroC can be both graphical and/or non-graphical (i.e. textual).

Graphical Languages

Structuring Language: Activity chart

The software structure is defined in the top-level Activity chart. In this graphical view of the application, the architecture of the software being developed is determined. TASKs and Interrupt Service Routines (ISRs) are defined as well as the functional content of them.

Another design level definition is done here. The bindings of signals to physical hardware ports and addresses is done using the flow lines to and from the various TASKs and ISRs in the chart. The generated application architecture is defined, by the user, in this view. TASK and ISR code frames are generated, according to the specific properties of the TASK/ISR. A TASK/ISR code frame invokes the Activities mapped underneath the TASK/ISR.

Decomposition Language: Activity Chart

This is a data-flow oriented graphical language. Functionality, in here referred to as “Activity behavior” is defined using the, well known, decomposition method. Each required functionality, i.e., “Activity” is sub-divided into functions, i.e., “Sub-Activities” that might be further divided into even smaller “Sub-Activities,” until no further decomposition is needed.

When no further decomposition is needed, the “Basic Activities” are defined – those that implement certain functionality. The implementation might be defined using the various languages described below.

The code generated for an Activity is a function (or a C Preprocessor macro). For a non-basic Activity, the function calls each of the Activity’s sub-Activity functions. For a basic Activity, the function contains the implementation code.

Activity Behavior: Graphical Implementation Languages

Statecharts

Statecharts are hierarchical state transition diagrams. That language is best in describing application modes and transitions between the modes, as well as application reaction to various events in each of the modes. This discrete behavioral language is very much powerful in describing such application modes and transitions between those modes. When other calculations needs to be defined, that are not mode-based, other languages, those that are described below, should be considered.

The implementation of Statecharts in MicroC is compact and efficient. The application uses a State Variable per each of the Activities implemented by a Statechart. States are encoded to reuse RAM bits. Several synthesis algorithms are used to reduce both the RAM and the ROM required to implement a Statechart on a base of “Pay for what is used”. The user should be aware that as the application maintains the State Variable, certain code (i.e., RAM and ROM) is required. Therefore, it is recommended to use that language whenever that information, i.e., the application state, is required.

Flowcharts

Here we refer to regular Flowcharts. Iterative algorithms, if-then-else constructs, switch statements and direct calculations should normally be defined as a flowchart. That graphical language enables the user to graphically debug the algorithm, and it recommended to be used whenever the calculation is not mode-based and the specific calculation can not enjoy the benefits of the other textual languages, listed below.

The code of a Flowchart runs from beginning to end, without stopping. If the Flowchart is ever run again, it starts from the beginning. The code generator tries to minimize the number of `goto` statements that are needed. This makes the code readable and structured.

Examples that benefit from flowcharts: tuning a radio via incremental frequency adjustment... is it tuned? y/n, stop or increment frequency.

Textual Languages

Textual, Non-Graphical, Implementation Languages are used to define Activity behavior.

Truth Table

The functionality of Activity might be directly defined using Truth Table. Truth Table is a table describing the inputs, the resulting outputs and the actions performed. Truth Tables are recommended to be used when the Activity has many inputs to consider and few states/modes to be in.

When the Truth Table is defined in a reduced form, it will be reflected in the generated code. This enables the user to build highly efficient implementations. For example: Radios are prime examples, once on they respond to button presses, perform an action and return to the on state. Another such function could be a climate control controller, once on and in control mode, button presses are generally responded to and the control state returned to.

Mini-Spec, using the Rational Statemate Action Language

Two modes of Activities may use the Activity's Mini-Spec as implementation:

- **Reactive Activities**

  When the functionality is best defined as pairs of triggers and actions, that language is the most suitable to define that behavior. The syntax is exactly trigger and action: E/A thus directly expresses the required behavior.

  This textual language allows most clear, straight forward and compact implementation when the required functionality might be defined as a set of triggers and resulting actions.

  For example: On/Off Behavior like the following:

  ```
  Button1Press/turn_on(); tr!(Lamp1);;
  Button2Press/fs!(Lamp1); shutdown();;
  ```

- **Procedural Activities**

  When the functionality is a pure calculation, defined as a sequence of "if then else," iterations and numerical calculations that language might be used. It is similar in its expressiveness to the Flowchart graphical language, however it does not requires any graphics, thus might be faster to complete when the algorithm is already proved to be correct.

  This textual language is the Rational Statemate Action Language. .

Time Model and Related Time Operators

MicroC has three model constructs that have a notion of time:

- Timeout and delay operators; referring to **Software** Counter(s)
- Schedule operator; referring to **Timer**
- Periodic Task; referring to **Timer**

Keep in mind that the concept of a Timer might not be supported on all targets. For example, in the `mainloop_sc` OSI supplied with MicroC there is no direct support for a **Timer**.

The tool assumes the existence of a primary single Software Counter**.** It is determined within the compilation profile whether this defaults to SYS_TIMER and possibly **Timer(s)**. Applications might use numerous software counters and timers.

Timeout and delay operators, referring to “Software Counter”(s):

delay(delay_time) (dly)	Related to the Primary Software Counter.
timeout (an_event, delay_time [, counter_id]) (tm)	Related to the Software Counter specified by the `counter_id`

`Delay() expires delay_time` after entering the state connected to the transition with the delay trigger/reaction.

`Timeout()` expires `delay_time` after `an_event` occurs, while the application is staying in the current state waiting for the timeout.

Related to the Software Counter specified by the counter_id

Note

When the 3rd argument is omitted in timeout(), the primary “Software Counter” is used.

`Delay` is actually `tm(en(S), d-time)` where `S` is the state name. *Delays* and *Timeouts* are “Soft” and “Passive.” This means that they are relatively cheap to implement internally, using 1 or less *Timeout Variable Type* variables and 1 or less bits of memory (i.e. RAM).

The actual delay might be greater than or equal to (i.e. >=) the specified delay, depends on the cycle used to schedule the task where the timeout is specified. It is conceivable that this might never occur. The actual implementation of the software counters is defined in the OSI.

The Schedule Operator refers to a hardware timer, **Timer**:

schedule(an_event, sc_time [, sc_cycle]) (sc)

schedule(an_action, sc_time [, sc_cycle]) (sc)

schedule(an_expression, sc_time [, sc_cycle]) (sc)

Note the optional cycle expressed as the 3 rd parameter to `sc!(exp, delay, cycle)`. The timer might be defined in the data dictionary of the scheduled operand (e.g. event or action), or automatically by the tool. Note that this type of timer is potentially more expensive than the delay operator. Actual invocation time accuracy and cost depends upon the **Timer** implementation. Hardware timers are very accurate and *Active*, however they are typically a scarce and expensive resource.

The actual implementation of the timers is defined in the OSI.

Asynchronous Aspects of MicroC

In a MicroC model we identify two basic forms of asynchronicity:

- Interrupt Service Routine (ISR)
- Task

Interrupt Service Routine

MicroC ISR - A Reactive Component that Models Interrupt Service Routine, with associated data and functionality, defined as an Activity sub-type. Might run at any time, regardless of the internal application‘ state. In some environments, when having interrupt levels, an ISR run might be interrupted and preempt by a higher priority interrupt.

TASK

MicroC Task: Reactive Component with associated interface, data and functionality, defined as an Activity sub-type. A “MicroC TASK” might be defined as a TASK in the environment, thus running on its own, like in OSEK, or might be plugged into existing time slice (also called TASK), using the OSI “Link with Scheduler” mechanism. MicroC Tasks run independent of each other. According to the environment, a TASK run might be interrupted and preempt by a higher priority TASK, or an interrupt.

MicroC recognizes two Task running modes [2,3]:

1. **RUN_TO_TERMINATE**: That MicroC Task will run, once entered the function frame, until it has stabilized, and then will leave the function frame (return, terminate, etc.)
2. **RUN_TO_WAIT_EVENT**: That MicroC Task will run, once entered the function frame, and will never leave [2] the function frame. It will be active until it has stabilized, i.e., it finished its calculation, and then it will enter a rescheduling call defined as “Wait for Event”/”Wait for Multiple Events”.

Note: Use the **Data Dictionary->Design Attributes-> Use Active Bit** flag.

Actual implementation details of TASK/ISR is defined in the OSI.

Note

- Some of the OSIs (for example, the mainloop_sc) might support only a subset of those.
- There is a mode, named “Use Active Bit” that allows further control and actually enables even such Tasks to sometimes return/terminate.

Synchronization

Synchronization can be implemented using **Semaphore** and **Signal** (TASK Event).

Synchronization: Semaphore

Used to co-ordinate accesses to shared resources such as memory or hardware by asynchronous entities, modeled as CONDITION sub type. Supported with special operators:

get(SEM1) (gt!)

release(SEM1) (rl!)

The actual implementation of those operators is defined in the OSI.

Note

Once defined as Semaphore, the condition can no longer be used as a regular condition.

Synchronization: Signal (TASK Event)

Used to signal to a TASK on some occurrence like timer expiration, message arrival etc., modeled using EVENT sub typed as TASK Event. Used like regular events:

- As Trigger, to wait on the event
- As Action, to set (generate) the Event

The actual implementation is defined in the OSI.

Serial Communication / Messages

Messages are modeled using DATA ITEM sub typed as message. Supported with special operators:

```
send(MESS_DI1) (sn!)
receive(MESS_DI2) (rc!)
```

The actual implementation of those operators is defined in the OSI.

Timers

Means to schedule TASK invocation, or a Signal (TASK Event) generation. Modeled indirectly:

- Using schedule operator (sc!)
- Using periodic TASK

The actual implementation and capabilities of those operators is defined in the OSI and intended to refer to Hardware Timers.

Activity Behavior: User-Defined Functions

User-defined functions might be implemented in any of the following languages:

- ANSI C
- Assembly Language Code

Use the old safe way to link with legacy code, i.e. call OS/ environment special services and utilize otherwise inaccessible functionality as inline assembly calls. This should be use like a glue, for reuse of legacy code and to implement tricky algorithms.

For example: Debouncing and filtering algorithms; continuous controllers like PI loops within HVAC and Cruise ECUs could also be implemented in this way.

Truth Tables

Very much as described earlier, this language is available for defining user functions, describing defined actions and directly defining Activity content.

Lookup Tables

This language's purpose is to support non-linear functions, such as *Y=F(X)*, so common in the world of micros. Such functions are typically used to represent characteristic curves of valves.

For example: A speed dependent intermittent wiper system will want to use a look-up table to define the time between wipes. Cut-out currents on electric motors can be accurately set using a look-up table to define the typical current at different positions.

Rational Statemate Action Language

This kind of programming language can be used where a function is needed in an application. It is the preferred language of choice, rather then plain C code, as all of the expressions are parsed. Thus, it is possible to define in the Data Dictionary tool relevant properties of the elements used. As such, compatibility between different targets can be achieved easier as the tool will generate the right expressions in each target environment. This can not be done if the function is already defined in C code.

For example: An automotive interior light ramping function that can be triggered from the doors, ignition key and switch.

Exact Case Usage

MicroC supports “exact case” naming of textual elements across the product. For each textual element (including data types, data items, subroutines, events, actions, and conditions), MicroC holds two names:

- Case-sensitive name
- Uppercase name

The case-sensitive name is a regular field in the database. Throughout MicroC (including the Dictionary, static reactions, mini-specs and so on), the exact-case name is used. The Code Generator uses the exact-case name when generating full expressions—when preprocessor macros (for data items, user-defined types, and subroutines) are not used. Preprocessor macros remain uppercase only.

Note

You cannot use different cases of the same name for different variables because they resolve to the same name. For example, both Ab and aB resolve to AB.

The first time you specify an element, MicroC records its exact case, and converts any subsequent references to it to the same convention. For example, if you first enter “aB,” MicroC converts any case combination of it (“AB,” “Ab,” “ab,” or “aB”) to aB. Use the *Rename* option in the dictionary to respecify the name or case of the element.

Note that the check model tool will warn you when two strings (the case-sensitive name and the uppercase name) do not match. This might happen if you change the setting of the Case Sensitive Name attribute. By default, MicroC uses case-sensitive names.

The following aspects of MicroC require exact-case handling:

- Generated code
- Expressions (going through the parser).Overview
- Dictionary-Editor/ Dictionary-Browser selection matrix
- Element information
- UDT Dictionary
- Action definitions

Exact-case usage is not supported in local parameters of subroutines, nor context variables.

Structuring Language: Activity Chart Implementation

A top-level Activity might be defined as a TASK or as an Interrupt Service Routine (ISR). Note that a TASK or ISR will have special meaning in OSEK 2.0 applications.

TASK Activities

In OSEK 2.0 OS – we identify two TASK types:

- Basic Task
- Extended Task

Various other properties might be related to a TASK, some depend on the TASK type and some common to both types, as described below.

Both types of TASKs might be scheduled to be activated at system startup, if desired, and to run periodically, with a user define period. Each TASK body contain calls to the functions it is running as well as some code, according to the TASK specific properties, as described in the examples below.

In general, BASIC TASK is less expensive to use regarding run time RAM usage, as after it complete its run it terminate and the OS free all the RAM associated with them, thus enabling reuse of this memory.

An EXTENDED TASK can never terminate after it has been activated, thus the RAM associated with it will never be freed.

However, it takes more time to activate a BASIC TASK. This is true because, once a request to activate the task has been received, it is required to initialize the RAM associated with it. On the other hand, an EXTENDED TASK is faster to react because, after it has been activated, the associated RAM will be kept and does not need to be initialized again for subsequent use.

As a general rule, use an EXTENDED TASK when the reaction time to some external event needs to be as short as possible, or when using the TASK EVENT inter-task communication mechanism. Otherwise, use a BASIC TASK. Refer to the discussion below, as well as to the OSEK/OS documentation for further details.

BASIC TASK

A BASIC TASK runs once, upon activation, and then terminates.

The code frame for a BASIC TASK (for example: TASK1 containing Activities A11 and A12), without controller, will resemble the following:

```
TASK (TASK1)
{
cgActivity_A11();
cgActivity_A12();
TerminateTask();
}.
```

If the TASK is periodic, with a period of 10 ticks, the code will change to look like this:

```
TASK (TASK1)
{
if ((cgGlobalFlags & ALARM_SET_TASK1) == 0){
cgGlobalFlags |= ALARM_SET_TASK1;
SetRelAlarm(TASK1_ALARM, 10, 10);
};
cgActivity_A11();
cgActivity_A12();
TerminateTask();
}
```

Note

Use the **Data Dictionary->Design Attributes->Schedule Periodic** flag to define a periodic Task.

If the TASK is periodic, containing Activities A11 and A12 with CTRL1 as controller, the code will change to look like this:

```
TASK (TASK1)
{
if ((cgGlobalFlags & ALARM_SET_TASK1) == 0){
cgGlobalFlags |= ALARM_SET_TASK1;
SetAbsAlarm(TASK1_ALARM, 10, 10);
};
do {
cgGlobalFlags &= ~BITSUPERSTEP_TASK3;
cgActivity_A11();
cgActivity_A12();
cgActivity_CTRL1cnt1();
} while ( (cgGlobalFlags & BITSUPERSTEP_TASK1) != 0);
TerminateTask();
}.
```

EXTENDED TASK

An EXTENDED TASK runs once, upon activation, and then suspends itself, calling the "`WaitEvent`" API function. A specific modification to this EXTENDED TASK behavior will be described below, at the end of that section.

