Telelogic
Statemate®
MicroC Code Generator

IBM®

Statemate®

MicroC Code Generator

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 Telelogic Statemate 4.5 and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Statemate's MicroC Code Generator

Statemate's MicroC Code Generator can be used to develop embedded real-time software for micro-controllers. In addition to the generation of the code, tools are provided for debugging and testing the software.

The Generator Overview

The MicroC Code Generator generates compact, readable ANSI C code, based on the model you have designed using Statemate's graphical tools. The graphical elements can be supplemented by linking in user-supplied C and or Assembly code.

The code generator allows you to define a wide range of settings in order to tailor the generated code to your target operating systems and hardware.

This guide contains detailed information regarding:

- The mapping of the various model elements to C code
- Configuring the MicroC Code Generator to output code tailored to your target operating systems and hardware
- Using the MicroC Code Generator to generate code and to then run, debug, and test the compiled code

Basic and Advanced Generator Information

In order to get you up and running quickly, this guide is divided into two sections:

- **The Basics**

  This section describes the basics of using the MicroC Code Generator. For many users, this information will be sufficient for their work.
- **Advanced Topics**

  This section covers more advanced topics, allowing you to further fine-tune the generated code to your target system.

MISRA Compliance

Statemate generates code that is compliant with MISRA Rule 60. This is accomplished when the Check-box **Options > Settings > General > Advanced Options > Generate** are selected to enable the "else" clause after "if...else if..." construct in the generated code that is required for this compliance.

Statemate Block in Rhapsody

MicroC code-generator supports integration of a Statemate model (block) into a Rhapsody model. When generating code for a Statemate block, the Statemate MicroC code generator performs the following operations:

- Generates all code in one single file that includes other required nongenerated files.
- Generates unique code to allow using few Statemate Blocks in a single Rhapsody model.

For more detailed information about this feature, refer to the *Statemate User Guide*.

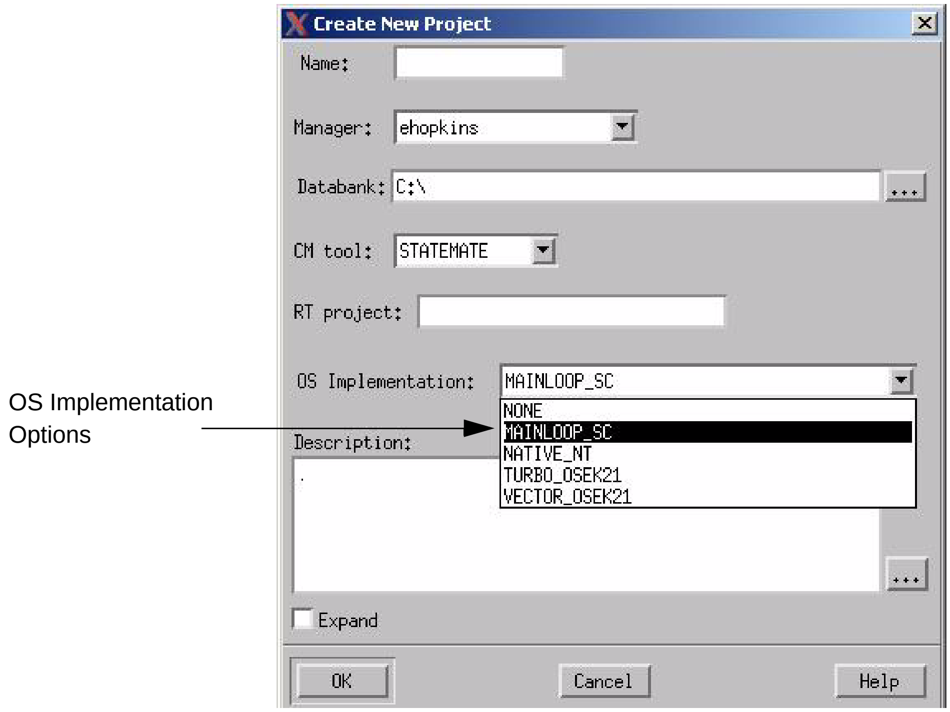
Selecting an OSI for your Statemate Project

When you create a new project in Statemate, you select an *OSI (Operating System Implementation)* for the project. The OSI selected customizes the code generator to produce code that is appropriate for the target operating system.

Statemate contains a standard set of OSIs that can be selected. In addition, these basic OSIs can be modified and saved as a new OSI using the OSDT tool that can optionally be installed with Statemate (see **Advanced: Creating Customized OSIs**).

The list of available OSIs is generated from the content of the CTD directory under Statemate. If the CTD directory contains “customized” OSIs, you will see these additional OSIs in addition to the standard set.

Once you have selected an OSI for a project, you cannot change the OSI used. However, the OSI list also contains an option **None**. This value can be selected temporarily, and you can then select an OSI at a later time. (Note that once you have selected an OSI, you cannot return to **None**.)



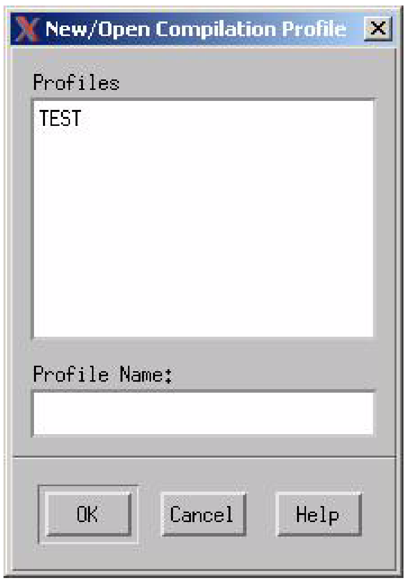
Working with Profiles

While the OSDT allows you to configure generated code for an operating system at a project level, profiles allow you to further configure the generated code for specific hardware targets. Because of the highly-variable hardware configurations available for embedded systems, this ability to fine-tune the code for the planned target hardware is essential. A number of different profiles can be defined and used for generating code in the same project.

Creating a Profile

A profile consists of the charts for which you would like to generate code, as well as the code generation options you would like to use when the code is generated.

Profiles are defined using the Code Generation Profile Editor. To open the profile editor, select **File > New/Open > Profile > Statemate MicroC Code Generator** from the Statemate main menu. The New/Open Compilation dialog box displays.



Setting the Target Configuration

The MicroC Code Generator allows you to modify a large number of settings that affect the generated code.

In addition, the code generator provides you with a list of configurations that have been set up for various hardware targets. When you select one of these configurations, the appropriate values are automatically set for the various code generation options. In many cases, this may be sufficient, and you may not have to make any more changes to the code generation options.

In some cases, however, you may want to further refine the code generation settings by manually changing the settings for certain options. In such cases, it is recommended that you choose one of the target configurations as a starting point, and then modify the specific options that must be changed.

To select a target configuration, select **Options** > **Set Target Configuration** from the MicroC Code Generator's main menu.

After you have selected a target configuration, when you open one of the code generation option tabs, you will see the appropriate values for that target.

Code Generation Options

The code generation options allow you to customize the generated code for your target hardware.

To modify the settings for these code generation options, select **Options** > **Settings** from the menu in the MicroC Code Generator Window.

The code generation options are categorized as follows:

- Target Properties
- Code Instrumentation
- Application Configuration
- OS
- General
- Test Driver
- Optimization

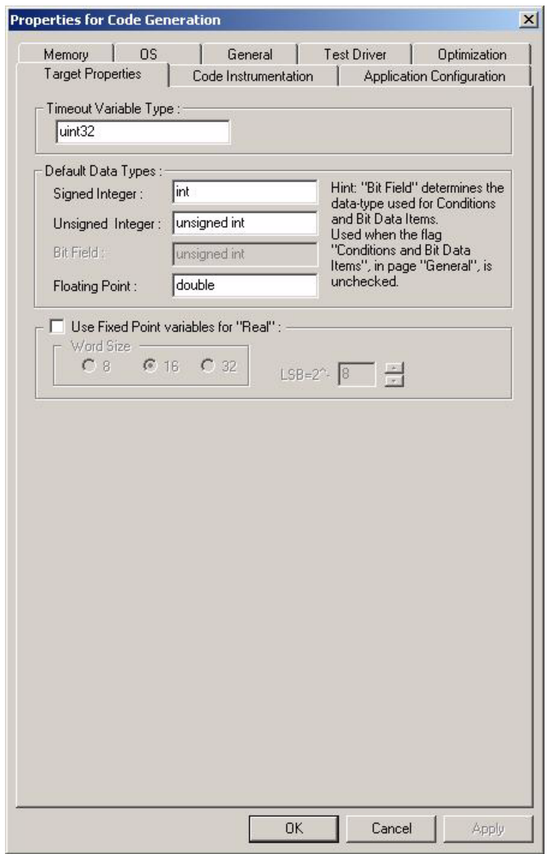
Each of these categories is described in detail in the following sections.

Note

Any changes made to these code generation options affects all of the modules included in the profile.

Target Properties

The following section describes the options available on the **Target Properties** tab.



Memory Structure

- **Word Size**
  - Determines the word size used for bit buffers, such as conditions and events.
  - You can select from among the following values: 8, 16, 32.
  - If a buffer size smaller than the selected word size is sufficient, then the smaller buffer is used.
  - Examples:
    - If there are 3 conditions in the model and the word size selected is 16, then an 8-bit buffer is allocated to hold the 3 conditions.
    - If there are 20 conditions in the model and the word size selected is 16, then two buffers are allocated to hold the 20 conditions—one with 16 bits (for the first 16 conditions) and one with 8 bits (for the remaining 4 conditions).

  Note: The State variable that holds the current state of a control activity does not follow this rule, and will allocate a buffer with sufficient size to hold the State's topology (up to 32 bits).

- **Bit Orientation**
  - Controls the orientation of the bits inside a single byte.
  - You can select one of the following values:
    - LSbit First
    - MSBit First
- **Byte Orientation**
  - Controls the orientation of the bytes inside allocated data larger than a single byte.
  - You can select one of the following values:
    - LSByte First
    - MSByte First

  In addition, you can select the **Use Instrumentation** check box to control the generation of byte orientation directives in the code (`#ifdef LSBYTE_FIRST` directives). If this option is selected, #ifdef directives will be used in the code to accommodate the two byte orientations. This adds flexibility by making the code easier to change manually.

Memory Initialization

- **Reset Global (Internal) Data**

  Selecting this check box enables the global data reset options:

  - **Compile Time, Static**

    Enables initialization of the data in the model using static initialization at the data allocation location.

  - **Run Time, Dynamic**

    Enables initialization of all the data in the model through a call to the macro `RESET_DATA` in the `TASKINIT` function.

    The `RESET_DATA` macro uses the function `memset` which should be defined in the environment. If this function is not defined, you can define the macro `AVOID_MEMSET` and use the function `rimc_mem_set` which is defined in the file <profile-name>.c.

- Reset User Model Data

  Selecting this check box enables the user model data reset options:

  - **Compile Time, Static**

    Enables initialization of the data in the model using static initialization at the data allocation location.

  - **Run Time, Dynamic**

    Enables initialization of all the data in the model through a call to the macro `RESET_DATA` in the `TASKINIT` function.

    The `RESET_DATA` macro uses the function `memset` which should be defined in the environment. If this function is not defined, you can define the macro `AVOID_MEMSET` and use the function `rimc_mem_set` which is defined in the file

Static Initialization of "Frequency-of-Activation" Data

The generated code for static initialization of data related to Frequency of Activation complies with the profile settings.

When setting the Statemate MicroC profile options:

- Reset Global (Internal) Data, to be: RunTime, Dynamic, and
- Reset User Model Data, to be Run Time, Dynamic, the static initialization of the global data: <task_name>_COUNTER_FREQ in the file glob_dat.c will be omitted.

Use “const” Keyword to define Constant Values

If this option is selected, constant elements will be generated with the `const` modifier in the files `glob_data.c and glob_dat.h`, rather than being generated as pre-processor macros in the file `macro_def.h`.

The attribute for Constant elements (Conditions and Data-items) controls whether the specific element should be generated using the “const” keyword. The name of the new design-attribute is “Use 'const' Keyword” with these possible values:

- **no -** The constant will be generated according to the settings of the MicroC code generation option: **Options** > **Settings...** > **Memory** > **Use “const” Keyword** to define Constant Values
- **yes -** The constant will be generated using the “const” keyword, regardless of the settings of the MicroC code generation option: **Options** > **Settings...** >**Memory** > **Use “const” Keyword to define Constant Values**

The “Use 'const' Keyword” design attribute is available with all predefined OSIs. This attribute has the following parameters in the OSDT APIs in Memory Management page: `Variable Declaration() and Extern Variable Declaration()`.

- IsConstant has the value “yes” if the element is defined as Constant in the model
- InitValue has the initial value related with the element (for constants it will be the element’s definition)

Timeout Variable Type

In this text box, enter the data type for the Timeout Variable which holds the expiration time of a pending timeout.

Default Data Types

These text boxes are used to specify the default data types that should be used for each of the following:

- Signed Integer
- Unsigned Integer
- Bit Field
- Floating Point

The data types specified will be used when declaring data, and, in the case of signed integers and unsigned integers, will be used when using bit-wise shift operators.

The data type specified for **Bit Field** is used for conditions and bit data items. This text box is only enabled when the option (**Use Macros For**) **Conditions and Bit Data Items** on the **General** tab is not selected.

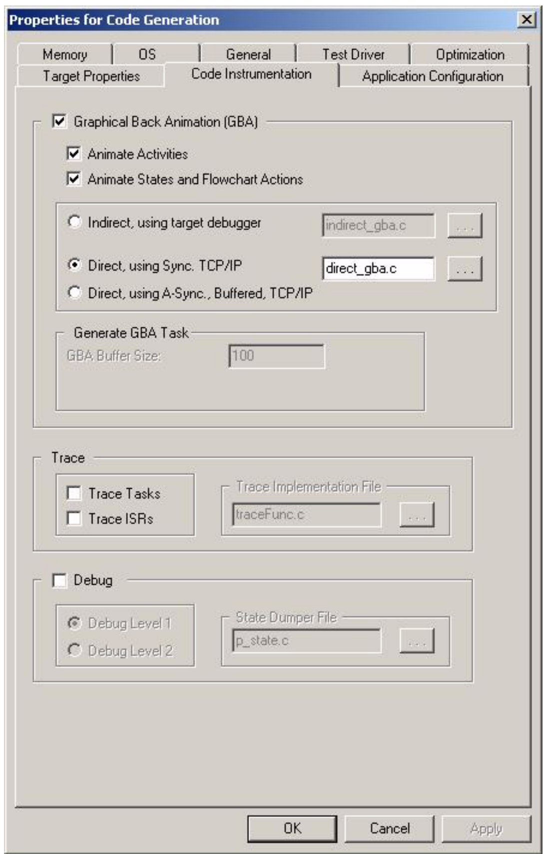
Use Fixed Point variables for "Real"

If this check box is selected, the code generated will use fixed point variables for data items of type “real.”

If you have chosen to use fixed point variables in the generated code, you can use the **Word Size** radio buttons and **LSB^-2** box to select a default word size and LSB. These default settings will be used for individual variables where “*” has been selected as the value for word size and LSB in the variable properties. (The value in the LSB box represents the negative exponent to use.)

Code Instrumentation

The following section describes the options available on the **Code Instrumentation** tab.



Graphical Back Animation (GBA)

Graphical Back Animation (GBA) allows you to view an animation of the activities/states/flowchart actions in your model. This is not a simulation, but rather an animation that runs in parallel to the running of your executable.

The GBA data is passed from the application to the Statemate model via the GBA server, which processes the data received from the application and performs the actual painting of the charts in the model.

If the **Graphical Back Animation** check box is selected, the code generator will generate the code required for this feature.

- **Animate Activities**

  If this check box is selected, the code generator will generate the flags required for displaying activity animation when the animation is run.

- **Animate States and Flowchart Actions**

  If this check box is selected, the code generator will generate the flags required for displaying animated statecharts and flowcharts when the animation is run.

If the **Graphical Back Animation** check box was selected, you can then choose one of the following animation methods:

- **Indirect, using target debugger**
  - Enables usage of GBA with a target debugger. When this method is used, the animation data is passed from the running application to the GBA Server indirectly, using a 3rd party debugger.
  - The text box next to the radio button contains the name of the source code file for the animation functions. If you prefer to use animation code that you have modified, type in the name of the appropriate source code file, or click **...** to browse for the file.
- **Direct, using Sync. TCP/IP**

  Enables usage of GBA synchronized with the running application. When this method is used, the animation data is read directly from the running application, and is passed immediately to the GBA server, using the TCP/IP protocol.

- **Direct, using A-Sync., Buffered, TCP/IP**
  - Enables usage of GBA where the animation is not synchronized with the running application. When this method is used, the animation data is read directly from the running application, but rather than being passed immediately to the GBA server, the animation data is stored in a buffer which sends the data to the GBA server when the GBA task is running, using the TCP/IP protocol.
  - This method allows you to have the animation run as a separate task.

When the buffered method is selected, you can define the following settings for the GBA task:

- **GBA Buffer Size**

  The size of the buffer to use to store the animation data.
- **GBA Task Priority (OSEK OS only)**

  Allows you to specify the priority of the GBA task.
- **More... (OSEK OS only**)

  Opens the **Task Specific Attributes** dialog box.

Trace

The Trace option allows you to include code that will print to the screen information regarding the current status of tasks and/or ISRs.

- **Trace Tasks**

  When selected, code will be included for tracing tasks.
- **Trace ISRs**

  When selected, code will be included for tracing ISRs.
- **Trace Implementation File**
  - The text box next to the check boxes contains the name of the source code file for the trace functions. If you prefer to use trace code that you have modified, type in the name of the appropriate source code file, or click **...** to browse for the file.
  - The trace functions receive two parameters—one is a value which identifies the task/ISR, and the other is a character that identifies the status of the task/ISR (started, terminated, entering/exiting wait for event).

Debug

The Debug option allows you to include code that will print to the screen the state the application is in for each level of detail in the statechart. (Level 1 means that each super-step is reported, while Level 2 means that each step is reported.)

When the Debug check box is selected, you can define the following debug settings:

- **Debug Level 1**

  The state the application is in will be reported at the end of every superstep of the state machine (stable state).

- **Debug Level 2**

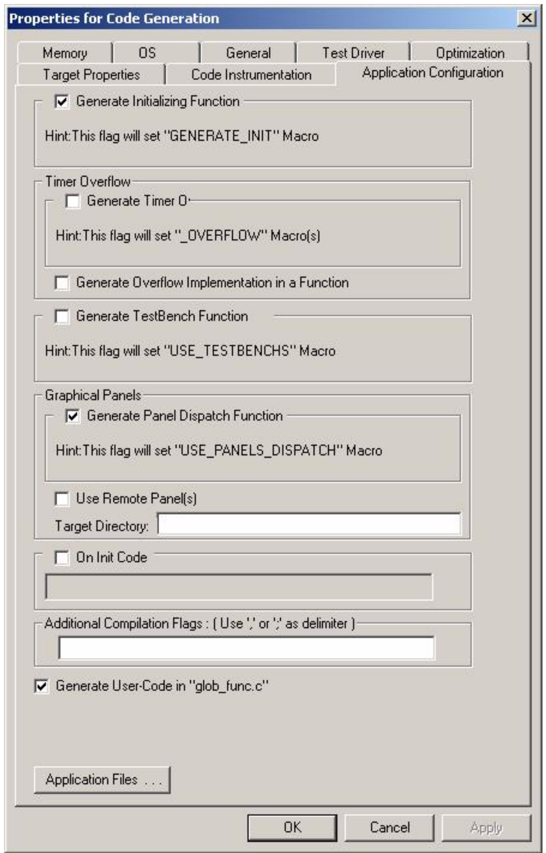
  The state the application is in will be reported at the end of every step of the state machine.

- **State Dumper File**

  The text box next to the radio buttons contains the name of the source code file for the implementation of the debug functions. If you prefer to use debug code that you have modified, type in the name of the appropriate source code file, or click **...** to browse for the file.

Application Configuration

The following section describes the options available on the Application Configuration tab.