The code frame for an EXTENDED TASK (for example: TASK2 containing Activities A21 and A22), without controller, will look like the following:

```
TASK (TASK2)
{
cgSingleBuffer_TASK2.eventMask = 0xff;
start_activity_A21;
start_activity_A22;
while(1) {
cgActivity_A21();
cgActivity_A22();
WaitEvent(cgSingleBuffer_TASK2.eventMask);
ClearEvent(cgSingleBuffer_TASK2.eventMask);
}
/* TerminateTask(); */
       }
```

Note

With regard to lines 3, 9, 10 in the last example: This has been changed from earlier implementations of MicroC. In newer versions of MicroC, the eventMask data variable is no longer allocated. The defined mask, in the example above 0xff is directly inlined in the call to WaitEvent and ClearEvent calls. This note is applicable to the rest of the examples in this document.

Note

Further Optimization: In certain implementations it is possible to call the WaitEvent and ClearEvent API functions with constants, thus avoiding the need for allocating RAM for the eventMask.

What can be seen is that upon invocation, the call to *start_activity_A21* and to *start_activity_A22* is done. The call is done only once, the first time the TASK is run. This will drive the event *started* of those sub-activities as well as the event *started* for the task itself. This is supported only for that task type.

After that call, the code enters an infinite loop running all of the TASK' sub-activities, and entering the suspension mode through call to "WaitEvent".

If somewhere underneath the TASK, not as direct descendant, we will add a Statechart, the code will change to be like:

```
TASK (TASK2)
{
cgSingleBuffer_TASK2.eventMask = 0xff;
start_activity_A21;
start_activity_A22;
while(1) {
do {
cgGlobalFlags &= ~BITSUPERSTEP_TASK2;
cgActivity_A21();
cgActivity_A22();
if(cgDoubleBufferNew_TASK2.cg_Events)
cgGlobalFlags |= BITSUPERSTEP_TASK2;
cgDoubleBufferOld_TASK2 = cgDoubleBufferNew_TASK2;
cgDoubleBufferNew_TASK2.cg_Events = 0;
} while ( (cgGlobalFlags & BITSUPERSTEP_TASK2) != 0);
WaitEvent(cgSingleBuffer_TASK2.eventMask);
GetEvent(TASK2, &cgSingleBuffer_TASK2.eventsBuff);
ClearEvent(cgSingleBuffer_TASK2.eventMask);
}
/* TerminateTask(); */
}.
```

If the TASK is periodic, with a period of 10 ticks, the code will change to look like this:

```
TASK (TASK2)
{
SetRelAlarm(TASK2_ALARM, 1, 10);
cgSingleBuffer_TASK2.eventMask = 0xff;
start_activity_A21;
start_activity_A22;
while(1) {
do {
cgGlobalFlags &= ~BITSUPERSTEP_TASK2;
cgActivity_A21();
cgActivity_A22();
if(cgDoubleBufferNew_TASK2.cg_Events)
cgGlobalFlags |= BITSUPERSTEP_TASK2;
cgDoubleBufferOld_TASK2 = cgDoubleBufferNew_TASK2;
cgDoubleBufferNew_TASK2.cg_Events = 0;
} while ( (cgGlobalFlags & BITSUPERSTEP_TASK2) != 0);
WaitEvent(cgSingleBuffer_TASK2.eventMask);
GetEvent(TASK2, &cgSingleBuffer_TASK2.eventsBuff);
ClearEvent(cgSingleBuffer_TASK2.eventMask);
if(cgSingleBuffer_TASK2.eventsBuff & 0x01)
GEN_IN_CURRENT(TASK2_EV);
}
/* TerminateTask(); */
}
```

Another setting option, available in MicroC for an *EXTENDED TASK* is *Guarded Activation* mode. In this mode the TASK will be active only while its control bit is set. The sample code will change to look like:

```
TASK (TASK2)
{
if ((cgGlobalFlags & ALARM_SET_TASK2) == 0){
cgGlobalFlags |= ALARM_SET_TASK2;
SetRelAlarm(TASK2_ALARM, 1, 10);
};
cgSingleBuffer_TASK2.eventMask = 0xff;
while((cgGlobalFlags& BITAC_TASK2) != 0) {
do {
cgGlobalFlags &= ~BITSUPERSTEP_TASK2;
cgActivity_A21();
cgActivity_A22();
if(cgDoubleBufferNew_TASK2.cg_Events)
cgGlobalFlags |= BITSUPERSTEP_TASK2;
cgDoubleBufferOld_TASK2 = cgDoubleBufferNew_TASK2;
cgDoubleBufferNew_TASK2.cg_Events = 0;
} while ( (cgGlobalFlags & BITSUPERSTEP_TASK2) != 0
&& (cgGlobalFlags& BITAC_TASK2));
WaitEvent(cgSingleBuffer_TASK2.eventMask);
GetEvent(TASK2, &cgSingleBuffer_TASK2.eventsBuff);
ClearEvent(cgSingleBuffer_TASK2.eventMask);
if(cgSingleBuffer_TASK2.eventsBuff & 0x01)
GEN_IN_CURRENT(TASK2_EV);
}
TerminateTask();
}
```

In order to make that task run, it must explicitly call the *start* operation for it. The definition of the start(TASK2) function, in that case, is:

```
#define start_activity_TASK2 { cgGlobalFlags|=
BITAC_TASK2; start_activity_CTRL2cnt1;
ActivateTask(TASK2); }
```

Note that now it is possible to terminate that TASK, by calling *stop* for it. Thus, it is possible to combine the benefits of an *EXTENDED TASK*, regarding reaction time while it is alive, as well as having the advantage of reusing RAM while running. Stopping those tasks that are not required in certain application configurations, and activating other tasks in the new configuration; all in run-time!

For an *EXTENDED TASK*, certain intertask communication/ synchronization mechanisms are available, using the *TASK EVENT* operator.

EXTENDED TASK may enter the *WaitEvent* state waiting for some of its *TASK EVENT(s)* to be set. The *EVENTs* the task is waiting for are marked using the event mask given to the *WaitEvent* API. Only the task that “owns” the *TASK EVENT* may wait for it to be set. However, the *TASK EVENT* might be set from any place in the code. MicroC links *TASK EVENT* to MicroC events. For example: consider the *EXTENDED TASK*, *TASK4*, with 2 associated *TASK EVENTs: EV1*, with mask 0x02, and *EV2*, with mask 0x04. The following code will be generated in order to link TASK EVENT to MicroC events:

```
...EXTENDED TASK BODY
WaitEvent(cgSingleBuffer_TASK2.eventMask);
GetEvent(TASK4, &cgSingleBuffer_TASK2.eventsBuff);
ClearEvent(cgSingleBuffer_TASK2.eventMask);
if(cgSingleBuffer_TASK2.eventsBuff & 0x02)
GEN_IN_CURRENT(EV1);
if(cgSingleBuffer_TASK2.eventsBuff & 0x04)
GEN_IN_CURRENT(EV2);
…EXTENDED TASK BODY (continued)
```

Note

The “GEN_IN_CURRENT” call sets the internal event passed to it as an argument in the next iteration of the task, which is in the “current” step of it.

Thus, if any reaction in the content of TASK4 is waiting for example to EV1 to be set, i.e., reaction like “EV1/ACT1(),” which will be translated, as explained later, into: “if (EV1) {ACT1();}” will be executed once the associated TASK EVENT 0x02 was set.

On the other hand, when certain action set a TASK EVENT: “[C1]/EV1,” which will be translated, as explained later, into: “if (C1) {GENERATE_EVENT(EV1);};” the following code will be generated for the GENERATE_EVENT(EV1) call:

```
cgEventMsgMask = 0x02;SetEvent(TASK2, cgEventMsgMask);
```

In addition, the following definitions will be made to link the TASK EVENT 0x02 (EV1) with the internal event EV1:

```
#define BIT_EV1 0x01
#define GEN_IN_CURRENT_EV1
(cgDoubleBufferOld_TASK2.cg_Events |= BIT_EV1)
#define EV1 ((cgDoubleBufferOld_TASK2.cg_Events &
BIT_EV1) != 0)
```

Such that the line in the above TASK body code:

```
if(cgSingleBuffer_TASK2.eventsBuff & 0x02)
GEN_IN_CURRENT(EV1);
```

Will set the internal event EV1 bit (BIT_EV1) thus linking the TASK EVENT mask 0x02 of EV1 to its internal bit 0x01 (BIT_EV1) .

Interrupt Service Routine Activities

An Interrupt Service Routine (ISR) runs once, upon activation, and then ends. For OSEK 2.0, MicroC identifies three ISR categories: 1, 2, and 3.

ISR Categories

The decision of which ISR category to use depends on the content of the functions it runs. According to the OSEK/OS specification, it is not allowed to call any OS API function from ISR category 1. For ISR categories 2 and 3, it is allowed to call some of the OS API functions only within the code section marked by *EnterISR()/LeaveISR()* calls.

The form of the generated code frame for an ISR depends on the.Structuring Language: Activity chart Implementation category and content. Some examples are shown below.

Example 1:

The code for an ISR category 1 or 2, named ISR0, containing Activities I01 and I02 without controller will be as follows:

```
ISR (ISR0)
{
cgActivity_I01();
cgActivity_I02();
}
```

Example 2:

The code for an ISR category 3 function named ISR0, containing Activities I01 and I02 without controller will be as follows:

```
ISR (ISR0)
{
EnterISR();
cgActivity_I01();
cgActivity_I02();
LeaveISR();
}
```

Example 3:

The code for an ISR category 3 function named ISR1, containing Activities I11 and I12 and a controller named CTRL1 will be as follows:

```
ISR (ISR1)
{
EnterISR();
do {
cgGlobalFlags &= ~BITSUPERSTEP_ISR1; MicroC 41
TASK/ISR Run Modes
cgActivity_I11();
cgActivity_I12();
cgActivity_CTRL1cnt1();
} while ( (cgGlobalFlags & BITSUPERSTEP_ISR1) != 0);
LeaveISR();
}
```

TASK/ISR Run Modes

A TASK/ISR can have one of the following run modes:

- **Single Step**— The TASK/ISR always runs a single step, then returns handling to the operating system.
- **Super Step**— The TASK/ISR runs the necessary number of tasks before returning handling to the operating system.

When you define a run mode, make the following checks:

1. Check the internal value once before executing the logic.

   In **Single Step** mode, check the internal value before calling any “logic” code, such as mini-spec, Activity, ControlAct, and so on.

   In **Super Step** mode, check the internal value before calling the loop that handles the logic, and the test for the need of another step. Note that choosing this run mode might result in an infinite loop for the TASK/ISR.

2. Check the internal value after logic execution.

   In **Single Step** mode, check the internal value after the call to any logic code.

   In **Super Step** mode, check the value inside the loop that handles the logic, and recheck the value after the calls to any logic code.

3. In **Super Step** mode, check the value after each logic execution. Check the value inside the loop that handles the logic, and recheck the value after calls to any logic code. This check is not relevant for **Single Step** mode.

Super Step Example

For example, if you select **Super Step** mode:

1. Check for internal value changes before logic execution.
2. The code for testing derived events and generating them is moved from the `do...while` loop of the Task and after the functional code. All the calls to the Activities and Controls in the Task are moved to be before the `do..while loop` of the Task (just like the test for buffered access elements and derived events on them).

Consider the following code:

```
void TASK_SINGLE_STEP(void)
{
do
{
cgGlobalFlags &=~BITSUPERSTEP_TASK_SINGLE_STEP;
cgActivity_SINGLE_STEP_CTRL();
if(!lval_COND && COND != lval_COND)
{
GENERATE_EVENT(BECAME_FALSE_COND);
};
if(lval_COND && COND != lval_COND)
{
GENERATE_EVENT(BECAME_TRUE_COND);
};
if(COND != lval_COND)
GENERATE_EVENT(CHANGED_COND);
};
if(cgDoubleBufferNew_TASK_SINGLE_STEP.cg_Events)
cgGlobalFlags |= BITSUPERSTEP_TASK_SINGLE_STEP;
cgDoubleBufferOld_TASK_SINGLE_STEP =
cgDoubleBufferNew_TASK_SINGLE_STEP;
cgDoubleBufferNew_TASK_SINGLE_STEP.cg_Events = 0;
}
while ((cgGlobalFlags & BITSUPERSTEP_TASK_SINGLE_STEP)
!= 0);}
```

The resultant MicroC code is as follows:

```
void TASK_SINGLE_STEP(void)
{
if(!lval_COND && COND != lval_COND)
{
GENERATE_EVENT(BECAME_FALSE_COND);
};
if(lval_COND && COND != lval_COND)
{
GENERATE_EVENT(BECAME_TRUE_COND);
};
if(COND != lval_COND)
{
GENERATE_EVENT(CHANGED_COND);
};
do
{
cgGlobalFlags &=~BITSUPERSTEP_TASK_SINGLE_STEP;
cgActivity_SINGLE_STEP_CTRL();
if(cgDoubleBufferNew_TASK_SINGLE_STEP.cg_Events)
cgGlobalFlags |= BITSUPERSTEP_TASK_SINGLE_STEP;
cgDoubleBufferOld_TASK_SINGLE_STEP =
cgDoubleBufferNew_TASK_SINGLE_STEP;
cgDoubleBufferNew_TASK_SINGLE_STEP.cg_Events = 0;
}
while ((cgGlobalFlags &
BITSUPERSTEP_TASK_SINGLE_STEP)!= 0);
}
```

Single Step Example

If you select **Single Step** mode, the code generated for the Task will not include the `do..while` structure— this creates a single-step Task. In this case, there is no need for the *NeedAnotherStep* bit named `BITSUPERSTEP_<TASK-NAME>` to be allocated, so all references to it are removed.

There are references to `BITSUPERSTEP_<TASK-NAME>` in:

- The `do...while` of a Task.
- In the non-Inline of *NeedAnotherStep* mode— at the end of the *cgDo_* function, there is a check if any *nextStep* is different than the *currentStep*. If it is, the `BITSUPERSTEP` is set.
- In the Task Code frame, there is a check if there are any events pending. If there are, the `BITSUPERSTEP` is set.
- When using an SCH in a generic, the `BITSUPERSTEP` of its task is passed via its structure. In this case, these references should not exist.