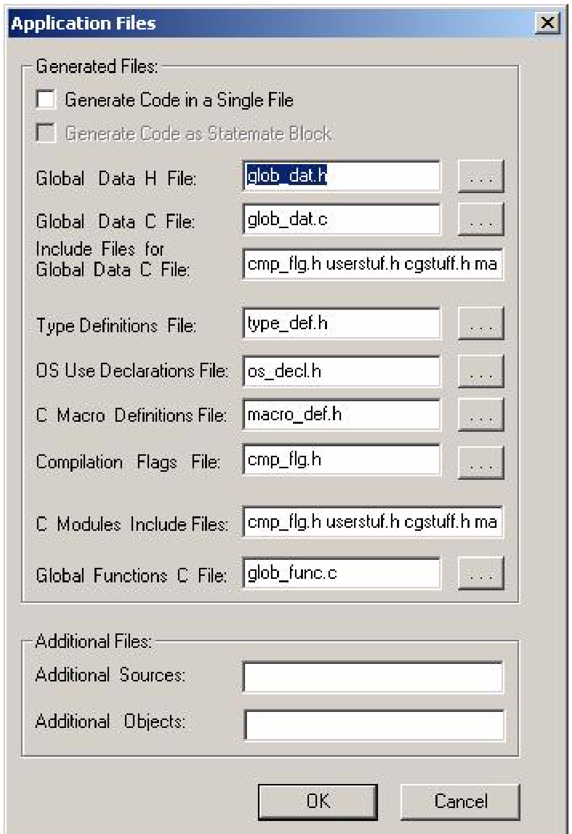
Parameter	Description
Generate Initializing Function / Task	• If you select this check box, an initializing function (or Task in OSEK) called TASKINIT will be added to the generated application. • The implementation code for TASKINIT is in the file usercode.c in the prt directory (originates in the OSI), and is later copied to the file <profile-name>.c in the output directory. • The TASKINIT function contains code related to initialization of the application, such as initialization of the panels, GBA, and model data. • You can add additional initialization code manually to usercode.c before generation, or add code in the On Init Code text box, described below. • If you are using an OSEK-based OSI, you can specify whether this task is basic or extended, specify the task priority, or define attributes in the **Task Specific Attributes** dialog box.
Generate Timer Overflow Function/Task	• If you select this check box, functions (or Tasks in OSEK) will be generated for each of the timers used with the timeouts in the model. The code in these functions handles the overflowing of the timer, and contains only generated code. If you want to modify this code, modify the appropriate APIs in the OSDT. • The calls to these functions are the user’s responsibility. • If you are using an OSEK-based OSI, you can specify whether these task are basic or extended, specify the task priority, or define attributes in the **Task Specific Attributes** dialog box.
Generate TestBench Function/ Task	• If you select this check box, a function (or Task in OSEK) will be generated to support the Testbenches in the profile. If not selected, no testbench code is generated. • If you are using an OSEK-based OSI, you can specify whether this task is basic or extended, specify the task priority, or define attributes in the **Task Specific Attributes** dialog box.

Parameter	Description
Graphical Panels	**Generate Panel Dispatch Function/Task** • If you select this check box, a function (or Task in OSEK) called PANEL_DISPATCH will be added to the generated application. • The implementation code for PANEL_DISPATCH is in the file usercode.c in the prt directory (originates in the OSI), and is later copied to the file <profile-name>.c in the output directory. • This function includes code for actions such as panel data update and panel graphics update. If you wish to use panels, this check box must be selected. • If you are using an OSEK-based OSI, you can specify whether this task is basic or extended, specify the task priority, or define attributes in the **Task Specific Attributes** dialog box. **Use Remote Panel(s)** • This check box should be selected if you want the panels and the application to run on different computers. This option can only be used if the remote host has file system capabilities (such as open, read). • If you are using this option, you must define a Target Directory. This directory is used for writing data used for communication between the panels and the application. Therefore, it must be a directory with write permission.
On Init Code	• Any code entered in this text box will be placed in a macro called ON_INIT_CODE, generated in macro_def.h • This macro is called from the function TASKINIT (which is included in the generated application if **Generate Initializing Function /Task** was selected). • The text box only accepts a single line of text, but it can contain a number of statements separated by a ";".
Additional Compilation Flags	• This text box allows you to provide values for flags in addition to those used by Statemate. These are generated in the file `cmp_flg.h`. Each flag is generated in a single line of code using the format: `#define <Flag>`. • When entering text in this text box, use ';' or "," to separate the different flags. • Flags can be specified by providing only the flag name (for example, *AAA*) or by providing the flag name and a value to assign (for example, *AAA=4*). • The `ifdef` statements for these flags can be added to code manually or can be used in defining APIs such as the memory management APIs in the OSDT.

Parameter	Description
Generate User-Code in "glob_func.c"	If this check box, is selected, all user code (functions and subroutines) will be generated into a file called glob_func.c. In this context, user code refers to the bodies of functions defined using Statemate. If this option is not selected, the code generator will generate this code in the relevant modules, as follows: • For regular subroutines, the code is generated in the module to which the subroutine belongs. • For subroutines defined with a GDS scope, the code is generated in glob_func.c. • For subroutines defined with a generic scope: bodies of functions are generated in the file g_<Generic-Name>.c. • For call-back functions for elements like data-items — code is generated in glob_func.c. • Before each section of functions in module/generic files, the code will include the definition of the API USER_FUNCTIONS_BODY_DEFINITION_SECTION_HEADER() • After each section of functions in module/generic files, the code will include the definition of the API USER_FUNCTIONS_BODY_DEFINITION_SECTION_FOOTER().

Application Files...

In order to create the code for your project, the MicroC Code Generator requires certain files. You can define the names of the files and their include lists, as well as other source files and object files by clicking **Application Files...** on the Application Configuration tab.

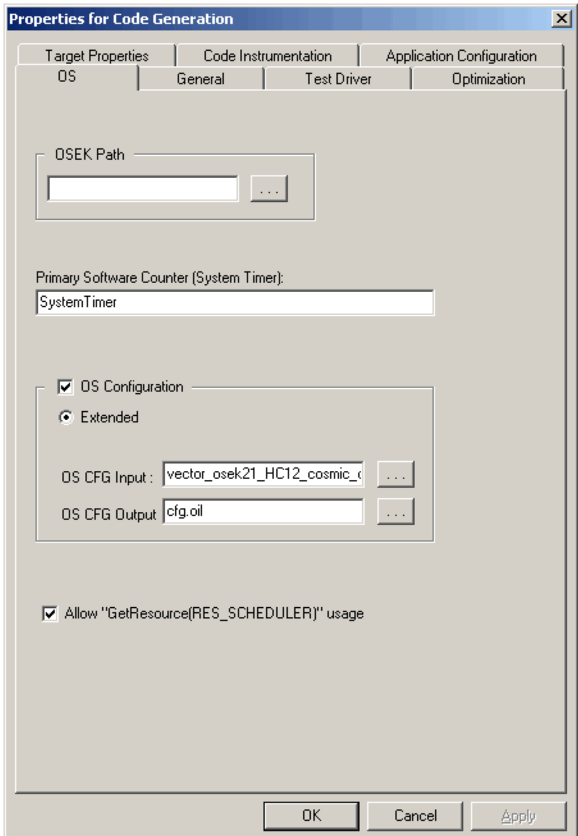


The types of files are as follows:

Type of File	Description
Global Data H File	The file in which global data forward declarations are generated (.h file).
Global Data C File	The file in which global data declarations are generated (.c file).
Include Files for Global Data C File	A list of files to be included in the global data C file. Use a space as the delimiter for separating the files in the list.
Type Definition File	The file in which types are generated (.h file).
OS Use Declarations File	The file in which OS Objects (like tasks) are declared (.h file).
C Macro Definitions File	The file in which macros are generated (.h file).
Compilation Flags File	The file in which compiler flags are generated (.h file).
C Modules Include Files	A list of files to be included in the module's C files. Use a space as the delimiter for separating the files in the list.
Global Functions C File	The file in which user functions and other global functions are generated (.c file).
Additional Sources	A list of additional sources to be included in the building of the application. This information is added to the makefile (if used).
Additional Objects	A list of additional objects (.obj, .lib etc) to be included in the building of the application. This information is added to the makefile (if used).

OS

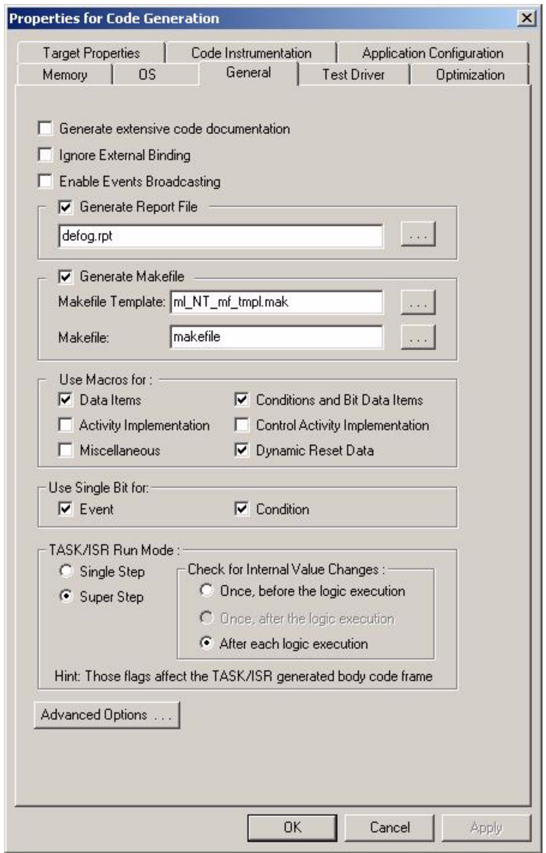
The following section describes the options available on the OS tab.



Parameter	Description
Osek Path	For OSEK-based OSIs only. The path to the OSEK installation root.
Primary Software Counter (System Timer)	The name of the primary software counter, used with the timeout and delay operations, and other time-related code.
OS Configuration	If you select this check box, the MicroC Code Generator will generate a static OS configuration file. This option is only displayed if you selected **Static OS Configuration** on the main screen of the OSDT. • **OS CFG Input** Enter in the text box (or locate with the **...** button) the template file to use for the creation of the OS configuration file. The keywords used in the template file will be replaced with concrete data from the model to create the OS configuration file that reflects the OS objects in the model. • **OS CFG Output** The name to use for the generated static OS configuration file.
Allow "GetResource(RES_SCHEDULER)" usage	This option is displayed for OSEK-based OSIs only. If this check box is selected, then when a Task/ISR has related timeouts, the code generator calls `GetResource(RES_SCHEDULER) / ReleaseResource(RES_SCHEDULER)` around the code section that swaps the Task/ISR event buffer. It also calls these functions around the call to `genTmEvent(...)` in on<TIMER>OVERFLOW tasks (in the file `glob_func.c`).

General

The following section describes the options available on the General tab.



Parameter	Description
Generate extensive code documentation	By default, the MicroC Code Generator includes basic commenting in the generated code. If you would like more extensive commenting in the generated code, select this check box. If this option is selected, comments will be included for all of the following: body of activity functions, statechart/flowchart implementation functions, state transitions, static reactions in a state, data declarations, and headers and footers for all generated files. In addition, the following will be included: • Model information • Information regarding the code generation profile used, such as profile name, date, version • Workarea and project name • For statecharts, textual transition table before the implementation function • For truth tables, a textual description of the table For timeout setting, the expression triggering the code
Ignore External Binding	This option allows you to generate code while ignoring any external bindings specified in the model. This means that you can ignore these external elements without having to make any changes to the model—you just have to use a profile where this option is selected.
Enable Events Broadcasting	If this option is selected, event broadcasting will be enabled. This means that if an event is generated anywhere in the model, then a duplicate event will be generated for each task waiting for that event (other than the task that “owns” the event), enabling them to react to the event. The duplicated events will have the name <original-Event-name>_<task-name>. For example, if the event EV is duplicated for a task named T1, the duplicated event will be given the name EV_T1. (If this creates a naming conflict, a number will be added to the end of the name.) Wherever the original event was used as a trigger, the name of the corresponding duplicated event will appear in the code. Wherever the original event was used in an action, the original event name will appear in the code. The duplicated events will be generated in the “new buffer” of their task event buffer. However, it is possible to use GEN_IN_CURRENT for the duplicated event if the following env-var is used: set AMC_GEN_IN_CURR_FOR_EV_BROADCAST=ON

Parameter	Description

Generate Report File	If this option is selected, a report file will be generated, using the file name that you provide.
	This report does not include messages generated during code generation. Rather, it provides information regarding the model.
	The report file has three parts:
	• Task description section, which includes description of all Tasks in the model and of the “Global” task.
	For each Task there is information about:
	Task's attributes like: Type, ID, priority etc.
	Data related to it, like: Dataitems Conditions and Events related to the Task, and their attributes like: double-buffer etc.
	• User defined Functions:
	A list of the user-defined functions in the model
	• Generated code description:
	A data-base like description of the elements that were used in the code generations: Activities, Conditions, Dataitems, Events, User-defined types, Actions and Functions.
	Each element has a different set of fields in its description structure according to the available information about the element in the following format:
	`{`
	`"type" : <type`
	`"Model Name": <model-name`
	`More…`
	`}`
	The messages generated during code generation can be found in a file called wrn.err. If any problems are encountered during code generation, the information in this file is displayed to the user.

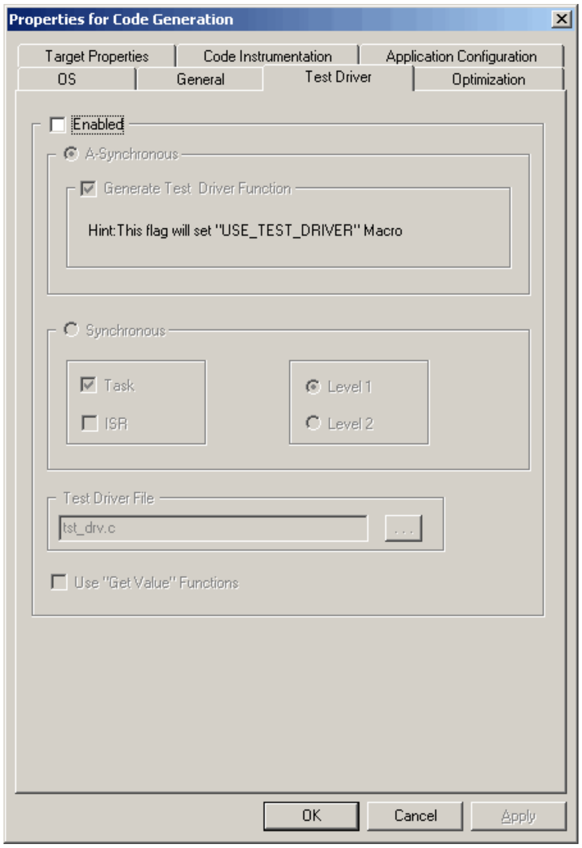
Generate Makefile	If this option is selected, a makefile will be generated for building the application. • **Makefile Template** Enter in the text box (or locate with the ... button) the template file to use for the creation of the makefile. The keywords used in the template file will be replaced with concrete information regarding the model’s files and objects. • **Makefile** The name to use for the makefile that is generated.

Parameter	Description
Use Macros for	• **Data Items** If selected, a macro will be used (rather than a full C expression) when referring to a data item. • **Activity Implementation** If selected, macros will be used (rather than functions) for activity implementation. • **Miscellaneous** If selected, macros will be used (rather than full expression / function) for elements such as actions. • **Conditions and Bit Data Items** If selected, macros will be used (rather than full expressions) for Conditions and Data Items of type Bit. • **Control Activity Implementation** If selected, macros will be used (rather than functions) for control activity implementation.
Use Single Bit For	• **Event** If selected, single bit with bit mask will be used for events, rather than a whole byte. • **Condition** If selected, single bit with bit mask will be used for conditions, rather than a whole byte.

Parameter	Description
Task/ISR Run Mode	The radio buttons provided for this option allow you to select the task/ ISR run mode (single step or superstep, as well as specifying when objects should be checked for value changes. **Single Step** • In this mode, each task/ISR performs a single step and then returns control to the operating system. • When single step mode is selected, the value checking options available are: once, before logic execution once, after logic execution **Superstep** • In this mode, each task/ISR performs as many steps as needed to reach a stable state (reached when there are no more pending events and no transition occurred in the last step) • When superstep mode is selected, the value checking options available are: once, before logic execution after, each logic execution • This means that values are checked for changes on each pass through the execution loop until a stable state is reached When superstep mode is selected, care should be exercised when using the once, before logic execution option. This is because there are situations where your system will remain in a certain state until a certain value changes. If the values are only checked once, before the logic is executed, then in many cases the system will never detect the value change that takes the system out of that state. This may result in a task running indefinitely.

Test Driver

The following section describes the options available on the Test Driver tab.

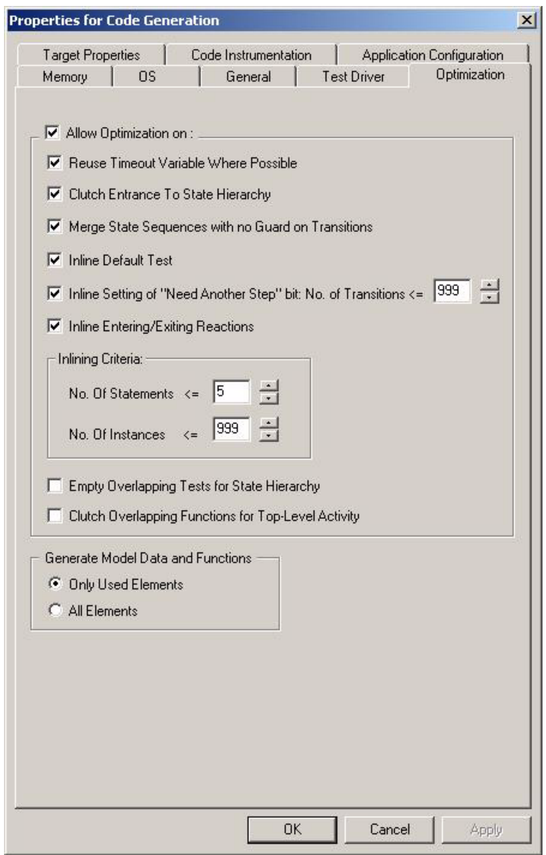


Parameter	Description
Enabled	If the **Enabled** check box is selected, then the code generator will produce the code required for using the MicroC Test Driver. The Test Driver allows the running of scenarios whose input has been saved to file. The outputs can then be compared with the recorded outputs. When the check box is selected, the remaining controls on the tab are enabled.
Asynchronous/ Sunchronous (The Test Driver can be run in either of these modes)	**Synchronous** The Test Driver is synchronized with the running application. When this method is used, the test-related data is read directly from the running application, and is executed immediately. • **Task** If this check box is selected, a call to the Test Driver dispatch function is generated in the task's code frame. • **ISR** If this check box is selected, a call to the Test Driver dispatch function is generated in the ISR's code frame. • **Level 1, Level 2** If **Level 1** is selected, then the call to the Test Driver dispatch function is made after every task/ISR's superstep. If **Level 2** is selected, then the call to the Test Driver dispatch function is made after every task/ISR's step. **Asynchronous** The Test Driver is not synchronized with the running application. When this method is used, the data is read directly from the running application, but rather than being executed immediately, the test data is stored in a buffer which stores the data until the Test Driver Task is running. • **Generate Test Driver Function/Task** If this option is selected, the code generator generates a function called `TEST_DRIVER_TASK` which launches the Test Driver. • **Type (OSEK OS only)** Allows you to specify the type of the Test Driver task - Basic or Extended. • **Task Priority (OSEK OS only)** Allows you to specify the priority of the Test Driver task. • **More... (OSEK OS only)** Opens the *Task Specific Attributes* dialog.

Parameter	Description
Test Driver File	This text box contains the name of the source code file for the Test Driver implementation. If you prefer to use Test Driver code that you have modified, type in the name of the appropriate source code file, or click **...** to browse for the file.
Use "Get Value" Functions	If this option is selected, "get" functions will be generated for retrieving the values of elements that are being tested. These functions are not called in the default Test Driver implementation, but if you are using modified code, you can include calls to these functions.

Optimization

The following section describes the options available on the Optimization tab.



Parameter	Description	
Allow Optimization on This check box enables selection of the various optimization options available.	**Reuse Timeout Variable Where Possible**	If selected, the code generator will attempt to combine multiple timeout variables into a single variable if they are exclusive, i.e., the timeouts can't be pending at the same time. (RAM optimization)
	Clutch Entrance To State Hierarchy	If selected, the code will be optimized to enter the innermost state in a state hierarchy, wherever possible. (ROM, runtime optimization)
	Merge State Sequences with no Guard on Transition	If selected, code will be optimized to merge state sequences with no guards on transition into a single state wherever possible. (RAM, ROM, runtime optimization)

	Inline Default Test	• If selected, the code will be optimized to inline the test on default transitions into other transitions' testing. (RAM, ROM optimization).
		• The code below represents the "before" and "after" for this option.
		if(currentState_S1 == 0){
		currentState_S1 =
		FS_DefaultOfS1;
		} else {...
		