Consider the original code:

```
void TASK_SINGLE_STEP(void)
{
do
{
cgGlobalFlags &=~BITSUPERSTEP_TASK_SINGLE_STEP;
cgActivity_SINGLE_STEP_CTRL();. MicroC 45
Single Step Example
if(cgDoubleBufferNew_TASK_SINGLE_STEP.cg_Events)
cgGlobalFlags |= BITSUPERSTEP_TASK_SINGLE_STEP;
cgDoubleBufferOld_TASK_SINGLE_STEP =
cgDoubleBufferNew_TASK_SINGLE_STEP;
cgDoubleBufferNew_TASK_SINGLE_STEP.cg_Events = 0;
} while ((cgGlobalFlags &
BITSUPERSTEP_TASK_SINGLE_STEP) != 0);
}
```

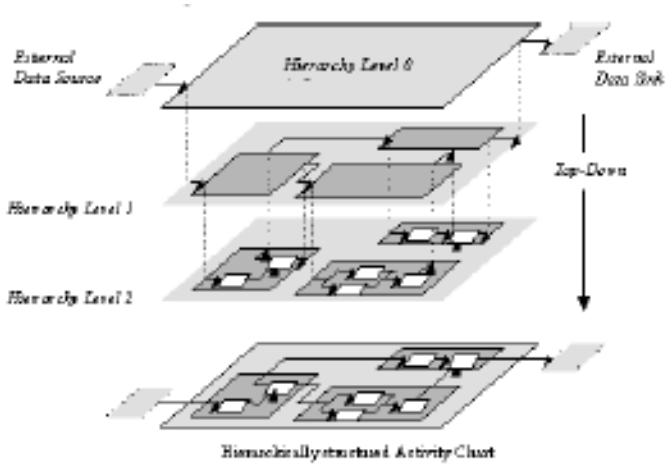
The resultant code is as follows:

```
void TASK_SINGLE_STEP(void)
{
cgActivity_SINGLE_STEP_CTRL();
cgDoubleBufferOld_TASK_SINGLE_STEP =
cgDoubleBufferNew_TASK_SINGLE_STEP;
cgDoubleBufferNew_TASK_SINGLE_STEP.cg_Events = 0;
}
```

Decomposition Language: Activity Chart Implementation

The Activity Chart is the focus of the graphical language used to decompose functionality into realizable sub-functions.

The classic illustration of Functional Decomposition is shown the following figure:



With regard to decomposition in MicroC, each non-basic Activity is being composed out of its sub-activities. We distinguish between 2 cases. The first case is when that non-basic Activity does not contain immediate descendant that is a control activity. In that case, all of the sub-activities are considered *active* when that Activity is *active*. The code for such a non-basic Activity (e.g. A11 with sub-activities A111 and A112 and with no immediate descendant controller) will look like:

```
void
cgActivity_A11acy1(void)
{
cgActivity_A111();
cgActivity_A112();
}
```

The order in which the sub-activities are called within the A11 Activity body is determined by the sub-activities attribute “Execution Order,” as defined in the Dictionary of A111, A112 and A113. In the example above, the “Execution Order” of sub-activity A111 is 1 and of A112 is 2. When that attribute is not set, the calling order is not defined.

If it is desired to save function calls overhead, it is possible to set (in the **Compilation Profile->Options->Settings->General Tab** dialog) the field to true. The resulting code will be:

```
#define cgActivity_A11acy1()\
{\
cgActivity_A111();\
cgActivity_A112();\
}
```

This setting is always applicable for non-TASK and non-ISR, and will not be repeated as the result might be derived from the examples given adding the “ \” at the end of each line, and defining instead of function a C Preprocessor macro.

Sub-Activities Code

Assuming both A111 and A112 are basic Activities, the basic activity can be defined in one of three activation modes:

- •Reactive controlled
- Reactive self
- Procedure like

For **Reactive controlled** and **Reactive self** modes, the code body of the Activity will look like the following code frame:

```
void
cgActivity_A111(void)
{
… Body implementation
}
```

While for the **Procedure like** mode, the code body of the Activity will look like the following code frame:

```
void
cgActivity_A112(void)
{
if ((cgActiveActivities1 & BITAC_A112) != 0) {
… Body implementation
stop_activity(A112);
}
}
```

The differences between the three will be found in the activation rules for each mode. **Reactive controlled** and **Reactive self** modes will perform a step, while they are active, each time the TASK containing them is running. Usually, a TASK will perform a *run to stable* run (also called a **super step**), that might require few steps (also called **micro step**s). Those using **Reactive controlled** and **Reactive self** modes will participate in each of the steps.

Procedure like mode performs only a single step each time the TASK containing it is running. At the beginning of the TASK, the relevant Activity active bit will be set. Then the Activity body will unset that bit after it ran, calling `stop_activity`.

Another difference, between **Reactive controlled**, **Reactive self** and **Procedure like**, is in the allowed syntax of the Mini-Spec which is described later.

Adding the controller A11_CTRL to A11 will make the code look like:

```
void
cgActivity_A11acy1(void)
{
cgActivity_A111();
cgActivity_A112();
cgActivity_A11_CTRLcnt1();
}
```

With the controller function, `cgActivity_A11_CTRLcnt1()`, like:

```
void
cgActivity_A11_CTRLcnt1(void)
{
cgDo_A11_CTRLcnt1();
}
```

The implementation of `cgDo_A11_CTRLcnt1()` depends on whether A11_CTRL is implemented as a Statechart or as a Flowchart. In this discussion we will only show a brief descriptions of each; a more detailed description is given later in the appropriate sections.

For a Statechart implementation:

```
void
cgDo_A11_CTRLcnt1(void)
{
StateInfo_A11_CTRLcnt1 nextState_A11_CTRLcnt1 = 0;
if (currentState_A11_CTRLcnt1 == 0) {
nextState_A11_CTRLcnt1 = FS_A11_CTRLst2;
}
else
{
… Rest of the Statechart logic
}
if (nextState_A11_CTRLcnt1 != 0) {
if (currentState_A11_CTRLcnt1 !=
nextState_A11_CTRLcnt1)
cgGlobalFlags |= BITSUPERSTEP_TASK1;
currentState_A11_CTRLcnt1 = nextState_A11_CTRLcnt1;
}
}
```

For a Flowchart implementation:

```
void
cgDo_A11_CTRLcnt1(void)
{
… The Flowchart logic
}
```

Activities within a certain TASK can communicate with each other using various method. Within a single TASK/ISR boundary, the Activity Chart Graphical Language of MicroC shares most of the semantics used in the Activity Chart Graphical Language of Rational Statemate. However, there are few discrepancies between those languages that should be noticed, and will be mentioned shortly below. The interaction between TASK/ISR and communication between Activities not residing in the same TASK/ISR has nothing equivalent in the language of Rational Statemate, and should only use the services provided by the run time OS, also described below. Activities defined to be TASK/ISR have already been discussed, but it must be remembered that such Activities are not fully compatible with the Activities that can be defined in Rational Statemate.

Discrepancies between MicroC Regular (i.e., not a TASK nor ISR) Activities running under the same TASK/ISR and Rational Statemate Activities that should be noticed include the following:

- Stable state criteria
- Implicit termination of Activity as result of termination of all its sub-activities
- Suspend, resume modes
- Status sensing – stopped/started
- Implementation as CA (Not supported at all in MicroC)

Note

We do not include here those language features that are only temporarily not supported, but will be supported in coming release of the product. Instead, we are focusing our discussions on those aspects that are not expected to change.

Communication and Synchronization Services

Communication and synchronization services between Activities, possibly not residing in the same TASK/ISR, include the following:

- Messages (for OSEK 2.0: non-queued and queued Messages)
- Signals (for OSEK 2.0:TASK EVENT)
- Semaphores (for OSEK 2.0: resources)
- Global data

Messages

The first communication mechanisms use the OSEK Messages support capabilities provided by MicroC.

The first of those, **Non-Queued messages**, uses a message identifier (i.e. the message name) to share data between various tasks in the application. The sender and or receiver TASK of such a message might be running in the same ECU, sharing the same memory address space, or running across an ECU network on some remote MCU. The user of the message need not be aware of the concrete implementation. Thus, using that mechanism ensures that the resulting design is correct, flexible and efficient.

Queued Messages

Queued messages use a very similar implementation mechanism to that for Non-Queued messages. The difference being in that those types of messages do not contain *value* but rather signals the occurrence of some event. Again, using these in a design makes the design easier to modify.

Note

Examples and discussion regarding each of those two methods is given in Rational Statemate Action Language Implementation.

Signals

The third method (i.e. those of TASK EVENT) is somewhat different from the first two. Similar to the *Queued messages*, described above, they signal the occurrence of some event. However, as they are not queued, there is no information regarding how many such events occurred until being processed. An additional difference is that a TASK EVENT must address a specific TASK with a specific EVENT, thus requiring knowledge of the application structure. A TASK EVENT implementation is much more efficient than the previously mentioned communication methods, however it requires the TASK to be of type **EXTENDED**, which is not always possible or efficient. The downside of requiring knowledge of the application structure is balanced by the improved performance. Those are design decisions that should be made regarding a specific problem at hand. Examples and further discussion can be found in **Structuring Language: Activity Chart Implementation**.

Global Data Usage

Global data usage is the fourth method of communication. As always in real time applications, caution should be made regarding the validity of the data when running in preemptive environment with multiple tasks and ISRs. The protection mechanism supported is the OSEK RESOURCE mechanism, which is similar to a binary semaphore Similar added, meant to help in protecting data and access to common resources.

Semaphores

Examples and discussion of using OSEK RESOURCE is given in **Rational Statemate Action Language Implementation**.

All the above said, it is a common situation that data is arriving through the bus or board ports, in some predefined messages and addresses, and is needed to be produces, again, to the bus or board, in some maybe other predefined messages and addresses.

In this situation the decision is rather easy, as it already has being taking, and the designer simply uses the defined interface for his application. However, the discussion above is relevant when one tries to build up implementation that will obviously use the appropriate interfaces, however will also be easy to maintain, modify and ported to various other environments, usually unknown at design time.

Statechart Implementation

Statecharts are used to define the behavior of a Control Activity. For the purposes of code generation in MicroC and our discussion here, a single Statechart is considered to be the Statechart directly connected to a Control Activity, all of its sub-charts, and the generics instantiated within them. In short, all the states under the root are represented by the control Activity.

For example, for the control Activity *A11_CTRL*, the following two functions will be generated:

```
void cgActivity_A11_CTRLcnt1(void)
void cgDo_A11_CTRLcnt1(void)
```

The bodies of the generated code for these functions resembles the following:

```
void
cgDo_A11_CTRLcnt1(void)
{
StateInfo_A11_CTRLcnt1 nextState_A11_CTRLcnt1 = 0;
if (currentState_A11_CTRLcnt1 == 0) {
nextState_A11_CTRLcnt1 = FS_A11_CTRLst2;
}
else
{
… The rest of the Statechart logic
}
if (nextState_A11_CTRLcnt1 != 0) {
if (currentState_A11_CTRLcnt1 !=
nextState_A11_CTRLcnt1)
cgGlobalFlags |= BITSUPERSTEP_TASK1;
currentState_A11_CTRLcnt1 = nextState_A11_CTRLcnt1;
}
}
```

```
void
cgActivity_A11_CTRLcnt1(void)
{
cgDo_A11_CTRLcnt1();
}
```

Note that the function *cgActivity_A11_CTRLcnt1()* simply calls *cgDo_A11_CTRLcnt1()*. A more detailed discussion of the *cgDo_…* function is found below.

Note

- Further Optimization: This might be changed, as the wrapping function, “cgActivity_A11_CTRLcnt1” in the above example, could be dropped.
- Use the **Compilation Profile >Setting >General >Use Macros** flag to control function generation vs. pre-processor macro.

Statechart Implementation: Data Usage

A *StateInfo* data type will be defined, and a few variables of that type will be declared, when a statechart is created.

For the previous example, the *StateInfo* data type would be named **StateInfo_A11_CTRLcnt1** and will be defined as an unsigned type of either 8, 16 or 32 bits; like “ *typedef int8 StateInfo_A11_CTRLcnt1*”. The size depends on the topology of the Statechart which is described later in this document.

The *StateInfo* variables will be *currentState, nextState, staySame.* For the example of *A11_CTRL*:

```
StateInfo_A11_CTRLcnt1 currentState_A11_CTRLcnt1;
(global variable)
StateInfo_A11_CTRLcnt1 nextState_A11_CTRLcnt1;
(automatic variable)
StateInfo_A11_CTRLcnt1 staySame_A11_CTRLcnt1;
(automatic variable)
```

The *currentState* and *nextState* variables will always be allocated. The *staySame* variable will be allocated only if either of the entering or exiting reaction functions is required, as discussed below.

currentState is allocated as global variable, while *nextState* and *staySame* are allocated as local, automatic, variables to the statechart function *cgDo_… .*

Note

Further Optimization: In specific topologies it is possible to use only a single StateInfo variable, i.e. the currentState.

Statechart Implementation: Generated Functions

Statechart Code Frame

Consider the following example of code generated from a Statechart (Note: line numbers are included in this code sample for discussion purposes):

```
1 void
2 cgDo_A11_CTRLcnt1(void)
3 {
4 StateInfo_A11_CTRLcnt1 nextState_A11_CTRLcnt1 = 0;
5 if (currentState_A11_CTRLcnt1 == 0) {
6 nextState_A11_CTRLcnt1 = FS_A11_CTRLst2;
7 }
8
9 else
10 {
11 … The rest of the Statechart logic
12 }
13 if (nextState_A11_CTRLcnt1 != 0) {
14 if (currentState_A11_CTRLcnt1 !=
nextState_A11_CTRLcnt1)
15 cgGlobalFlags |= BITSUPERSTEP_TASK1;
16 currentState_A11_CTRLcnt1 =
nextState_A11_CTRLcnt1;
17 }
18 }
```

In general, the overall code frame of a Statechart looks like the *cgDo_A11_CTRLcnt1* function shown above. However, you will discover in the following discussions that code frame is not fixed.

Line 4 resets the *nextState* variable. This variable will be set only if a transition has been made, and will hold the new state configuration of the Statechart.

Lines 13 and 14 check the *nextState* variable, to determine if a transition was taken and whether to enforce another step in the TASK holding the Statechart...

```
Line 15: cgGlobalFlags | = BITSUPERSTEP_TASK1
Line 16: currentState_A11_CTRLcnt1 = nextState_A11_CTRLcnt1
```

advances the Statechart configuration a step, to hold the configuration of the next step.

Lines 5 to 12 will be replaced with specific code resulting from the specified Statechart logic. For example, two additional functions might be commonly generated here: *entry actions* and *exit actions.* If the Statechart logic requires entering/exiting reactions, the functions will resemble the following:

```
void
cgEnterActions_A11_CTRLcnt1(void)
{
… entering reactions code
}
void
cgExitActions_A11_CTRLcnt1(void)
{
… exiting reactions code
}
```

When either of these function are needed, the following changes to *cgDo_…* will also be made:

```
void
cgDo_A11_CTRLcnt1(void)
{
StateInfo_A11_CTRLcnt1 nextState_A11_CTRLcnt1 = 0;
staySame_A11_CTRLcnt1 = 0;
if (currentState_A11_CTRLcnt1 == 0) {
nextState_A11_CTRLcnt1 =
FS_DefaultOf_Chart_A11_CTRL;
}
else
{
… The rest of the Statechart logic
}
if (nextState_A11_CTRLcnt1 != 0) {
cgExitActions_A11_CTRLcnt1();
cgEnterActions_A11_CTRLcnt1();
if (currentState_A11_CTRLcnt1 !=
nextState_A11_CTRLcnt1)
cgGlobalFlags |= BITSUPERSTEP_TASK1;
currentState_A11_CTRLcnt1 = nextState_A11_CTRLcnt1;
}
}
```

Of course, the function calls to cgExitActions_A11_CTRLcnt1() required. See **Optimization Algorithms** for information on the MicroC algorithms that create more efficient code.

Order of Function Execution

The order of doing exiting actions, entering actions, transition actions and static reactions for a state is as follows:

1. Static reactions are done, as the generated code reveals, in descending order down the state hierarchy, where the state has not changed.
2. When a transition is detected, then the transition action is done immediately.
3. Exiting actions are then done, in which all the states that are exited are given an opportunity to do exiting static reactions. Exiting reactions are done from the innermost state to the outermost state.
4. Entering actions are then done, in which all the states that are entered are given an opportunity to do entering static reactions. Entering reactions are done from the outermost state to the innermost state.

Note

In specific topologies it is possible, and more efficient, to put the entering/exiting reactions inline, while taking the transition. Use the **Compilation Profile->Setting->Optimization** flags.

In the balance of this section we will discuss the implementation of the following language features:

- Default state implementation
- AndState implementation
- Timeout implementation
- History and Deep History implementation
- Short list of guide lines to get the most efficient code

Default State Implementation

The Default connector is treated as a state. This means that when a state is entered, we spend one *step* going into the Default.Statechart Implementation and then on the following step, we actually enter the desired state. Note that this is a slight change to the classic Language of Rational Statemate semantics. The motivation behind this change is that, as it is allowed to put a guard on the default transition, it might be that no transition could be taken. This means that the code might otherwise get *stuck* in a default connector.

In a practical sense, however, this does not represent a significant difference and should be negligible in any practical example.

Note

In specific topologies – when there is no guard on the default transition it is possible to directly enter the default state.Use the **Compilation Profile->Setting->Optimization** flags.

AndState Implementation

The implementation of AndState compresses a few otherwise different StateInfo variables into a single one, thus using potentially less RAM. However, in order to relate to each of the different parallel state hierarchies, some ROM is required to implement bit-masking. As a general rule, it is preferable (from a code size perspective) to use AndState when having few independent very small statecharts. The difference in the generated code will be that instead of few control activities, each having related `cgDo_...` functions as is the case with a few Statecharts, here only one such control activity is required with one related `cgDo_...` function. This code for such a function will only appear once. The function's code frame would resemble the following:

```
…
StateInfo_A11_CTRLcnt1 nextState_A11_CTRLcnt1 = 0;
if (currentState_A11_CTRLcnt1 == 0) {
nextState_A11_CTRLcnt1 = FS_A11_CTRLst2;
}. MicroC 65
Timeout Implementation
else
{
… The rest of the Statechart logic
}
if (nextState_A11_CTRLcnt1 != 0) {
if (currentState_A11_CTRLcnt1 !=
nextState_A11_CTRLcnt1)
cgGlobalFlags |= BITSUPERSTEP_TASK1;
currentState_A11_CTRLcnt1 = nextState_A11_CTRLcnt1;
}
…
```

On the other hand, the test for being in leaf-state will be done using the *inState* test and not *inLeafState* test. The *inState* test requires one more integer comparison then the *inLeafState* test. Thus, it is recommend for each particular case that the developer test both options and compare the results to choose the optimum implementation.

Timeout Implementation

Software Counter(s) are used as the basis for the implementation of timeouts. When a timeout or delay is set, the current value of the relevant Software Counter will be added to the requested delay time, and stored in a variable, using a defined macro: `INSTALL_TIMEOUT`. By default, MicroC relates to the primary Software Counter defined in the compilation profile.

Note

Use **Compilation Profile >Setting >OS >System Counter Timer** to define the primary Software Counter.

Other Software Counter(s) might be referenced using an optional third argument in the timeout operator. The name of the counter is as written in the model using the syntax:

```
tm(en(S1), 12, myCounter)
```

In this example, the name of the counter is: `myCounter`. Each counter receives an index value defined as <*counter_name*>*_INDEX*. That index value identifies that specific counter in the application.

Note

The counter definition is found in the “macro_def.h file.

The `INSTALL_TIMEOUT` macro has three arguments:

- The name of the event
- The requested delay
- The index of the Counter that it is pending on

This allows the code to reuse the same `Timeout` variable with different counters. The first argument is concatenated to the INSTALL macro, as shown here. In the code, a call like the following will be used:

```
INSTALL_TM(tm_999999962, 10, SYS_TIMER)
```

This call will set a timeout to expire 10 ticks from the current time of `SYS_TIME`R. The macro itself will be defined as follows:

```
#define INSTALL_TM_tm_999999962(D, C) \
cgTimeoutsMask |= tm_999999962_TM_MASK;\
tm_999999962_TIME = currentTick + (D);
```

That call will assign to tm_999999962_TIME which is a variable of type Timeout Variable Type the current counter value, help in currentTick plus the requested delay time help in D. In addition, the bit tm_999999962_TM_MASK is set to flag that this timeout is pending.

A test for Timeout expiration is done in the function:

```
genTmEvent_<CTRL_CHART_NAME>(<Timeout Variable Type>
currentTickVar, <Buffer> * buff, uint8 counterIndex)
```

The third parameter *uint8* `counterIndex`, holds the index of the Counter that is referred to in the current call to this function. Before each call to this function, the correct counter would be read into the *currentTick* global variable.

For each Timeout Variable there are three options for code generation inside the `genTmEvent_...` function:

1. When there is only one Counter in the model no check will be made for the counter.
2. When there is only one counter that the timeout.variable can be installed for, then the code will look.like:

```
if(counterIndex == <ITS_COUNTER_NAME>_INDEX &&
cgTimeoutsMask & tm_999999993_TM_MASK &&
currentTickVar >= tm_999999993_TIME) {
GEN_IN_BUFF(tm_999999993, buff);
cgTimeoutsMask &= ~tm_999999993_TM_MASK;
}
```

3. If there is more than one counter that the Timeout Variable can be installed for, then the code will include the following provisions:

   In the `glob_dat.c` file an `uint8` variable is generated: tm_999999993_counter¡that holds the index of the current relevant counter.

   In `macro_def.h` file along with the previous code that was generated for the `INSTALL_TIMEOUT` macro, there is one more statement that keep the `INDEX` of the counter that the timeout was installed for.

The index that is passed to the function is compared with the index of the counter that was used when the timeout was installed. This enables the application to identify the counter that the timeout is pending on.

When the option *Generate Timer Overflow Task* is selected, in the compilation profile setting, then the following code elements are generated:

OSEK 2.0 Implementations

OSEK-targeted applications have special requirements:

1. For each Counter, an overflow Task named `<counter_name>_OVERFLOW` is generated. This includes the task declaration (found in `os_decl.h`) and body code (found in `glob_func.c`).
2. In each Task there is overflow management provided **only** for the Timeouts variables that are referring to the specific counter.
3. For each Counter, an Alarm named `<counter_name>_ALARM` is generated. This includes the alarm declaration (found in `os_decl.h`) and installation (found in `macro_def.h`). In the `macro_def.h` file, a new macro is generated:

```
#define SET_ADDITIONAL_OVERFLOW_ALARMS() {\
SetAbsAlarm(<counte_name>_ALARM, 0,
OSMAXALLOWEDVALUE);\
}
```

This macro installs all the overflow alarms that activates the overflow tasks. A call to this macro is in the file `<profile>.c` after the installation of the `SYS_TIMER_ALARM` (formerly known as `SYS_TIME_OVERFLOW`).

Compare that to non-OSEK implementations:

1. For each counter, an overflow function named `on<counter_name>_OVERFLOW` is generated. In each Task, overflow management is provided only for the Timeouts variables that refer to that specific counter.
2. **IMPORTANT** - there is no call to these functions in the generated code. Therefore, in order to use them, additional code should be added by the developer that decides when to call these functions (on overflow), possibly in `usercode.c`.

Note

Set from within the Code Generation Profile Editor. Use **Options->Settings->General->Timeout Variable Type**.

The goal is to have a variable that is bigger then the counter, thus avoiding the “value overflow” problem.

Note

(**OSEK only**) When a TASK/ISR has related timeouts, MicroC calls GetResource(RES_SCHEDULER)/ ReleaseResource(RES_SCHEDULER) around the code section that swaps the TASK/ISR event buffer, and both before and after the call to genTmEvent(...) in on<TIMER>OVERFLOW Tasks (in the file `glob_func.c`). This resource usage can be avoided. Within the Code Generation Profile Editor, select **Options > OS TAB > Allow "GetResource(RES_SCHEDULER) Usage**. Uncheck this option.

History and Deep History Implementation

History and Deep History implementation requires a *StateInfo* variable per each state holding a *History Connector(s)* and a *StateInfo* variable per each state holding a *Deep History Connector(s)*.

The state configuration is stored in that *StateInfo* variable, such that when taking a transition into the History/Deep History that configuration is assigned to the *nextState* variable, causing an entrance to the stored state configuration.

When used, the operators `history_clear` and `deep_clear` assign to the corresponding *StateInfo* variable the corresponding default state configuration.

Optimization Algorithms

MicroC includes several algorithms to generate the most efficient code, including:

- **Inline Default Test**
- **Inline Setting of the “Need Another Step” Bit**
- **Inline Entering and Exiting Reactions**
- **Merge State Sequences With No Guard on Transitions**
- **Timeout Optimization**
- **Clutch Entrance to a State Hierarchy**

In addition to these algorithms, use the following guidelines to get the most efficient code:

- Avoid redundant intermediate states (i.e., not persistent states).
- Avoid duplication of code segments— use functions or defined actions instead of hardcoded duplicates.
- For a simple, single state with self-transition scheduling some operation, use static reaction or an ISR.
- Use the state hierarchy to represent priorities.

Inline Default Test

MicroC can inline the initial and default test. Consider the following code:

```
if(currentState_S1 == 0){
currentState_S1 = FS_DefaultOfS1;
} else {...
```

The inlined code generated by MicroC is as follows:

```
if(currentState_S1 == 0 || inState(DefaultOf_S1)){...
```

Inline Setting of the “Need Another Step” Bit

To improve code efficiency, you can specify *No. of Transition <= 0*. This criteria determines whether the optimization is performed. When you apply this optimization, MicroC makes the following changes to the generated code:

- The declaration of `StateInfo_<CTRL Activity`

  `Name > nextState_<CTRL ActivityName = 0;`

  is removed— there is no need for this local variable after the optimization.
- All the assignments to `nextState_<CTRL Activity Name>` are replaced with assignments to `currentState_<CTRL Activity Name>`.
- After every transition, MicroC makes the following assignment:

  `cgGlobalFlags |= BITSUPERSTEP_<Task Name>;`
- The code at the end of the `cgDo...()` is removed. This is the code that was inlined:

```
if (nextState_<CTRL Activity Name> != 0) {
if (currentState_<CTRL Activity Name> !=
nextState_<CTRL Activity Name>)
cgGlobalFlags |= BITSUPERSTEP_<Task Name>;
currentState_<CTRL Activity Name> =
nextState_<CTRLActivity Name>;
}
```

- If a transition is inside an AndState component, the assignment to *currentState* includes a reset of the bits that represent the component that is the LCA of the transition. For example:

```
nextState_OPT_NEXT_STATE_CTRL =
(nextState_OPT_NEXT_STATE_CTR &~
(FM2_<ComponentLCA of Transition>)) |FS_<Next State>;
```

Note

The optimization will not take place if there is an entering or exiting reaction that could not be optimized out.

Inline Entering and Exiting Reactions

Inlining entering or exiting reactions is based on the following criteria:

```
No. of Statements <= 5
No. of Instances <= 999
```

Note that this criteria is based on the *average* number of inlined statements for the number of reaction statements. For example, if the number of reactive statements is 5 and the number of transitions is 10, the average is 5 statements.

An exit reaction is inlined when none of the following scenarios are encountered:

- An AndState exists with the exit reaction, or with a descendant that has an exit reaction.
- The operator `stop_activity` is used for any ancestor of the control activity with which the statechart is connected, at least one state has more than a single descendant, and at least one of its descendants has an exiting reaction.
- A transition exiting from a state exists and has more than a single descendant, and at least one of its descendants has an exiting reaction.

When inlining take place, the exit reaction code is added to the transition code segment after the transition action itself, but before the entering action code. If an inlining scenario is encountered but inlining cannot be performed, MicroC does one of the following:

- If there are entering reactions, MicroC adds a call to the exiting reaction function (`cgExit...`) to the transition code segment. The `cgExit...` function will not be called at the end of the statechart code.
- In the absence of an entering reaction, MicroC does not add a call to the transition code segment. The call to `cgExit...` is done at the end of the statechart code, as occurs when optimization is not used.

Merge State Sequences With No Guard on Transitions

MicroC can merge sibling Or-States when there is a single transition between them that has no guard on it. Consider the following topology:

```
...[S11]-t12->[S12]...(states S11, S12 transition t12)
```

The goal of the optimization is to merge S11 and S12, as well as the static reactions of the two states and the transition action (referred to as “merged actions”). The merge is allowed (considered correct) when the following conditions are met:

- The transition (t12) is the only transition that exits S11 or enters S12.
- The transition (t12) has no guard.
- There is no conflict in double-buffered element assignments and usage in the actions to be merged.
- There is no conflict in event generation and usage in the actions to be merged.
- When the merged states are inside an AndState:
  - There is no conflict in element assignment and usage between merged actions and actions/ reactions in the other AndState components.
  - There is no conflict in event generation and usage between merged actions and actions/reactions in the other AndState components.
- In user function calls:
  - When the usage is Out/Inout, the call is regarded as “assignment/events generation” of the actual function parameters.
  - When the usage is In/Inout, the call is regarded as “usage/events test” of the actual function
  - parameters.
  - Function “Global Usage” elements are ignored.

Note the following:

- No check is done regarding sibling activities.
- When using GBA, no painting is done for the states that have been merged to another state; only the remaining state is painted.
- A reference to/usage of an element of an array is considered as a reference to/usage of the entire array. For example, if you use `MY_ARR[2]=3;` there will be a conflict for the whole array or any member of it, such as `DI=MY_ARR[4];`.
- A reference to/usage of a record field is considered example, if you use MY_REC.F1=3;, there will be a conflict for the whole record or any of its fields, such as `DI=MY_REC.F2;`.

Note

This optimization, when used with the optimizations inline entering/exiting reactions and clutch of state hierarchy might result in an action sequence that is not identical to the action sequence performed without those optimizations. Make sure the difference is acceptable.

Timeout Optimization

The Code Generator performs optimization of data allocated for timeouts. Data allocated for a timeout is reused for another timeout if these timeouts trigger transitions outgoing from exclusive states.

Note

Use the menu selections **Options->Settings->Optimization ->Reuse Timeout Variable** to set that optimization.

Note the following:

- The optimization reuses the same *Timeout/Delay* variable for other timeouts/delays.
- A variable can be reused only if the states waiting for the timeouts are exclusive states.