		`if(currentState_S1 == 0 ||`
		`inState(DefaultOf_S1)){...`

Parameter	Description	
Allow Optimization on (Continued)	**Inline Setting of “Need Another Step” bit**	• At the end of the statechart code, there is a section that tests whether a transition was made in the current step. If so, a flag is raised to indicate that the superstep task should perform another step over the task's code frame. This test of whether another step is needed uses a state variable that stores the information about the state that is being entered. • When this option is selected, the code flagging the need for another step will be put inline in the transition code, eliminating the need for the state variable that stores the transition's target state. • When selected, you can enter the maximum number of transitions you are willing to tolerate. If the number of actual transitions will be greater than this, the optimization will not be performed. (RAM, ROM optimization)
	Inline Entering/ Exiting Reactions	• If selected, the code is optimized to try to inline the entering and exiting reactions of states, in order to avoid generating the entering and exiting reaction functions. (RAM, ROM optimization) • When this option is selected, you can also specify the maximum number of statements that should be inlined. You can also specify the maximum number of instances - you may not want this inlining to be carried out if it will have to be done a large number of times.
	Empty Overlapping Tests for State Hierarchy	• When there is a state hierarchy, you are obviously not in the inner state if you have not entered the surrounding state. When this option is selected, the code will be optimized by skipping such overlapping comparisons. (ROM optimization, but may reduce runtime efficiency)

Parameter	Description
Generate Model Data and Functions	This option allows you to specify that code should be generated for all elements in the model, or only for elements that are used. If the latter option is selected, then if variables/functions are defined but not used/called anywhere in the code, no code will be generated for them.

Setting the Time Expression Scale

The **Options** menu in the MicroC Code Generator Window also contains an item called **Time Expression Scale**.

This option allows you to set the time scale for all expressions in the model. For a detailed explanation of this feature, see the MicroC Code Generator section in the Statemate User Guide.

Modules, Adding Charts

As mentioned earlier, defining a profile consists of defining what code should be generated, and then defining options to determine how it will be generated. You define what code should be generated by creating modules in a profile and then adding charts to the modules created.

Create Modules

Modules allow you to organize the code that is to be generated. Once you have created a module, select the charts that you would like to add to the module. When the code is generated, the code for all charts in a module will be included in a single file.

Add Charts to Modules

To add charts to modules:

1. Select the relevant module
2. Select **Edit > Add with Descendants** from the main menu or click the corresponding button in the toolbar.

Direct Editing of Profile Files

While you will most likely use the Profile GUI for making modifications to the profile, it is also possible to directly edit the file that contains the profile information.

To edit the file:

1. Select **File> Profile Management** from the MicroC Code Generator menu.
2. Select the profile.
3. Click **Show**.

Checking for Errors in Profiles

After a profile has been defined, before generation of the code, you can check the profile for errors.

To check the profile, select **Compile > Check Profile** from the MicroC Code Generator menu.

Generating and Running MicroC Code

Once a profile has been defined, you can use the MicroC Code Generator to generate code from your model. After the code has been generated, you can:

- Edit the code
- Compile the code
- Run the code
- Run the code with animation

All of these options are selected from the Compile menu in the MicroC Code Generator window.

Checking Profile Before Generating Code

Since the content of the generated code is determined by the profile you have defined, it is recommended that you use the Check Profile feature before generating code from the model.

This verifies that the profile complies with the scoping rules. For example, the profile settings will be checked to make sure that they are legal and that they do not conflict with each other.

To check the current profile, follow these steps:

1. Select **Compile** > **Check Profile...** from the MicroC Code Generator window. Any error, warning, or information messages will be displayed in the Check Profile dialog.
2. Click **Dismiss** to close the Check Profile dialog.

Generating Code

To generate code, follow these steps:

1. Select **Compile** > **Generate Code**... from the MicroC Code Generator window.
2. When the directory tree is displayed, select the directory where the generated code files should be stored, and click **OK**.
3. The Generate Code dialog will display relevant messages regarding the progress and results of the code generation process. Select **Dismiss** to close the Generate Code dialog.

Editing Code

To edit one of the generated files, follow these steps:

1. Select **Compile** > **Edit Code...** from the MicroC Code Generator window.
2. The contents of the output directory you selected will be displayed. Select the file that you would like to edit.
3. The contents of the selected file will be displayed in the default text editor that has been defined for Statemate.

Note: To change the editor that is launched, select **Project** > **General Preferences** from the main Statemate menu, and enter the relevant command line for the parameter Editor Command Line. (You may have to close and reopen the MicroC Code Generator window for the change to take effect.)

Compiling Code

To compile the code, follow these steps:

1. Select **Compile** > **Make Code...** from the MicroC Code Generator window.
2. The contents of the output directory you defined will be displayed. Select the makefile to use, and click **Open**. (The name of the makefile defined in the profile's properties is offered by default.)

Running Code

To run the compiled code, follow these steps:

1. Select **Compile** > **Run Code** from the MicroC Code Generator window. The Run Command dialog is displayed.
2. Locate the file to run, and click **Open**.

Running Code with Animation

If you have defined an interface panel for your application, you can run the code with animation to follow the transition between different states, the activation/deactivation of activities, or the execution of a flowchart.

To run your code with animation, follow these steps:

1. Select **Options** > **Settings** from the MicroC Code Generator menu.
2. On the Code Instrumentation tab, select the **Graphical Back Animation (GBA)** check box.
3. Save the modified profile (**OK** to close the Settings dialog, and then **File** > **Save** from the MicroC Code Generator menu.)
4. Regenerate the code (**Compile** > **Generate Code...** from the MicroC Code Generator menu).
5. Recompile the code (**Compile** > **Make Code...** from the MicroC Code Generator menu).
6. Launch the GBA server (**Tools** > **Open GBA** from the MicroC Code Generator menu).
7. Run the application (**Compile** > **Run Code...** from the MicroC Code Generator menu).

    Note: If you selected the GBA option in the profile settings, then the generated application will automatically try to connect with the GBA server in order to run the animation. If you have not launched the GBA server, a message will be displayed indicating that the connection could not be established, and the generated application will continue running without the animation.

Designing Your Model: Model-Code Correspondence

The code that is generated is based on the model's graphical elements, textual elements, and the design attribute settings within these elements.

Note

This section contains many code examples. Most of these were taken from OSEK projects. Keep in mind that these are examples only. The same principles apply to non-OSEK projects as well.

To design the model, Statemate provides graphical tools for the following:

- Activity charts
- Statecharts
- Flowcharts

In addition, Statemate allows you to define the following:

- Truth tables
- Lookup tables

Activity Charts

When designing an activity chart, activities are broken down into sub-activities, which are further broken down into their sub-activities, and so on, until no further decomposition is possible. Activities that cannot be broken down further are considered “basic” activities.

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 subactivity functions. For a basic activity, the function contains the implementation code.

Activities can represent functions, tasks, or ISRs (Interrupt Service Routines).

Task Activities

In the OSEK 2.0 operating system, Tasks can be divided into *basic tasks* and *extended tasks*:

- A *basic Task* runs once, upon activation, and then terminates.
- An *extended Task* runs once, upon activation, and then suspends itself, calling the API function “WaitEvent”.

Basic Task - Generated Code

The code for a basic Task that contains activities A11 and A12 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 resemble the following:

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

The code for a periodic Task, containing activities A11 and A12 with CTRL1 as a controller, will resemble the following:

```
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 - Generated Code

The code for an extended Task that contains activities A21 and A22 will resemble 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(); */
}
```

If a statechart is added beneath the Task, but not as a direct descendant, the code will resemble the following:

```
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 resemble the following:

```
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(); */
}
```

ISR (Interrupt Service Routine) Activities

Interrupt Service Routines (ISRs) run once, upon activation, and then end.

ISR Categories

For OSEK 2.0, three ISR categories can be used: 1, 2, and 3.

The decision of which ISR category to use depends on the content of the functions it runs. According to the OSEK/OS specification, an OS API function cannot be called from a category 1 ISR. For categories 2 and 3, some OS API functions can be called, but only within code sections marked by `EnterISR()/LeaveISR()` calls.

The following are some code examples for different types of ISRs:

ISR - Examples of Generated Code

Code for a category 1 or 2 ISR, named ISR0, containing activities I01 and I02:

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

Code for a category 3 ISR function names ISR0, containing activities I01 and I02:

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

Code for a category 3 ISR function named ISR1, containing activities I11 and I12, and a controller named CTRL1:

```
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 Mode

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.

Decomposition of Non-basic Activities

When a non-basic activity does not contain an immediate descendant that is a control activity, all of the activity's subactivities are considered active when the activity is active. For such a non-basic activity, the generated code will resemble the following:

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

Execution Order (for Subactivities)

The order in which the subactivities are called within the A11 activity body is determined by the subactivity design attribute *Execution Order*, as defined for A111, A112, A113. In the previous example, the value of this attribute was “1” for subactivity A111 and “2” for subactivity A112.

If the *Execution Order* attribute is not set, the calling order is not defined.

Code for Basic Subactivities

Basic activities can be defined in one of three activation modes:

- Reactive controlled
- Reactive self
- Procedure-like

For reactive controlled and reactive self modes, the code for the basic activity will resemble the following:

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

For the procedure-like mode, the code for the basic activity will resemble the following:

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

Adding a 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()*, looking 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.