To reduce the number of data allocations for the timeout operation, the algorithm has been changed. The description of the algorithm uses the following terms:

- **Source state of a timeout**— The source state of the transition that the timeout is on, or the state in which its static reaction contains the timeout.
- **Clutch a timeout**— Add the clutched timeout to the list of timeouts for the timeout that represents the data allocation. Tagging the clutched timeout as `NOT` requires data allocation. In the clutched timeout, the Code Generator keeps a reference to the timeout that represents its data allocation.
- **Parent**— State 1 (S1) is a parent of state 2 (S2) if S1 is an ancestor of S2.

The algorithm merges data allocation for two timeouts if their sources are mutually exclusive. The steps of the algorithm are as follows:

1. Prepare the list of timeouts.
2. For each timeout in the list (in no particular order), look through the other timeouts for a candidate to be clutched to it.
3. For a timeout to be clutched, the following conditions must be met:
   - The timeout being tested requires data allocation.
   - The source state and every timeout in the list are not parents of the source state of the tested timeout, or of any of the source states of the timeouts in the list.
   - The source state of the tested timeouts and the source state of every timeout in its list are not parents of the source state, or the source state of any timeout in the list.
   - None of the following are an AndState:
     - The first common parent of the source state
     - The source state of each of timeout in the list
     - The source state of the tested timeout
     - Any of the timeouts in tested timeout's list

If all four conditions are satisfied, the tested timeout is clutched. The following code sample represents the algorithm:

```
LIST TimeoutsList = CREATE TimeoutsList.
FOR EACH Timeout (TM) in (TimeoutList) DO
{
State TMSourceState = FIND SOURCE STATE OF (State).
LIST SiblingsStatesList = CREATE SIBLINGS LIST OF (TMSourceState)
FOR EACH State (S) in (SiblingsStatesList) DO
{
LIST CurrSiblingTimeoutsLists = CREATE LIST OF TIMEOUTS UNDER (S)
FOR EACH Timeout (T1) in (CurrSiblingTimeoutsLists) DO
{
if (T1) NOT (Requires Data Allocation) then EXIT FOR LOOP
State SiblingTimeoutSourceState = FIND SOURCE STATE OF (T1)
LIST MySourceStatesList = CREATE LIST OF SOURCE STATES OF
(TMSourceState AND ALL THE TIMEOUTS IN ITS LIST)
FOR EACH (S1) in (MySourceStatesList) DO
{
```

```
if (S1) IS PARENT OF (SiblingTimeoutSourceState) OR
(SiblingTimeoutSourceState) IS PARENT OF (S1) then EXIT FOR LOOP
State FirstCommonParentState = FIND FIRST COMMON PARENT OF
(S1, SiblingTimeoutSourceState)
if (FirstCommonParentState) IS AND-STATE then EXIT FOR LOOP
CLUTCH TIMEOUT (T1) TO TIMEOUT (TM). Statechart Implementation
}
}
}
}
```

After the algorithm has finished, each timeout is marked with one of the following tags:

- The timeout requires data allocation, and the list of all the other timeouts that are using its data.
- The timeout does not require data allocation, and a reference to the timeout whose data will be used.

Clutch Entrance to a State Hierarchy

MicroC can perform a clutch of steps, intermediate states, and default states when entering state hierarchy. The clutch entrance algorithm steps directly into the lower-most leaf state in the state hierarchy. All the entering reactions are performed appropriately, according to the state hierarchy.

The algorithm stops at the following items:

- A default transition with guard AND/OR action
- A state with more than a single descendant state and no default
- A state with a History/Deep History connector

Flowchart Implementation

Flowcharts are another graphical language used in MicroC to define the behavior of a Control Activity. For the purpose of code generation, including this discussion, a single Flowchart is considered to be the Flowchart directly connected to a Control Activity, and all of its sub-charts and the generics instantiated within them.

Consider the Control Activity `A12_CTRL`. The following two C functions will be generated for it:

```
void cgActivity_A12_CTRLcnt1(void)
void cgDo_A12_CTRLcnt1(void)
```

The body of these functions look like the following:

```
void
cgDo_A12_CTRLcnt1(void)
{
… The flowchart logic
}
void
cgActivity_A12_CTRLcnt1(void)
{
cgDo_A12_CTRLcnt1();
}
```

The function `cgActivity_A12_CTRLcnt1` simply calls `cgDo_A12_CTRLcnt1`.

Detailed discussions of the `cgDo_…` function construct will be found below.

Note

Further Optimization: This example might be optimized by dropping the wrapping function, `cgActivity_A12_CTRLcnt1`, unless it serves some additional purpose not considered here.

Use the **Compilation Profile->Setting->General->Use Macros** flag to control the use of function generation vs. pre-processor macro.

Flowchart Implementation

The Flowchart Language in MicroC graphically describes a **Structured C Program.** It is used as an alternative approach to the *Rational Statemate Language* emphasis on Statecharts to describe control activity logic.

The graphics and semantics used in Flowcharts are very much different from what is used in Statecharts. This gives the designer the option to choose the language that is best suited to a specific algorithm implementation.

The code of a Flowchart runs from beginning to end, without stopping and without explicitly maintaining its internal state. Each time the Flowchart is run, it must start from the beginning. The Flowchart does not have a notion of *State* or *Internal State*.

While Flowcharts allows the creation of highly visible, graphical algorithms, there is no inherent overhead in the generated code. The code generator is able to generate optimized structured code from a flowchart just as readily as from a statechart.

The use of flowcharts is recommended where ever clear and visible (graphical) algorithms are desired, while preserving maximum performance.

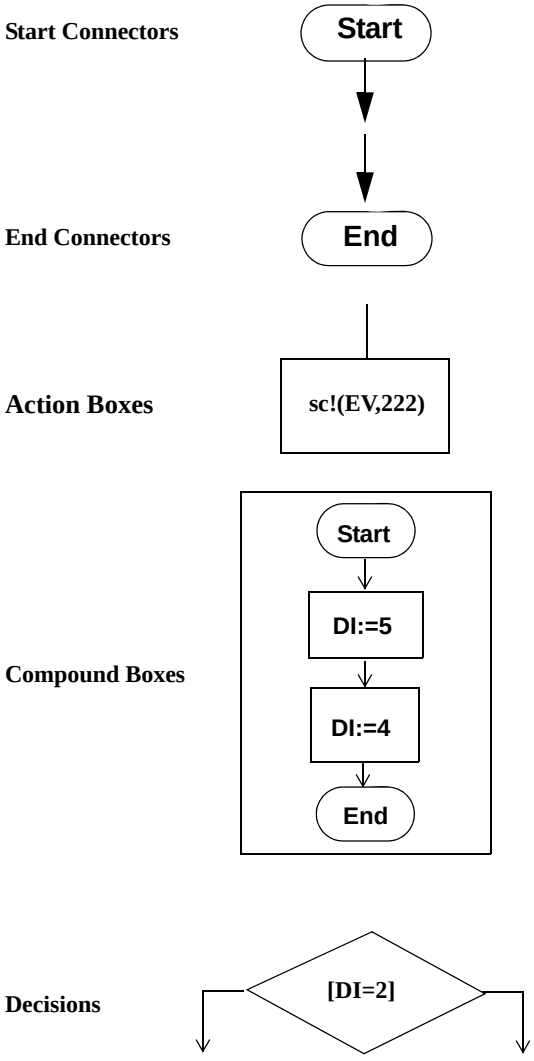
If a Flowchart is properly constructed, it will result in the generation of highly optimized structured code. However, it is the responsibility of the designer to build appropriate charts with proper syntax, logic, and association with a valid control activity. Otherwise, the results could be non-structured code.

Supported Constructs

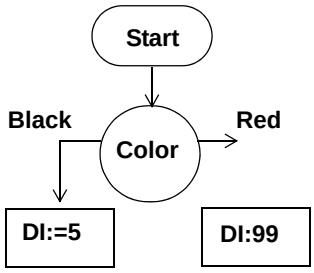
When discussing the structuring of a flowchart, we refer to two categories of graphical elements:

- Boxes
- Arrows

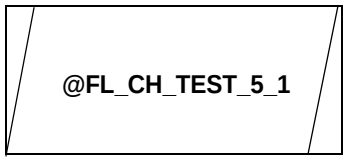
Compound Boxes, that is boxes containing other boxes, represent code blocks. Individual graphic elements include Those shown in the following figure.



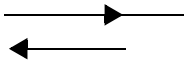
Switches



Instance (off Page) Boxes



Arrows



Labels

As with statecharts, the graphical elements in flowcharts can be assigned labels for purposes of identification and describing associated logic or value assignments. Labels on arrows are considered to be **literal constants** and are allowed *only* for arrows exiting either Decision or Switch elements.

Decision Expressions

Allowed expressions in “Decision” are:

- Event (like: ON_POWERUP)
- Condition (like: [POWER_ON])
- Expressions (like: [TEMP > 27])

Allowed expressions on Arrows exiting “Decision” are:

- yes
- no
- true
- false

Switch Expressions

Allowed expressions in “Switch” are:

- Value type expressions (like: F1(3) + 5)

Allowed expressions on Arrows exiting “Switch” are:

- Literal constants:

  else

  default

Forbidden Constructs

There are some uses of graphical elements that are not allowed in flowcharts here. These include the following:

- Arrows that cross the boundaries of boxes are not allowed.
- Arrows may only go between sibling boxes.

Goto Minimization

The code generator tries to minimize the number of `goto` statements that are needed. This tends to make the code readable and structured. However, this is not always possible and `goto` statements may appear in the generated code.

Restructuring the flowchart or using statecharts instead of flowcharts may eliminate generated `goto` code.

Code Structure

The code is generated in C-Blocks. Compound (non-basic) boxes are translated into blocks. Basic boxes are interpreted as control positions between executable statements.

Begin/End Points

The "**START**" point for each block, i.e. the entering point to the non-basic box, is marked using a Start arrow in that box. The "**END**" point is marked using an End connector in that box. Specifically, the "**START**" point for the execution of the whole Flowchart is marked using a Start arrow in the upper most level. The "**END**" point for the whole Flowchart is marked using End

connector in the upper most level.

The Flowchart execution will stop as soon as it can make no more progress. This may be due to reaching an End connector, or it may be because it reaches some box for which all the outgoing arrows have triggers that evaluate to false.

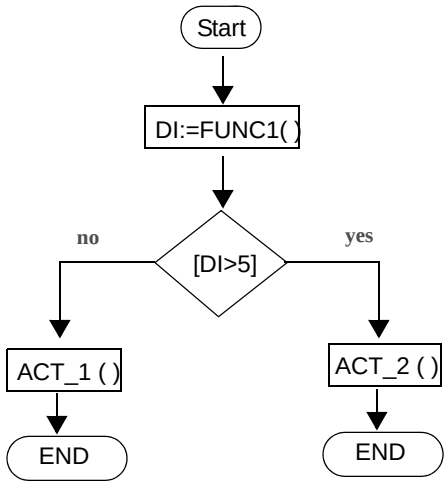
Arrows and Labels

In the case of nested boxes, all arrows on the inside boxes are tried first. If none of them can be taken, then "higher level" arrows are tried. If none of them can be taken then higher level arrows are tried, etc. If no arrows can be taken, then the code finishes executing, i.e., the function returns.

Flowchart Examples

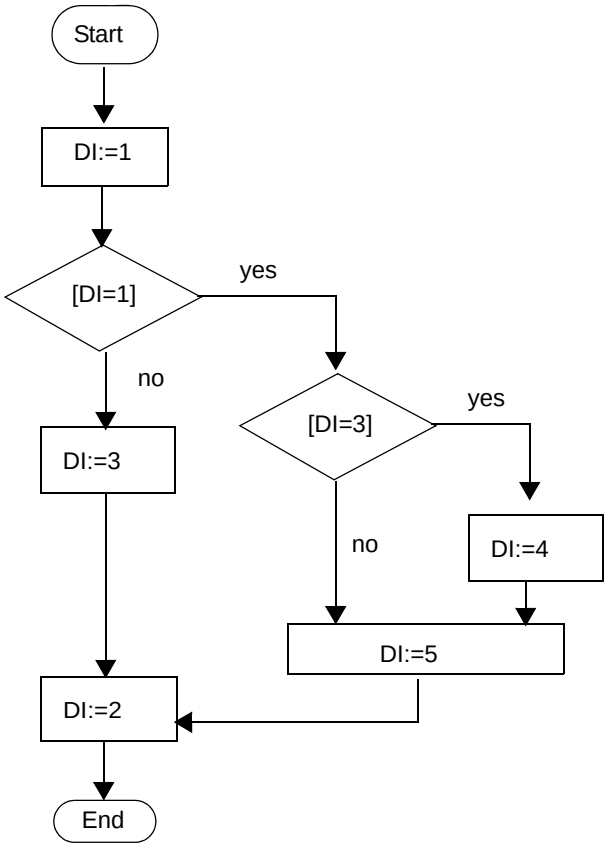
In the following examples, we give the graphics and then the generated code for the graphics.