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
}
```

Communication and Synchronization Services between Activities

Communication and synchronization services between activities, including those not residing in the same task/ISR, consist of the following:

- Non-queued messages
- Queued messages
- Signals
- Global Data
- Semaphores

Non-queued Messages

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 for such a message can be running in the same ECU, share the same memory address space, or run across and ECU network on a remote MCU. The user of the message need not be aware of the concrete implementation. Thus, use of this mechanism ensures that the resulting design is correct, flexible, and efficient.

Queued Messages

The queued messages mechanism is similar to that of non-queued messages. The difference is that queued messages do not contain values but rather signal the occurrence of some event.

Signals

Signals indicate the occurrence of some event. However, since they are not queued, there is no information regarding how many such events occurred, until they are processed.

Also, these Task Event signals require that a specific task is addresses with a specific event, thus requiring knowledge of the application structure. The Task Event implementation is more efficient than ordinary or queued messages, however the task must be of type extended, which is not always possible or efficient.

The downside of requiring knowledge of the application is balanced by the improved performance. The weight assigned to these two issues will depend on the problem at hand.

Global Data

As always with real time applications, when using global data, caution should be taken regarding the validity of the data when running in a preemptive environment with multiple tasks and ISRs. The protection mechanism supported is the OSEK RESOURCE mechanism, which is similar to a binary semaphore.

Semaphores

It is common for data to arrive through the bus or board ports, in some predefined messages and addresses, and must be reproduced to the bus or board in the form of other predefined messages and addresses.

In such situations, the designer simply uses the defined interface for his application. However, the discussion above is relevant when one tries to build an implementation that will use the appropriate interfaces but will also be easy to maintain, modify, and ported to various other environments, usually unknown at design time.

Statecharts

Statecharts define the behavior of activities defined in activity diagrams, and are linked to an activity with a control activity. Statecharts can contain sub-charts (nested statecharts).

This section provides details regarding the code that is generated for statecharts in your Statemate model.

Functions Generated for Statecharts

For a control activity All_CTRL, the following two functions will be generated:

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

The code generated for these functions will be as follows:

```
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();
}
```

Statechart - Data Usage

When a statechart is created, a `StateInfo` data type is defined and a few variables of that type are declared.

For the previous example, the `StateInfo` data type would be named `StateInfo_A11_CTRLcnt1`, and would be defined as an unsigned type of 8, 16, or 32 bits (e.g., `typedef int8 StateInfo_A11_CTRLcnt1`)

The StateInfo variables will be `currentState`, `nextState`, and `staySame`:

```
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 the entering or exiting reaction function is required.

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

Statechart - Generated Functions

In general, functions generated from statecharts will resemble the following (the provided line numbers are used in explanations of the code below):

```
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 line 4, the `nextState` variable is reset. This variable will be set only if a transition has been made, and will hold the statechart's new state configuration.

Lines 13 and 14 check the `nextState` variable to determine whether a transition was made, and whether to enforce another step in the task holding the statechart.

Line 16 advances the statechart configuration a step to hold the configuration of the next step.

In your statechart, lines 5 to 12 will be replaced with specific code resulting from the specified statechart logic. For example, in many cases, two additional functions will be 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;
    }
}
```

Order of Function Execution Rules

The following rules determine the order in which exiting actions, entering actions, transition actions, and static reactions are carried out for a state transition:

1. Where the state has not changed, static reactions are carried out in descending order down the state hierarchy.
2. When a transition is detected, the transition action is carried out immediately.
3. Exiting actions are then carried out, from the innermost state exited to the outermost state.
4. Finally, entering actions are carried out, from the outermost state entered to the innermost state.

Note: In order to optimize code, you may sometimes want to inline entering/exiting reactions. For more details, see **Optimization**.

State Variable Validation Macro

The `IS_IN_VALID_STATE_<ctrl-activity-name>` macro is defined in the generated file *macro_def.h* for each Control-Activity Statechart hierarchy. This macro includes code for validating that the state variable has a valid value. The validation is accomplished using the `inLeafState()(inState()` for And States macros against all possible leaf states in the hierarchy. In addition, there is a test against the valid "0" value.

For example, for the Control Activity **(CTRL)** with two leaf states **(S2** and **S2),** the generated macro is as follows:

```
#define IS_IN_VALID_STATE_CTRL ( \
inLeafState(currentState_CTRL, S1, StateInfo_CTRL) || \
inLeafState(currentState_CTRL, S2, StateInfo_CTRL) || \
currentState_CTRL == 0 \
)
```

Timeout Implementation

Software counters 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.

Other software counters can 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)
```

where `myCounter` is the name of the counter. Each counter receives an index value defined as <counter_name>_INDEX. The index value identifies that specific counter in the application.

INSTALL_TIMEOUT Macro

The INSTALL_TIMEOUT macro has three arguments:

- The name of the event.
- The requested delay.
- The index of the counter on which it is pending

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_TIMER. 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);
```

This 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 carried out 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:

   a. In the file, *glob_dat.c* a uint8 variable tm_999999993_counter; is generated, holding the index of the current relevant counter.

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

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 on which the timeout is pending.

Special Requirements for OSEK-targeted Applications

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 Timeout variables that refer 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-name>.c after the installation of the `SYS_TIMER_ALARM` (formerly known as SYS_TIME_OVERFLOW).

Compare this 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 Timeout 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.

History and Deep History Implementation

History and deep history implementation require a *StateInfo* variable for each state with a history connector or deep history connector.

The state configuration is stored in this *StateInfo* variable, such that when a transition occurs into the history/deep history, this configuration is assigned to the *nextState* variable, causing an entrance to the stored state configuration.

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

Optimization of Statechart Code

There are a number of optimization options available for state chart code. For more information regarding these optimization options, see **Optimization** in **Working with Profiles**.

Recommendations for Efficient Code

To increase the efficiency of your code, it is recommended that you:

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

Flowcharts

A flowchart is another type of diagram that can be used to define the behavior of an activity.

For the purpose of code generation, a flowchart is considered to be the flowchart directly connected to a control activity, as well as all of its sub-charts and the generics instantiated within them.

Functions Generated for Flowcharts

For a control activity, A12_CTRL, the following two C functions will be generated:

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

The body of these functions will resemble 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`.

Flowchart Implementation

The flowchart language that is used graphically describes a structured C program.

While both flowcharts and statecharts define the behavior of activities, the graphics and semantics used in flowcharts are very different from those used in statecharts. In some cases, you may prefer this approach to the statechart approach.

The code of a flowchart runs from beginning to end, without stopping and without explicitly maintaining its internal state. Flowcharts do not have a notion of state or internal state.

While flowcharts allow the creation of highly visible, graphical algorithms, there is no inherent overhead in the code that is generated from the chart. The MicroC code generator produces optimized structured code, just as it does for statecharts.

If a flowchart is properly constructed, it will result in the generation of highly optimized 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.

Flowchart Elements

The elements found in flowcharts can be divided into two categories:

- Boxes
- Arrows

Compound boxes—boxes containing other boxes—represent code blocks.

Labels

As with statecharts, the graphical elements in statecharts 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

The following expressions are permitted for decisions:

- Event (such as ON_POWERUP)
- Condition (such as [POWER_ON])
- Expressions (such as [TEMP > 27])

The following expressions are permitted on arrows exiting decisions:

- Yes
- No
- True
- False

Switch Expressions

The following expressions are permitted for switches:

- Value-type expressions (such as F1(3) + 5)

The following expressions are permitted on arrows exiting switches:

- Literal constants
- else
- default

Minimization of Goto Statements

The MicroC Code Generator tries to minimize the number of goto statements in the code. This results in more structured and readable code. However, this is not always possible, and in some cases `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 in the flowchart are translated into blocks.

Basic boxes are interpreted as control positions between executable statements.

Begin/End Points

The start point for each block (the point at which the non-basic box is entered) is marked using a Start arrow in that box. The end point is marked using an End connector in that box.

The start point for the entire flowchart is marked using a Start arrow at the highest level. The end point for the entire flowchart is marked using an End connector at the highest level.

Flowchart execution stops when it can make no more progress. This may be due to reaching an End connector, or it may be due to reaching a box for which all of the outgoing arrows evaluate to false.

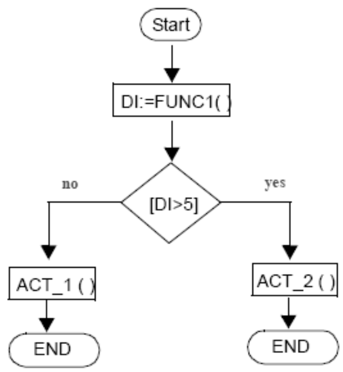
Arrows and Labels

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. This continues until the highest level is reached. If no arrows can be taken at that level, the code finishes executing, i.e., the function returns.

Flowchart Examples

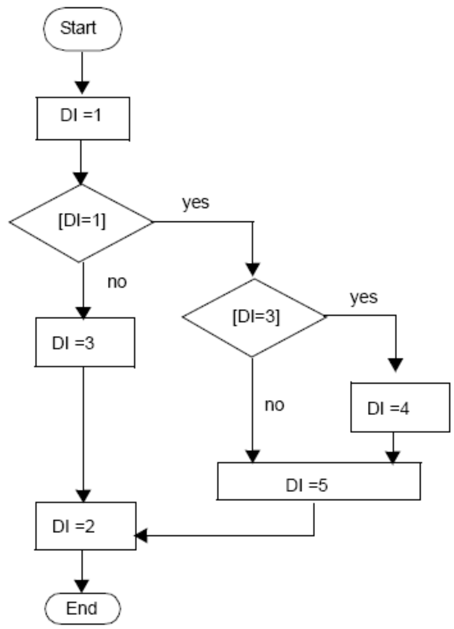
In the examples below, the flowchart is followed by the code generated for the flowchart.

Simple Flowchart



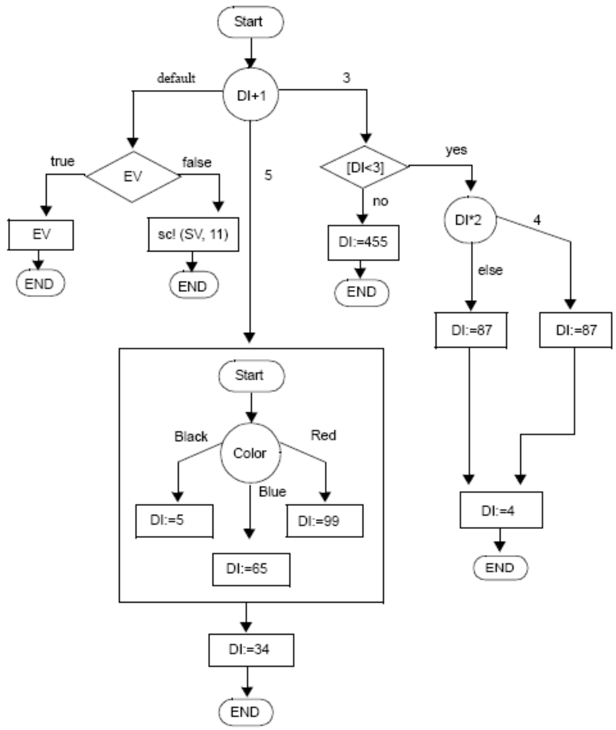

```
void cgDo_FL_CH_TEST_3()
{
       DI=FUNC1();
       if (DI > 5) {
           ACT_2();
       }
       else {
           ACT_1();
       }
}
```