Simple Flowchart



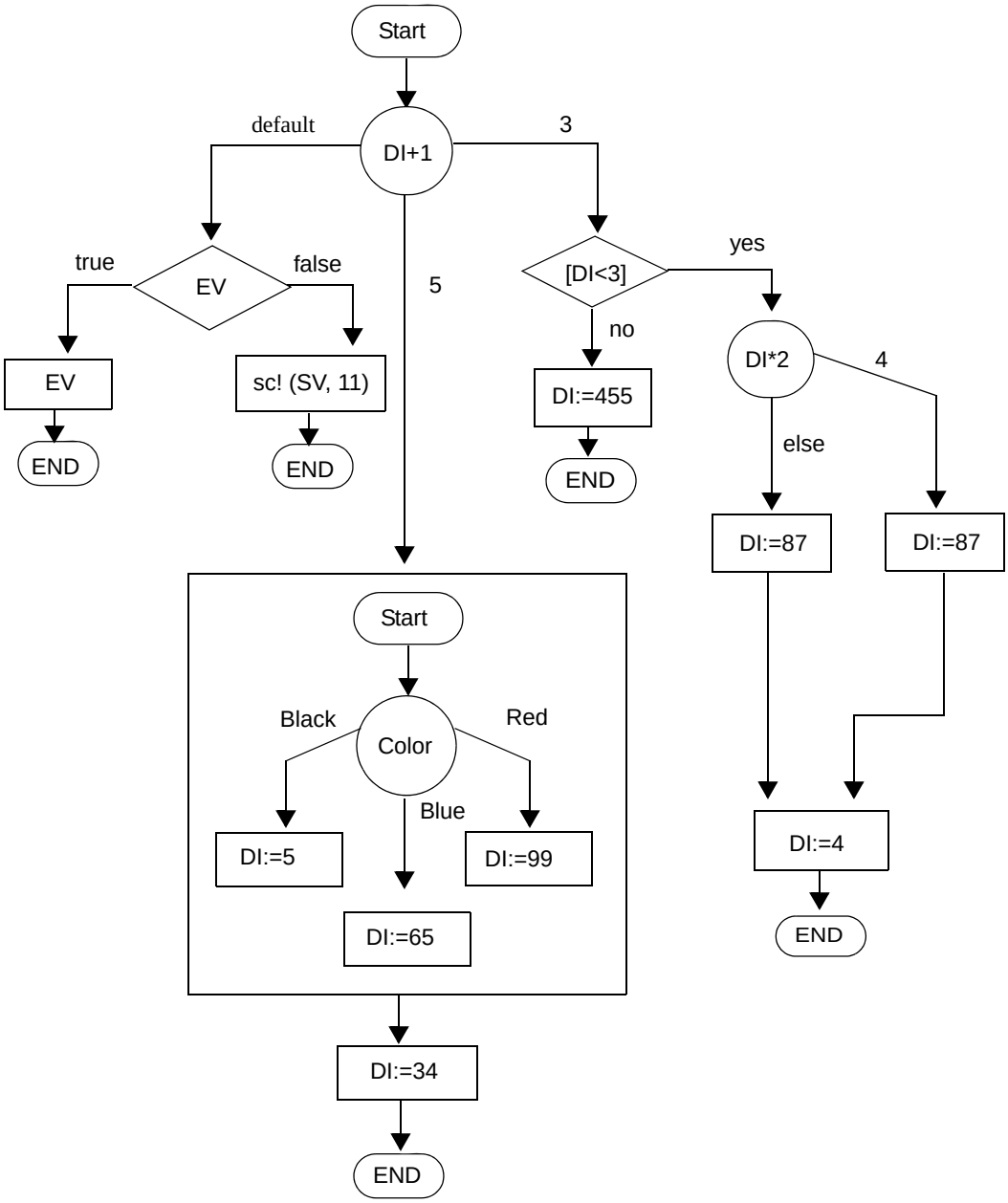

```
void
cgDo_FL_CH_TEST_3()
{
DI=FUNC1();
if (DI > 5) {
ACT_2();
}
else {
ACT_1();
}
}
```

Find/Merge Logic




```
void
cgDo_FL_CH_FIND_MERGE_BOX()
{
DI = 1;
if ((DI == 1)) {
if ((DI == 3)) {
DI = 4;
}
DI = 5;
}
else {
DI = 3;
}
DI = 2;
}
```

Switch Control



```
void
cgDo_USE_SWITCH_CTRL()
{
switch(DI + 1) {
case 3:
if ((DI < 3)) {
switch(DI * 2) {
case 4:


DI = 43;
break;
default:
DI = 87;
break;
}
DI = 4;
}
else {
DI = 455;
}
break;
case 5:
{
switch(COLOR) {
case BLACK:
DI = 5;
break;
case BLUE:
DI = 65;
break;
case RED:
DI = 99;
break;
default:
break;
```

```
}
}
DI = 34;
break;
default:
if (EV) {
GENERATE_EVENT(EV);
}
else {
SetRelAlarm(EV_ALARM, 11, 0);
}
break;
}
}
```

Truth Table Implementation

The Truth Table implementation in code is relatively straight-forward from the table itself. The basic code structure might be seen in the example below. For a Truth Table implementing *function F* with *C1* and *C2* input conditions:

DI1	**DI2**	
1	1	**A1**
	2	**A2**
2	3	**A3**

The generated code would be:

```
void F()
{
if(DI1== 1){
if(DI2== 1){
A1;
} else {
if(DI2== 2){
A2;
};
};
} else {
if(DI1 == 2 && DI2== 3){
A3;
}
```

Note

If the Truth Table is being factorized, as in this example, so is the generated code. This results in compact and fast code. It is recommended to factorize the table at the end of the development stage to make modifications easy, while not paying the cost on production.

Mini-Spec Implementation

The implementation of an Activity can be defined using the **Rational Statemate Action Language**. This definition is called a *mini-spec*. The mini-spec definition of an Activity's behavior is entered into the **Data Dictionary**. The mini-spec is then activated when the associated Activity is active and stops when the associated Activity stops.

As mentioned, the mini-spec is defined in the Data Dictionary Editor. The syntax is similar to that used to describe static reactions, i.e. a list of reactions of the form **trigger/action,** separated by a double semicolon (**;;**).

States that have mini-specs are distinguished by a '>' symbol after their chart name (e.g. ALARM>).

Two modes are supported in MicroC: *Reactive* (either Self or Controlled) and *Procedural*. For both modes, the generated code is a relatively straight-forward implementation of the Mini- Spec itself. The basic code structure might be seen in the examples below.

Reactive Activities

The syntax for reactive mini-spec is "E[C]/A," that is on the event E, when the condition C is true, do the reaction A. Consider the following example, for a mini-spec defined as:

```
ALARM_ON[WORKING]/SET_SIGNAL


With Event - ALARM_ON, Condition – WORKING, and Event -
SET_SIGNAL
The code would be:
if(ALARM_ON && WORKING){
GENERATE_EVENT(SET_SIGNAL);
};
```

Note

- For repeating large actions, it is preferable to use a user-defined function.
- For a repeated scenario of activating some action, the preferred style is:

  [C1 or C2]/A0

  rather than

  [C1]/A0;;[C2]/A0

Procedural Activities

The syntax for procedural mini-spec is comparable to that of the trigger part, without the action. For example, `if( C ) then A endif while(something) do anotherthing end` and so on.

Consider the following example, for a mini-spec defined as:

```
if(WORKING) then SET_SIGNAL endif
The generated code would be:
if(WORKING){
GENERATE_EVENT(SET_SIGNAL);
};
```

ANSI C Code Usage

Only two programming languages are available in MicroC:

- ANSI C
- Assembly Language

ANSI C, includes both C language code (with possible extensions to the ANSII standard) and Assembly language code.

It is best to use the old safe way to link with legacy code (i.e. call the OS/environment special services) and to utilize otherwise inaccessible functionality as inline assembly language calls.

One way to use legacy/library code that is available external to the model, might be done through a user-defined C function calling them. Another way would be to include the definition within the model, in either C or ASM languages.

Lookup Table Implementation

The MicroC Style Guide implementation of the Rational Statemate Language has been extended to include Lookup Tables.

The language supports non-linear “Y=F(X)” functions that are so common in the world of micros. Typically, these non-linear functions are used to represent characteristic curves of valves in a table structure. Such a table may consist of a list of pairs of digitizing points, **Xi**, and its corresponding value, **Fi**. The data might be imported from any ASCII data file. A choice is given whether to perform (linear) interpolation between points, or to use a histogram like mode. In addition, saturation values might be defined, for the upper and lower range bounds, as well as a search order to support performance sensitive scenarios.

For example, consider the following definition and implementation of such a function with return value defined to be “Real” and input defined to be “Integer”:

In “Interpolation,” High to Low mode, Lower Bound=0, Upper Bound =4 The following code will be generated.

X	F(X)
1	1
10	2
100	3
1000	4

Note

Define default mapping between Real and either “`double`” or “`float`” and `Integer` vs. `int8/16/32`.

```
double LOOKUP1(int IN1)
{
/*
Interpolation Function:
if(In < X2 && In >= X1)
Out = (Y2-Y1)/(X2-X1)*(In-X1)+Y1
*/
double LOOKUP1_retval;
if(IN1 < 1)
LOOKUP1_retval = (0);
else if(IN1 >= 1000)
LOOKUP1_retval = (4);
else if(IN1 >= 100)
LOOKUP1_retval = (4 - 3)/((double)1000 - 100)*(IN1 -
100) + 3;
else if(IN1 >= 10)
LOOKUP1_retval = (3 - 2)/((double)100 - 10)*(IN1 -
10) + 2;
else if(IN1 >= 1)
LOOKUP1_retval = (2 - 1)/((double)10 - 1)*(IN1 - 1)
+ 1;
return(LOOKUP1_retval);
}
```

Rational Statemate Action Language Implementation

This kind of programming language can be used where a function is needed in an application. It is the preferred language to use, rather then plan C code, as all the expressions are parsed. Thus, by using the Dictionary tool, it is possible to define relevant properties of the elements used. As such, compatibility between different targets can be more easily achieved since MicroC generates the right expressions in each target environment. This cannot be done if the function is already defined in C code.

The available operators and syntax in the Rational Statemate Action Language is similar to a programming language. As such, it is easily learned and readable.

A detailed discussion of the Rational Statemate Action Language is beyond the scope of this publication.

Integration with the Target

In a MicroC model you can have direct access to ports, memory mapped I/O and external memory (i.e. external to the modeled feature). This can be done using *Data-Items* and *Conditions* bound to *External Symbols*.

This feature is normally used during code generation for H/W (i.e. hardware) properties, using the symbol mapping definition menu. The binding is done in 2 stages:

- **Stage 1** - Data Item/Condition tagged as bounded to external symbol, identifying "Logical" signal name.
- **Stage 2** - Mapping of the "Logical" signal names to "Physical" signals.

For example:

DI1 is defined as bound to **PORTA.**

- **Stage 1** - From the model via the Data Dictionary, **DI1** is tagged as being bound to an external symbol, identified by the *Logical* signal name **PORTA**.
- **Stage 2**: From the compilation profile editor, a mapping of the Logical signal name **PORTA** to a physical location value of 0x03 is accomplished using the data matrix tool.

  Result: On this target, DI1 is now mapped to physical address 0x03.

Data-Items and Conditions might be bound in two modes:

- **Direct—**The previous description holds.
- **Buffered—**In buffered access mode, two additional definitions are used for the element:
  - "**Get Value Call**"—A user-defined API function to receive a value
  - "**Set Value Call**"—A user-defined API function to set a value

When in buffered mode, the internal value is kept. At the beginning of a step, of the correspondence TASK, (Defined in the Condition dictionary as "Its Task") a call to the "Get Value Call" API is done to ask for the new value. The "Set Value Call" API is called whenever there is assignment to the corresponding element in the model.

Instrumentation for Testing and Debugging

Design-level debugging is supported through a combination of various instrumentation inserted into the generated code. This instrumentation typically consists of code that calls external functions (i.e. APIs) and source-level libraries implementing those functions.

Localization of these instrumentations might be done either by predefining the instrumentation calls or by modifying the provided API implementations.

Design-level debugging features include:

- GBA
- Panels (Only available when running on Windows)
- Trace – time stamps
- Trace – State transitions (reportState function)
- Test Driver

GBA: Graphical Back Animation

GBA provides a form of animation to indicate the execution of the application code under test. In the Activity charts, the active boxes are highlighted to indicate their execution. In the Statecharts, the current state of the application is highlighted.

Note

As only 1 box can be executing at a time with one processor, the designer can see if all the code is being serviced.

GBA is supported in two modes:

- Direct
- Indirect.

Direct Mode GBA

- **Asynchronous:** Buffers changes, and then uses a task to pass data to MicroC.
- **Synchronous:** Passes data directly as the code runs.

Indirect Mode GBA

Uses calls in the code to communicate to the target debugger. The target debugger talks to the GBA server to animate the model.

Panels

A task in the code drives the panels. Buffers data and only when active does it drive the panels (OSEK only). It is a basic cyclic task with a high priority, it will slow down the execution of the model. The cyclic rate and priority can be changed to alter this.

Trace (Time Stamp)

Consider the following code:

```
#ifdef TRACE_TASK
#ifndef TRACE_TASK_STARTED
extern void traceTask();
#define TRACE_TASK_STARTED(t) traceTask((t),'S')
#endif
TASK (MAIN_LOOP)
{
if ((cgGlobalFlags & ALARM_SET_MAIN_LOOP) == 0){
cgGlobalFlags |= ALARM_SET_MAIN_LOOP;
SetRelAlarm(ALARM_SET_MAIN_LOOP, 10, 10);
};
TRACE_TASK_STARTED(MAIN_LOOP);
do {
:
} while ( (cgGlobalFlags & BITSUPERSTEP_MAIN_LOOP) !=
0);
TRACE_TASK_TERMINATED(MAIN_LOOP);
TerminateTask();
```

This is from the traceFunc.c file and can be easily modified to send the output anywhere.

```
#ifdef TRACE_TASK
void
traceTask(TaskRefType t, char indx)
{
TickType sysTime;
GetCounterValue(SYS_TIMER, &sysTime);
OSPrintf("%c Task ID %d %ld\n", indx, (int) t, (long
int)sysTime);
}
#endif
```

Trace Tasks

When this flag set, the generated code is instrumented to call the task tracing function in the following places:

- Just after entering the Task frame
- Just before calling the “TerminateTask” API, and leaving the Task frame Instrumentation for Testing and Debugging

Extended Tasks

Applies to EXTENDED Tasks only:

- Just before calling “WaitEvent” API
- Just after leaving “WaitEvent” API

Design Level Debugging: Trace

```
TASK (MAIN_LOOP)
{
if ((cgGlobalFlags & ALARM_SET_MAIN_LOOP) == 0){
cgGlobalFlags |= ALARM_SET_MAIN_LOOP;
SetRelAlarm(ALARM_SET_MAIN_LOOP, 10, 10);
};
TRACE_TASK_STARTED(MAIN_LOOP);
do {
:
} while ( (cgGlobalFlags & BITSUPERSTEP_MAIN_LOOP) !=
0);
TRACE_TASK_TERMINATED(MAIN_LOOP);
TerminateTask();
}
```

Trace ISR

When this flag set, the generated code is instrumented to call the ISR tracing function in the following places (For ISR Type 2 and 3):

- Just after calling the “EnterISR” API
- Just before calling the “LeaveISR” API

Example:

```
#ifndef TRACE_ISR_ENTER
extern void traceIsr();
#define TRACE_ISR_ENTER(i) traceIsr((i), 'N')
#endif
```

Debug Options: Trace State Transitions (reportState function)

This is from the traceFunc.c file and can be easily modified to send the output anywhere.

```
ISR (MY_INTERRUPT)
{
EnterISR();
TRACE_ISR_ENTER(1);
COUNTERdi = 0;
TRACE_ISR_LEAVE(1);
LeaveISR();
}
#ifdef TRACE_ISR
void
traceIsr(int isrNo, char indx)
{
TickType sysTime;
GetCounterValue(SYS_TIMER, &sysTime);
OSPrintf("%c ISR No %d %ld\n", indx, isrNo, (long
int)sysTime);
}
#endif
```

Debug Options: Trace State Transition (reportState function)

The debug option inserts calls to the "`reportState`" functions. The “`reportState`” functions are placed in a file *p_state.c* and are called according to the defined “Debug Level”:

Debug Level1

The reportState functions are called when a Task enters a stable state mode.

Debug Level2:

The reportState functions are called after each step in the statecharts.

For example, consider the following file extract from *P_STATE.C*:

```
cgReportState(unsigned char whichChart, unsigned char*
baseAddress)
{
if(whichChart == 1){
OSPrintf("Statechart %s ", "INIT_MODE_SC");
OSPrintf("In State: ");
if ((((*(StateInfo_INIT_MODE_SC*)baseAddress)) & 0)
== 0) {
OSPrintf("%s", "Chart_INIT_MODE_SC");
OSPrintf(".");
if ((((*(StateInfo_INIT_MODE_SC*)baseAddress)) &
3) == 3) {
OSPrintf("%s", "DefaultOf_Chart_INIT_MODE_SC");
}
else if ((((*(StateInfo_INIT_MODE_SC*)baseAddress))
& 3) == 2) {
OSPrintf("%s", "VOLT_OUT_OF_RANGEst3");
}
else if ((((*(StateInfo_INIT_MODE_SC*)baseAddress))
& 3) == 1) {
OSPrintf("%s", "STATE_15");
OSPrintf(".");
```

Note

OSPrintf is used only in OSEK applications, and outputs to the stdout, which can be redirected. Redefining the OSPrintf function can allow the debug info to be directed anywhere. For Non-OSEK applications, printf is used.