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

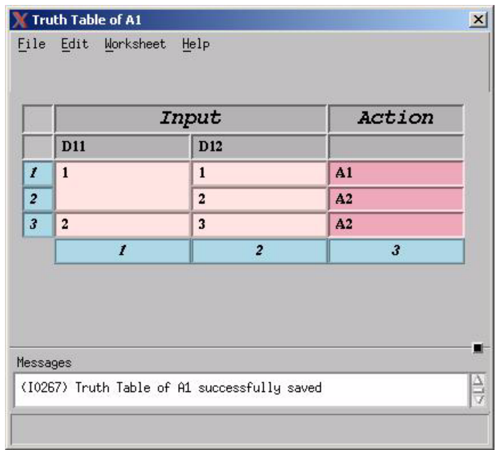



```
void cgDo_USE_SWITCH_CTRL()
{
        switch(DI + 1) {
          case 3:
              if ((DI < 3)) {
                  switch(DI * 2) {
                  case 4:
                   DI = 43;
                   break;
                  default:
                   DI = 87;
                   break;
                  }
                  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 code implementation of truth tables is demonstrated below using the following sample truth table implementing function *F*, using data items *DI1* and *DI2* as input.



For this truth table, the following code would be generated:

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

Lookup Table Implementation

Statemate allows the definition of lookup tables to represent the type of non-linear Y=F(X) functions that are so common in the world of microcontrollers. The data for a lookup table can be defined manually in Statemate, or imported from any ASCII data file. You can elect to have linear interpolation between defined points, or a histogram-like mode. Upper and lower bounds can be defined, as can the search order to use (low-to-high, high-to-low).

In the sample lookup table below, the input is defined as *Integer*, and the return value of the function is defined as *Real*.

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

Using the settings *linear interpolation*, *high-to-low* search order, *lower bound* = 0, *upper bound* = 4, the following code will be generated:

```
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);
}
```

Fixed-Point Variable Support

This section describes the MicroC Code Generator's fixed-point support for integer arithmetic, which scales integer variables so that they can represent non-integral values (fractions). This feature allows you to perform calculations involving fractions without requiring floating-point support from the target.

Statemate's MicroC Code Generator supports fixed-point arithmetic at the model level, as well as in the generated code.

Fixed-Point Variable Implementation Method

Statemate's MicroC Code Generator uses the "2 factorials" implementation method—redefining the least significant bit (LSB) to represent zero, or the negative power of 2. This implementation is not the most accurate method but it provides reasonable code size and runtime performance.

For example, take the binary 8-bit value 0b00010001. Usually, the value represented here is "17":

- The LSB (1st bit) corresponds to 2^0 (1).
- The 5th bit corresponds to 2^4 (16).
  Rescaling this 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

The `wordSize` and `shift` of an object are defined by its design attributes (specified in the element properties). The MicroC Code Generator determines the `wordSize` and `shift` of expressions made of objects and operators by using the formulas listed in the macro definition table below.

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 evaluated `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, the special functions (or C macros) provided in the `FXP` package are used to perform the calculations. The following table lists these macros

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 using` 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**

  The MicroC Code Generator 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

To specify fixed-point variables in the Code Generator, follow these steps:

1. Select **Options** > **Settings** > **Target Properties** from the MicroC Code Generator window.
2. Select the option **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. Data is allocated 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);
```

Usage of Upper Case / Lower Case in Statemate

When you create a model element in Statemate, the element name that is saved in the Statemate database reflects your exact case usage (lower case, upper case, mixed).

However, Statemate also provides a preference called *Exact Case Mode* (under **General Preferences**), which allows you to specify how the element name should appear in both the graphical editors and in the generated code. If this preference is set to *On*, then the element names will appear just as you typed them, in both the graphical editors and in the generated code. If this preference is set to *Off*, then the element names will appear in all upper case.

The mode that you select does not affect the way the names are saved in the database. There, they always remain exact case. So to restore exact case usage in both the graphical editors and the generated code, all you have to do is change the setting of the *Exact Case Mode* preference.

Statemate will not let you define two elements with the same name, differing only in the case used, for example, an event called *aB* and an event called *Ab*. If you enter a different-case variation of a name, Statemate automatically converts it to the exact case usage of the original element.

If you want to change the case used for an element, you must use the *Rename* option.

Advanced: Creating Customized OSIs

In addition to the code customization that can be achieved through the use of profiles (described above), you can use the OSDT (Operating System Definition Tool) to further customize code generation in order to create code appropriate for your target operating system. The result of this process is an OSI (Operating System Implementation).

Since it is assumed that this level of customization will be performed at the project level (and not at the individual user level), the OSDT is only installed with Statemate if it is selected during a custom installation.

Using the OSDT to Customize OSIs

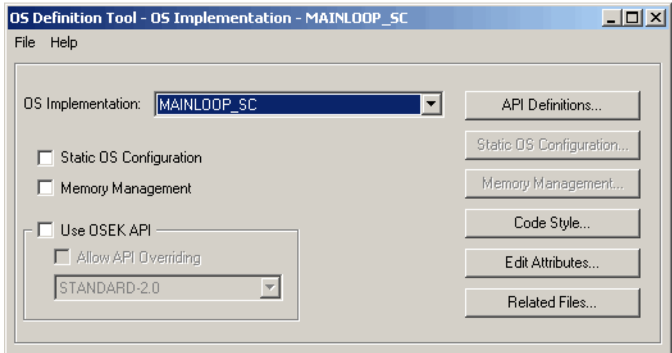
The OSDT includes predefined OSIs (operating system implementations) for a number of operating systems. If you are using one of these systems, you do not need to perform any customization.

If, however, customization is required, it is recommended that you select the OSI that is closest to the system you will be using and use this as the base to which you add the required customizations.

To use an existing OSI as a template, select the desired OSI from the **OS Implementation** drop-down list. Then, select **File** > **Save As** from the OSDT menu, and provide the name that you would like to use for the new OSI you are creating.

After making any changes to individual OSI settings on the various tabs, select **File** > **Save** from the OSDT menu to save the changes to your OSI. If you try to close the OSDT before changes have been saved, you will be asked if you want to save the changes.

Statemate will not allow you to save changes to the predefined OSIs. If you make any changes to these OSIs, you will be asked to save the modified profile under a different name.



Static OS Configuration

If you are designing software for a system that requires a static configuration file, such as an OSEK OIL file, select the check box **Static OS Configuration**.

When this option is selected, the **Static OS Configuration...** button is enabled, allowing you to define the relevant APIs. This will also result in the static OS options being displayed on the OS tab of the profile settings.

Memory Management

If you select the **Memory Management** check box, the **Memory Management...** button will be enabled, allowing you to define various code options such as:

- directives specifying how data should be stored in memory
- directives such as *#ifdef* to include/exclude parts of the code

OSEK API

If you are designing an OSEK-based system, select the **Use OSEK API** check box, and select one of the OSEK implementations from the drop-down list below the check box.

If you want to customize the OSEK API, select the **Allow API Overriding** check box. When this options is selected, the **API Definitions...** button is enabled.

Types of Customization Available

The code customization that the OSDT allows can be categorized as follows:

- **Customizing Design Attributes**

  These attributes allow you to specify additional information for Statemate model elements. They can also be used as building blocks when defining APIs (see below).

- **Customizing API Definitions**

  These allow you to define the code that should be generated for a wide variety of basic actions. In addition, you can customize the code for static OS configuration and memory management. You can also specify formats for issues such as variable and function naming or file headers/footers.

- **Specifying Related Files**

  This allows you to select the files that you will want to have available in your workarea, for example, makefiles, .h files, and .oil files.

Customizing Design Attributes

Elements in a Statemate model are further defined using design attributes. Every element has attributes that are relevant to that type of element. For example, the design attributes for activities may include Type, Task Run Mode, Generate Function (yes/no). These attributes appear on the Design Attributes tab of the Properties dialog.

For most of the provided OSIs, a default set of design attributes are defined for the various Statemate elements. You can also add new design attributes for an element, using the Attributes Editor, which is part of the OSDT. In addition to allowing you to define new attributes, this editor allows you to modify the various field values for the default design attributes.

The design attributes for an element, both the default attributes and any new ones, can be used as tokens when defining the different APIs with the OSDT, allowing you to include the values of element attributes in the generated code.

For each design attribute, the editor displays:

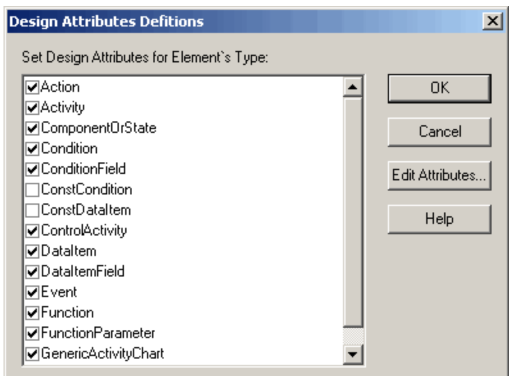
- The basic attribute information, such as name, type, default value, key name.
- The attribute’s dependencies
- Information that should be displayed when Statemate’s Info tool is used.

Note

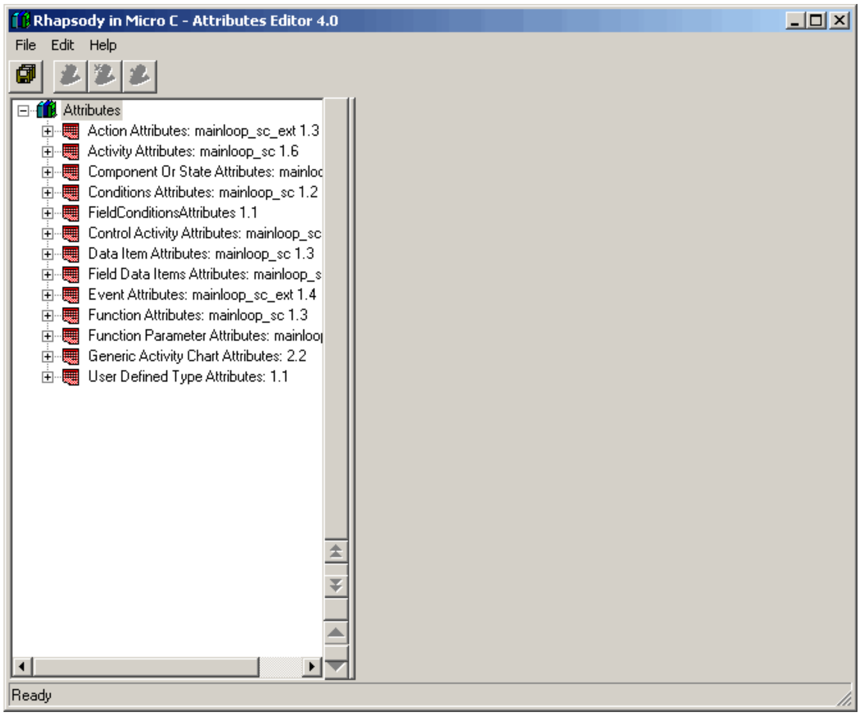
The *key name* is the string that is used when the attribute is used as a token. Since both other attributes and API's may reference an attribute's key value, you will be asked to confirm any changes to an attribute's key name. When such a change is made, the name change will be propagated to any attributes for the same element that have a reference to the attribute.