Test Driver

The test driver supports testing of the application using test vectors. Test vectors can be used to drive inputs as well as to record outputs. The generated code calls an API that is provided in the `tst_drv.c` source-level library.

Two execution modes are supported:

- Synchronous
- Asynchronous

Synchronous Execution Mode

The test driver functions are called directly from the code tasks and the data is then streamed to the necessary output.

Asynchronous Execution Mode

A task is created to stream all the data, which is buffered, to the relevant place. The task has a high priority and can be set as either Basic or Extended (For OSEK Applications only). All the information regarding how the task calls the test driver code is in “glob_func.c” file and the test driver code is in “tst_drv.c.”

Redirecting the Output

The test driver utility is fully automated for Windows. This can be enabled via the setting menu in the code profile window. All IO on the panels can be used as automated IO. Normally the data is streamed to a DOS box. To redirect the output, set the following environment variables to the file names that the data should be streamed to, or read from:

- TESTDRIVER_INPUT_FROM_PNL_FILE
- TESTDRIVER_IN_FILE
- TESTDRIVER_OUT_FILE

One way to do this is to create a small batch file to run before running the model.

Retargeting the Test Driver

The full API for the test driver is provided. To enable test drivers to operate on a target, the input and output needs to be redirected. This requires that the `tst_dvr.c` file be modified appropriately so that, perhaps, the I/O is transmitted via a serial communication link. The actual modification to `tst_dvr.c` is a function of the application environment and whose implementation must be left to the developer to determine

Specifics of Statechart Implementation

The calculation for the size of The *StateInfo* data type will be named *StateInfo_<Controller Name>* - for example, might be `StateInfo_A11_CTRLcnt1`. The data type will be defined as an unsigned type of either 8, 16 or 32 bits. The size depends on the topology of the Statechart. A general method for calculating the required size is:

1. Summarize the bits required for each level in the state hierarchy.
2. To calculate the bits required for each level in the state hierarchy, take the maximal number of states in that level, add one and calculate how many bits are required to count, in binary, to that number:

```
(log 2 (number_states))
```

Note

For And-States, perform the calculation for each of the And-state descendants, and take the largest.

Generated Data Types, Data Usage, and Functions

All of the variables, data types and functions that are generated in MicroC are directly derived from the model.

Some are directly user-defined data (data-items, conditions, events, user-functions) and some relate to the graphical elements, like states and activities.

Those that the tool generates, and the naming convention used is customizable, through the OSDT (MicroC OS Definition Tool), are marked with **custom**.

When having `<NAME>` the intention is to replace that sequence with the relevant model element name.

Data is generated to the (custom) `glob_dat.c` file. The variables which are not in that file are context variables, that are generated as automatic variables for the activity/statechart/ flowchart they are used in. Functions resulting from the graphical model are generated in `<MODULE>.c` file (that is the module name, in the compilation profile, containing the chart in scope).

User functions and other functions needed, that are not explicitly in any module scope, are generated to (custom)

Data Types

```
custom: cgSingleBufferType_<NAME>
```

```
custom: cgDoubleBufferType_<NAME>
```

Those are type definitions (typedefs) for structures, with the activity name as postfix. The data assigned to the activity, in the Its Task field will be located in that structure, and instantiate later as either `cgSingleBuffer_<NAME>`, for all of the non doublebuffered data elements, or `cgDoubleBufferNew_<NAME>` and `cgDoubleBufferOld_<NAME>` for all of the doublebuffered data elements.

```
custom: StateInfo_<NAME>
```

This is type definitions (typedefs) for int8/16/32, to hold the internal state configuration of a Statechart.

User Data

```
custom: cg_Events
```

This data (either int8/16/32) holds the events and the derived events (such as ch/fs/tr as well as tm/dly) related to a certain activity. When more then a single variable is needed, the tool will add indexed postfix like “cg_Events1,” “cg_Events2,” etc.

```
custom: cg_BitsConditions
```

This data (either int8/16/32) holds the conditions related to a certain activity. When more then a single variable is needed, the tool will add indexed postfixes.

Data Supporting Statechart Generation

```
currentState_<NAME>
```

```
nextState_<NAME>
```

This data hold the internal (in current step and in the next step) state configuration of a Statechart. Is required per control activity implemented by a statechart.

```
staySame_<NAME>
```

This data hold the internal state configuration in which no change occurred between current and next step. Is required only when the Enter State/Exit State functions are required, per control activity implemented by a statechart.

Functions Supporting Statechart Generation

```
custom: cgEnterActions_<NAME>
```

```
custom: cgExitActions_<NAME>
```

Those void functions will be generated in case of a control activity, with statechart underneath, that has (in accordance) entering/exiting reactions.

Note that when the optimizer is enabled, some of the reactions might be placed directly on transitions code, thus avoiding the need of having those functions.

```
custom: cgDo_<NAME>
```

That void function will be generated for each control activity, and contains the code implementing the logic.

Data Supporting Activity Chart Generation

```
custom: cgGlobalFlags
```

For task containing Statechart underneath. Indicates when a task is in non-stable state, i.e., need to perform another step. Might be 8/16/32 bits, according to the no. of tasks having a Statechart underneath. And Indicates active/ inactive activities might be 8/16/32 bits, according to the no. of activities requiring active bit.

This is if either the activity is procedural or its parent has a control-activity, or it is flagged as “Guarded Activation” (=”yes”).

```
custom: cgSingleBuffer_<NAME>
```

When activity has non double-buffered data (user data, conditions) associated with.

```
custom: cgDoubleBufferNew_<NAME>
```

```
custom: cgDoubleBufferOld_<NAME>
```

When activity has a double-buffered data type (user data, conditions, events) associated with it.

Functions Supporting Activity Chart Generation

```
custom: cgActivity_<NAME>
```

That void function contains the implementation for the activity. In case when the implementation is either statechart or flowchart, that function will call the statechart/flowchart code. In case of lookup table, truth table, or other textual implementation, the code will be contained in that function.

Data Supporting Timeout/Delay Implementation

```
custom: cgTimeoutsMask
```

Indicates pending timeouts/delays. Might be 8/16/32 bits, bits. That is less then or equal to the number of timeout/delay in the model.

```
custom: currentTick
```

TickType variable, as defined in the compilation profile. Will be used when having delay or timeout in the model.

Functions Supporting Timeout/Delay Implementation

```
custom: genTmEvent_<NAME>
```

The `void` function will be generated for each control activity having a background timeout or delay. The function checks to see if the time has expired.

Data Supporting Instrumentation Implementation

GBA

There are two important arrays of type "**unsigned char**," named "`gba_states`" and "`gba_acts`" in "`glob_dat.c`". The "`gba_states`" array will hold 1 bit per each state in the application. The "`gba_acts`" array will hold 2 bits per each activity in the application. Those bits are packed together into the 8 bits (char) chunks.

Panels

```
static char *panels_table[]
```

This array of char* holds the panels in scope.

```
struct
       PanelBindings_PreviousValues
       panelBindings_PreviousValues
```

The variable `panelBindings_PreviousValues` holds the previously reported values of elements that are bound to panels.

Test Driver

```
struct
       TestDriver_PreviousValues
       testDriver_PreviousValues
```

The variable `testDriver_PreviousValues` hold the previously reported values of elements that are bound to panels, and are reported to the test driver API.

Functions Supporting Instrumentation Implementation

GBA

```
cgColorState_<NAME>
```

That void function will be generated for each activity, to build the corresponding data to be used to highlight states.

```
<NAME>_CB
```

When either of the “Panels” or “Test Driver” is enabled, setting functions will be generated. Those functions will be generated in “`glob_func.c`” file, for each of the model elements that is bound in either “`Input`” or “`Input/Output`” mode.

Panels

```
void init_panels(void)
```

```
void update_panels(void)
```

Each of these `void` functions will be generated when having panels in scope. `init_panels` is called once at startup, to initialize the panels. `update_panels` is called periodically to update and refresh the panels.

Test Driver

```
void init_test_driver_table (void)
```

```
void call_test_driver (void)
```

Each of those void functions will be generated when having panels in scope and the “Test Driver” is enabled. “`init_test_driver_table`” is called once at startup, to initialize the test driver. “`call_test_driver`” is called periodically to update the test driver, i.e., to report changes and poll inputs.

Debug

`cgReportState`

This function will be generated in (custom) `p_state.c` file when `debug` is enabled. The function calculates and reports the current state configuration of an activity chart. Because the function is generated to a separate file, and the prototype of it is:

```
void cgReportState(unsigned char whichChart,
unsigned char* baseAddress)
```

It is possible to use that functionality across target-host communications when providing a target implementation that saves the data to a file and uses that generated file on the host.

`cgReportStates_<NAME>`

This function will be generated when debug is enabled per control activity, to the `<MODULE>.c` file. That function call `cgReportState` with the appropriate data and timing.

OSDT Naming Styles

The following sections describe the naming styles of OSDT models and variables.

Model Names

The Code Style page includes the page *Model Data - Naming Style,* with two API definitions:

- Model Data Prefix()
- Model Data Postfix()

The prefix or postfix strings are added to the name of global model data elements for which the field `Its Task` is `global`.They are added just before or after the element's model name in the generated code. When in `Case Sensitive` mode, the case-correct name is used as the element model name (`nameid`). The definitions can use attributes of the model object.

Variable Names

The Variables Naming Style tab includes three new definitions for customization of the variable names used for statecharts:

- Current State Info Variable Prefix - Specifies the prefix to use for the `currentState` variable.
- `Stay Same State Info Variable Prefix -` Specifies the prefix to use for the `staySame` variable.
- `Next State Info Variable Prefix -` Specifies the prefix to use for the `nextState` variable.

New Function Call

The file `type_def.h` no longer includes the `CALL_` macro. The prototype of the function did not change, but the call did:

- If the parameter is `OUT/INOUT`, MicroC adds an ampersand (&) before it.
- If the parameter is a string or an array, an ampersand is not added.
- If there is an ampersand in the parameter, the parameter is enclosed with parentheses. For example:

```
&(PRM)
```

Examples

Prototype:

```
void myFunc(int inPrm0,
int * outPrm1, int * inOutPrm2);
```

Call:

```
myFunc(INPRM, &(OUTPRM), &(INOUTPRM));
```

Prototype:

```
void myFuncStr(char * inPrmStr,
char * outPrmStr, char * inOutPrmStr);
Call: myFuncStr(PRMSTR, PRMSTR, PRMSTR);
```

Linking Generated Code with External Data Types

This section describes how to link the MicroC generated code with external functions and data types.

External User-Defined Subroutines

You can have an external user subroutine in your MicroC model. To define a subroutine to be external, set the *Selected Implementation* to *External Code/None*. In this case, MicroC generates only the call to the subroutine—not the prototype or body.

When a user-defined function is not defined in the model (that is, unresolved text) the code generator does not generate a prototype for that function. To generate an external user defined function prototype in the model, complete the following steps:

1. Save the user-defined function with the relevant return type and arguments list.
2. Define a dummy implementation for that function. The implementation cannot empty; otherwise, MicroC aborts code generation.
3. Set the design attribute for the user-defined function “External Function” to “yes.” The OSI's `mainloop_sc` and `mainloop_sc_ext` has that attribute for functions.

External Data Types

Micro includes a design attributes file for User-defined Data Types (UDTs), which is named *UserDefinedType.dat*. A UDT that has its `Data Type` attribute defined as `CK_itsDataType` generates a `typedef` statement in the code, which is then used to define variables of that type.

Exceptions:

- Singleton-Record and Singleton-Union

  For these types, the name of the UDT (the value of the `CK_itsDataType` attribute) is used to define the variable.

- Variables that override the value of `CK_itsDataType`.

To use an external data type in the model, set the **Data Type** field in the *Design Attribute of a User-Defined-Type (UDT)* to be a value other than "**Default**." To have design attributes for a UDT in an OSI, complete the following steps:

1. Open the OSI from the OSDTool.
2. 2. In the Edit Attributes dialog box, select the list item *UserDefinedType*.
3. 3. Click OK and save the OSI.

The design attributes for the UDT will be available from the Data Dictionary.

Fixed-Point Variable Support

This section describes fixed-point support for integer arithmetic, which scales integer variables so they can represent non-integral values (fractions). This functionality enables you to perform calculations involving fractions without the need of special floating-point support from the target.

MicroC supports fixed-point arithmetic in the model level, through the Dictionary and the Check Model tools, as well as in the generated code.

Implementation Method

MicroC uses the “2 factorials” implementation method— redefining the least significant bit (LSB) to represent zero, or the negative power of 2. This implementation method provides reasonable code size and run-time performance, but is not the most accurate method.

Consider the binary 8-bit value 0b00010001. Usually, the value represented here is “17” because:

- The LSB (1st bit) corresponds to 2^0 (1).
- The 5th bit corresponds to 2^4 (16). Rescaling that value to begin at 2^{-3} gives:

  $2.125 = 1*2^{-3}$ (or 0.125) + $1*2^1$ (or 2)

The parameter required here is the power (of 2) represented by the LSB. This is also the resolution.

Supported Operators

You can use the following operators with fixed-point variables:

- Arithmetic (+, –, *, /)
- Assignment (=)
- Comparison (<, >, <=, >=, ==, !=)
- Functions (return value, parameters, local variables)

Evaluating the wordSize and shift of an Object

The `wordSize` and `shift` of an object are defined by its attributes (specified in the Data Dictionary Editor). MicroC determines the `wordSize` and `shift` of an expression made of objects and operators using the formulas listed in the **Fixed-Point Macros Macro Definition Description** table.

The conventions used in the table are as follows:

- **WS—**The `wordSize` of the object
- **SH—**The `shift` of the object
- **RG—**The range (`wordSize – shift`)
- **MAX(A, B)—**`A>B:A:B`
- **SUM(A, B)—**`A+B`
- **SUB(A, B)—**`A-B:`

Operator or Object	Formula Used
=	`wordSize` and `shift` of the left operand
*****	`WS=SUM(MAX(RG1, RG2), SUM(SH1, SH2)), SH=SUM(SH1, SH2)`
/	WS=SUM(MAX(RG1, RG2), SUB(SH1, SH2)), SH=SUB(SH1, SH2)
funcCall	`wordSize` and `shift` of the left function
ActualParameter	Converted to the FXP type of the FormalParameter
All Other Parameters	All other parameters `WS = SUM(MAX(RG1, RG2), MAX(SH1,` SH2)), SH=MAX(SH1, SH2)

If the `wordSize` is greater than 32 bits, MicroC displays the following messages:

- **wrn_err.inf** - Warning: Fixed-Point Overflow in Expression:<Expression>
- **generated code** - `/* Warning - Fixed- Point Overflow in Expression. */`

  This message is located right after the expression.

When you use fixed-point variables in integer arithmetic, MicroC uses the special functions (or C macros) provided in the `FXP` package to perform the calculations. The following table lists these macros.

Fixed-Point Macros Macro Definition Description

Macro Definition	Description
FXP2INT (FPvalue, FPshift) (FPvalue >> FPshift)	Converts a fixed-point number with `shift=FPshift` to an integer.
LS_FXP2FXP8 (FPvalue, fromFPshift, toFPshift) ((sint8(FPvalue)) << ((toFPshift) - (fromFPshift)))	Converts a fixed-point number with `shift=fromFPshift` to an 8-bit fixed-point number with `shift=toFPshift` using left shifting
RS_FXP2FXP8 (FPvalue, fromFPshift, toFPshift) ((sint8(FPvalue)) >> ((fromFPshift) - (toFPshift)))	Converts a fixed-point number with `shift=fromFPshift` to an 8-bit fixed-point number with `shift=toFPshift` using right shifting
LS_FXP2FXP16 (FPvalue, fromFPshift, toFPshift) ((sint16(FPvalue)) << ((toFPshift) - (fromFPshift)))	Converts a fixed-point number with `shift=fromFPshift` to a 16-bit fixed-point number with `shift=toFPshift` by using left shifting
RS_FXP2FXP16 (FPvalue, fromFPshift, toFPshift) ((sint16(FPvalue)) >> ((fromFPshift) - (toFPshift)))	Converts a fixed-point number with `shift=fromFPshift` to a 16-bit fixed-point number with `shift=toFPshift` right shifting
LS_FXP2FXP32 (FPvalue, fromFPshift, toFPshift) ((sint32(FPvalue)) << ((toFPshift) - (fromFPshift)))	Converts a fixed-point number with `shift=fromFPshift` to a 32-bit fixed-point number with `shift=toFPshift` using left shifting
RS_FXP2FXP32 (FPvalue, fromFPshift, toFPshift) ((sint32(FPvalue)) >> ((fromFPshift) - (toFPshift)))	Converts a fixed-point number with `shift=fromFPshift` to a 32-bit fixed-point number with `shift=toFPshift` using right shifting

Unsupported Functionality

The following functionality is not supported:

- FXP parameter passed by reference

  MicroC generates the following error message:

  Error: Unsupported usage of Fixed-Point parameter used by reference.

  In function: <FUNC_NAME> Parameter number: <PARAM_NUM>.

- MicroC ignores the remainder in division operations that result in remainders.

  For example:

  FXP1(WS=8, SH=2) = 5
  FXP2(WS=8, SH=2) = 2
  FXP1/FXP2 = 2 (not 2.5)

Specifying Fixed-Point Variables

The following sections describe how to specify fixed-point variables in MicroC.

The Code Generator

To specify fixed-point variables in the Code Generator, complete the following steps:

1. Select **Compilation Profile->Setting-> Target Properties**.
2. Click **Use Fixed Point variables for "Real"**.
3. Select the default word size (8/[16]/32) and LSB= 2^-([0],1,2,..*n*).

The Generated Code

Fixed-point variables are implemented using `uint` variables (`sint8, sint16, sint32`), with hardcoded shift values. MicroC allocates data according to the `wordSize` of the variable.

wordSize	Data Type
8 bits	sint8
16 bits	sint16
32 bits	sint32

All calls to functions or expressions requiring integer values are done through an `FXP-to-int` cast, including the test-driver/ panel driver. Specifically, the operators “ROUND” and “TRUNC” are called with an `FXP-to-int` cast.

For example, given a fixed-point variable `fxp_var`, an integer variable `int_var`, and the following actions:

```
INT_VAR := FXP_VAR + 4;
FXP_VAR := INT_VAR/5;
```

The generated code is as follows, if you specify fixed-point mode:

```
INT_VAR = RS_FXP2FXP16(FXP_VAR + LS_FXP2FXP16(0x4,
0, FXP_VAR_FXP_SHIFT), FXP_VAR_FXP_SHIFT, 0);
FXP_VAR = LS_FXP2FXP16(INT_VAR / 0x5, 0,
FXP_VAR_FXP_SHIFT);
```

OSI Definition ToolAPI Syntax Definition

Each OSI (Operating System Implementation) contains a predefined list of API definitions. Each such API definition is intend to define the structure of the generated code that will result from use of that API. An API definition is often referred to as an OSI. OSIs are managed using the OSDT (Operating System Definition Tool) utility that is supplied with MicroC.

Each API definition might have predefined values that it would use, similar to the formal parameters of a function. To use predefined values in the API definition, you must use the predefined value name wrapped with the "$<" prefix and ">" postfix delimiters. Note that this notation is standard throughout the OSDT utility.

For example, consider the following API definition:

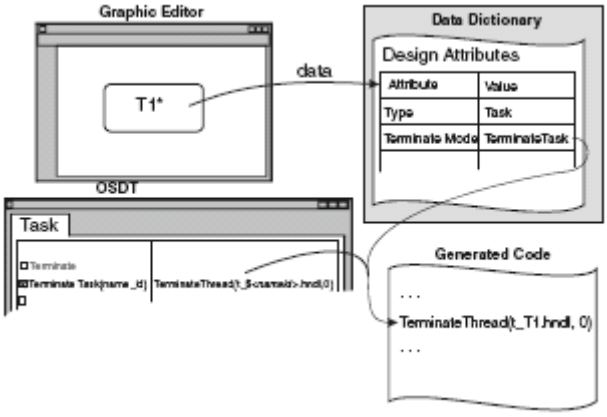
```
API Name Terminate Task(nameid),
API definition TerminateThread (t_$<nameid>.hndl, 0);
```

The resulting generated code, for a Task named "T1," will be:

```
TerminateThread (t_T1.hndl, 0);
```

This is illustrated in the following figure.

Parameterizing an API Definition, Method 1



Another way to parameterize the API definition is to use the property value of the element itself, as defined in the **Data Dictionary** for it. For example, suppose the element has a design attribute named *Create Mode* that uses the attribute key word *CK_createdMode*, which then evaluates to:

```
CREATE_SUSPENDED
```

For the following API definition:

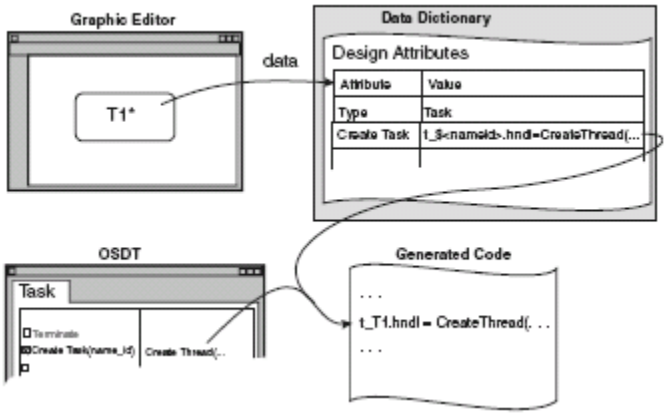
```
API Name Create Task(nameid)


API definition t_$<nameid>. hndl = CreateThread ( NULL ,
0 , ( LPTHREAD_START_ROUTINE )$<nameid> , NULL ,
$<CK_createdMode> , &t_$<nameid>.tid );
```

The resulting generated code, for a Task named “T1,” will be:

```
t_T1. hndl = CreateThread ( NULL , 0 ,
( LPTHREAD_START_ROUTINE ) T1, NULL ,
CREATE_SUSPENDED, &t_T1.tid );
T1*
```

Parameterizing an API Definition, Method 2



A third way to parameterize the API definition is to use the *property value* of the element itself (as defined in the DataDictionary for it) as the API definition itself. For example, suppose the element has a design attribute, possibly hidden, that uses the attribute key word:

```
CK_sendMessagesAPI
```

This evaluates to:

```
mySendMessage($<nameid>,...)
```

For the following API definition:

```
API Name Send Message(nameid)
API definition $<<CK_sendMessagesAPI>>
```

And design attribute definition:

```
mySendMessage($<nameid>, $<CK_MessagePriority>);
```

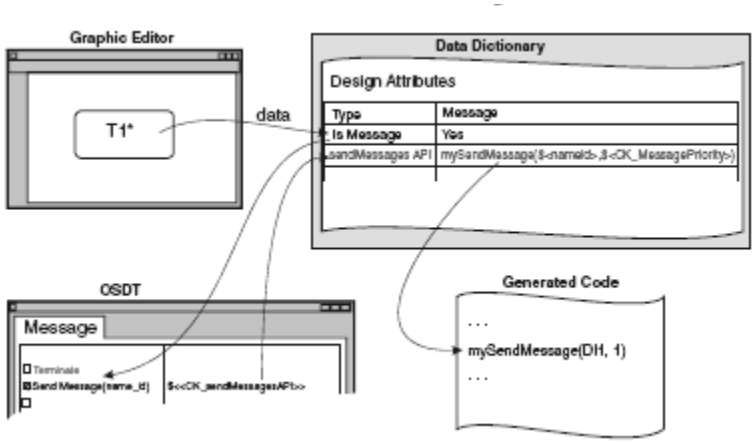
The resulting generated code, for a data item named *DI1*, will be:

```
mySendMessage(DI1, 1);
```

assuming that the `CK_MessagePriority` property evaluates to **1**.

This third method is illustrated in the following figure.

Parameterizing an API Definition, Method 3



Note

A good use for this approach is to modify the API definition just for a local implementation requirement.

Note

Browsing of the defined key word, in the data dictionary, is supported in the tool. When in the definition field of the API, you can enter either of the following sequences, which will result in a list popping up with the predefined keys. The sequences are:

"**$<**" and "**$<<**". While in this list, pressing space, enter or "**>**" will close it.

Conditional Expressions

In addition to the **$<** and **$<<** expressions discussed above, other conditional expressions are supported. These are described below.

The general syntax is `?<Conditional Operator>` where `Conditional Operator` can be one of the following:

?<begin>	Marks beginning of a conditional expression
?<end>	Marks end of a conditional expression
?<?>	
?<:>	
?<&&>	Logical AND
?<||>	Logical OR
?<==>	Equal strings
?<!=>	Not equal strings

Example 1

?<begin> $<prop1> ?<==> prop1val ?<?> expression when yes ?<:> expression when no ?<end>

In this conditional expression we mimic the C conditional expression, "**? :**" with some syntax modifications.

Syntax

The syntax require that a conditional expression will begin with the operator "**?<begin>**" end with the operator "**?<end>**" and will contain in between the operators: "**?<?>**" and then "**?<:>**".

So, a conditional expression looks like:

```
?<begin> sub expression 1 ?<?> sub
expression 2 ?<:> sub expression 3 ?<end>
```

All that is legal in an API definition might appear before the **?**`<begin>` and after the `?<end>` markers.

Semantics

The segment defined between the ?<begin> ?<end> operators will be replaced by “sub expression 2” when “sub expression 1” evaluates to true, and by “sub expression 3” when “sub expression 1” evaluates to false.

Take another look at example 1:

```
?<begin> $<prop1> ?<==> prop1val ?<?> expression when
yes ?<:> expression when no ?<end>
```

The API line will be *expression when yes* if `$<prop1>` evaluates to *prop1val* and will be *expression when no* otherwise.

Syntax Definition

sub expression 1

This sub expression may be composed of the ?<&&>, ?<||>, ?<==>, ?<!=> binary operators and operands in between.

Note

New-lines and conditional expressions are forbidden here.

Operators definition, with the highest precedence level at the top of the table:

?<==>	Equal strings
?<!=>	Not equal strings
?<&&>	Logical And
?<||>	Logical Or

Expressions that contains neither **?<==>** nor **?<!=>** are evaluated to false.

sub expression 2 and sub expression 3:

These can consist of any legal expression in the API definition, including conditional expressions.

Example 2

```
Some prefix, fix code ?<begin> $<prop1> ?<==> prop1val
?<&&> $<prop1.1> ?<==> prop1.1val ?<?> ?<begin>
$<prop2> ?<==> prop2val ?<||> $<prop2.1> ?<==>
prop2.1val ?<?> exp 1.1 when yes ?<:> exp 1.2 when no
?<end> ?<:> exp 2 when no ?<end> Some postfix code,
then another conditional expression ?<begin> $<prop3>
?<==> prop3val ?<?> exp 3.1 when yes ?<:> exp 3.2 when
no ?<end>
```

In this example we are trying to illustrate a “full” capability expression. Begin with the inner expression:

```
?<begin> $<prop2> ?<==> prop2val ?<||> $<prop2.1> ?<==>
prop2.1val ?<?> exp 1.1 when yes ?<:> exp 1.2 when no
?<end>
```

That expression will be evaluated to “*exp 1.1 when yes*” when either $<prop2> evaluates to “prop2val” or $<prop2.1> evaluates to “prop2.1val”. When none of them is true, it will be evaluated to “*exp 1.2 when no*”.

So that expression will be replaced by either “*exp 1.1 when yes*” or “*exp 1.2 when no*”, let us mark it as *“exp 1”*.

Now, substituting for “exp 1”, the first conditional expression will look like:

```
?<begin> $<prop1> ?<==> prop1val ?<&&> $<prop1.1> ?<==>
prop1.1val ?<?> exp 1 ?<:> exp 2 when no ?<end>
```

That expression will be evaluated to *exp 1* when $<prop1>evaluates to *prop1val* and $<prop1.1> evaluates to *prop1.1val*. Otherwise, the expression will be evaluated to exp 2 when no.

So, assuming that:

```
$<prop1> = prop1val
$<prop1.1> = prop1.1val
$<prop2> = prop2val DIFFER
$<prop2.1> = prop2.1val DIFFER
$<prop3> = prop3val DIFFER
```

The API result will be:

```
Some prefix, fix code exp 1.2 when no Some postfix code, then another
conditional expression exp 3.2 when no
```

mainloop_sc_ext OSI

MicroC includes the OSI `mainloop_sc_ext`, which is an extension to the simple main-loop scheduler (`mainloop_sc`). This extended OSI supports the following functionality:

- Predefined time slices of 8 and 84 milliseconds.
- Segmented memory support—code and data can be mapped.
- Conditional compilation, using `#ifdef` for Activities.

Naming Styles

The following sections describe the tabs that enable you to specify the naming styles of models and variables.

OSDT Model Naming Style

The Code Style tab includes the page *Model Data - Naming Style,* with two API definitions:

- Model Data Prefix()
- Model Data Postfix()

The prefix or postfix strings are added to the name of global model data elements for which the field *Its Task* is *global*.They are added just before or after the element's model name in the generated code. When in *Case Sensitive* mode, the case-correct name is used as the element model name (*nameid*). The definitions can use attributes of the model object.

Naming Style of Variables

The Variables Naming Style tab includes three new definitions for customization of the variable names used for statecharts:

- `Current State Info Variable Prefix -` Specifies the prefix to use for the *currentState* variable.
- `Stay Same State Info Variable Prefix -` Specifies the prefix to use for the *staySame* variable.
- `Next State Info Variable Prefix -` Specifies the prefix to use for the *nextState* variable.

Index