To edit attribute information, follow these steps:

1. Click **Edit Attributes...** on the main OSDT screen.
2. The Design Attributes Definition dialog is displayed. This screen is not used for editing attribute information, but simply for turning on/off the use of design attributes for specific model elements. If you clear the check box next to an element name, the design attributes will not appear when you display the Properties dialog for elements of this type.



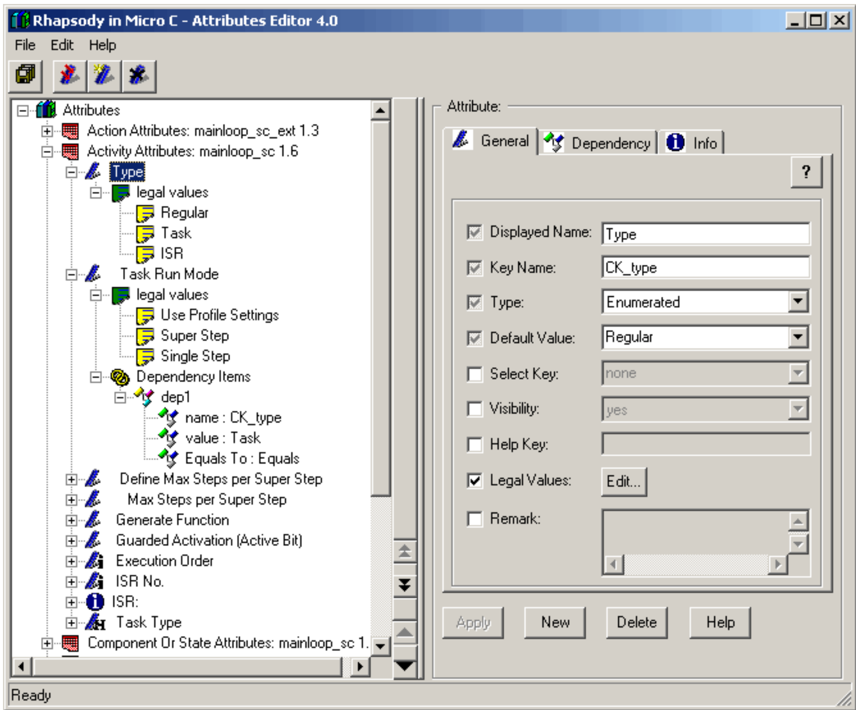
3. In the Design Attributes Definition dialog, click **Edit Attributes...** . The Attribute Editor screen will be displayed. All editing of information is done on the right side of this dialog. The left side serves as a browser, and contains controls that can be used to modify the order in which the attributes appear in the Properties dialog.



To save any changes/additions you have made to the design attributes, select **File > Save All** from the menu, and after you have closed the Attribute Editor dialog, select **File > Save** from the OSDT menu. The changes will not be saved to the OSI if you have only saved them in the Attribute Editor dialog.

Design Attribute Fields

General

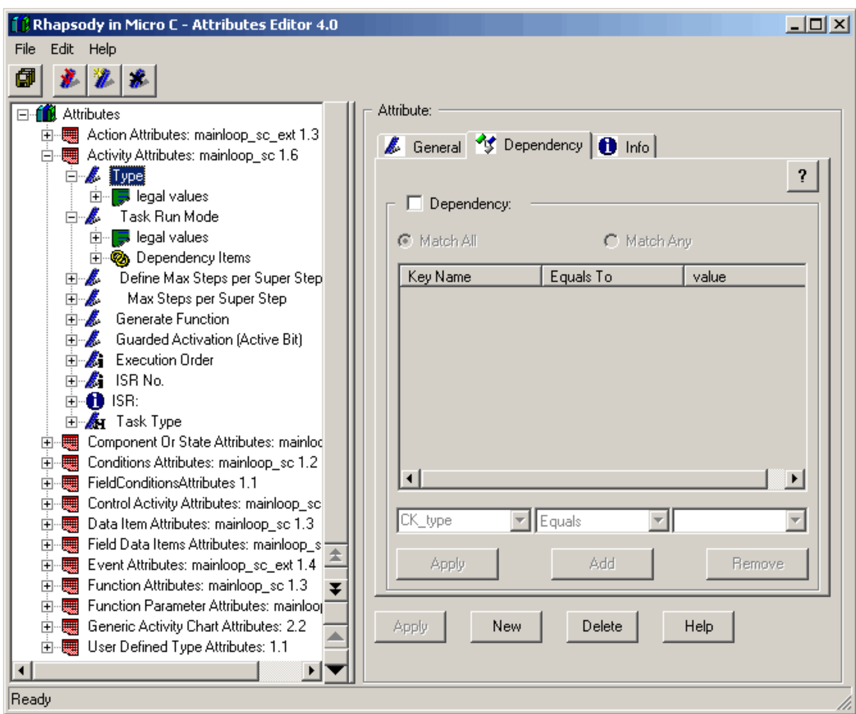


Parameter	Description
Displayed Name	The string that is displayed for the attribute on the Design Attributes tab of the Properties dialog.
Key Name	The string used when using the attribute as a token in an API definition, or as a dependency for another design attribute. This string cannot contain spaces.
Type	The type of value that can be used for this attribute, for example, string, integer, enumerated.
Default Value	The default value to display for the attribute.
Select Key	Allows you to specify a predefined list of values that can be presented to the user when they click the Choose button on the Design Attributes tab, for example, a list of tasks for the design attribute Its Task.

Parameter	Description
Visibility	Specifies whether or not the attribute is displayed on the Design Attributes tab. There are certain situations where you may want to use a hidden attribute—an attribute that does not appear on the Design Attributes tab, but is included in the code.
Legal Values	The possible values to display for attributes of type *enumerated*.
Remark	Field that can be used for describing the attribute. This text only appears here, in the Attribute Editor.

Dependency

The Dependency tab allows you to make the existence of a design attribute dependent upon other attributes such that the design attribute will be used only under certain conditions. If these conditions are not met, the attribute is not included in the generated code and not visible in the Design Attributes tab in the Properties dialog.



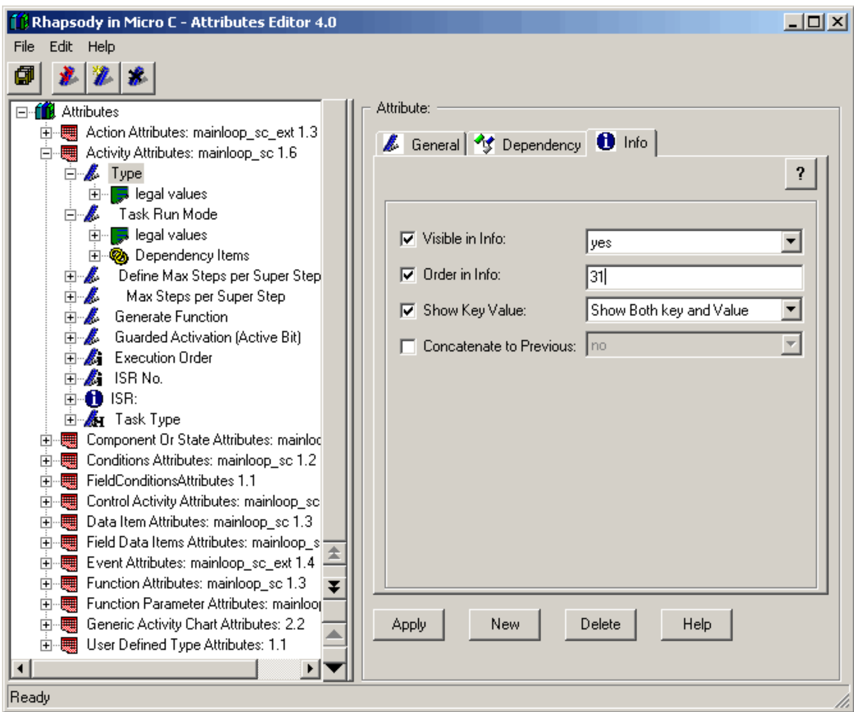
For example, the attribute Task Run Mode will be displayed only if the activity is of type *Task*.

The drop-down lists can be used to compose conditions involving relevant attributes.

You can then specify whether the design attribute is available only if all of the listed conditions are met or if any of the listed conditions are met.

Info

The Info tab allows you to specify whether the design attribute should be included in Statemate's Info dialog when the user displays this dialog for a given element, and, if so, how it should be displayed.



Parameter	Description
Visible in Info	This boolean field is used to indicate whether or not the design attribute should be included in the Info dialog.
Order in Info	You can specify the order in which design attributes are displayed in the Info dialog, by entering an integer in this field. The values used for the different attributes do not have to be sequential.
Show Key Value	This field is used to indicate what should be displayed for this attribute in the Info dialog—the attribute name and value, the attribute value only, or the attribute name only.
Concatenate to Previous	• If you select *yes* for this field, the information for this attribute will be displayed in the same row as the information for the previous attribute. • If you select *no*, the information for this attribute will be displayed on a new line.

Customizing API Definitions

The API definitions are divided into the following categories:

- General API Definitions
- Customizing Code Style
- Customizing Memory Management
- Customizing the Static OS Configuration

The OSDT contains a number of features that facilitate API definition. These features apply to each of the above categories, and will be described in the following section. Afterward, each of the API definition categories will be discussed in detail.

Features that Facilitate API Definition

The following features can be used when defining the APIs:

- Browse Properties from OSDT
- Using Parameters for the Generated Code
- Conditional Expressions in API Definitions

Browse Properties from OSDT

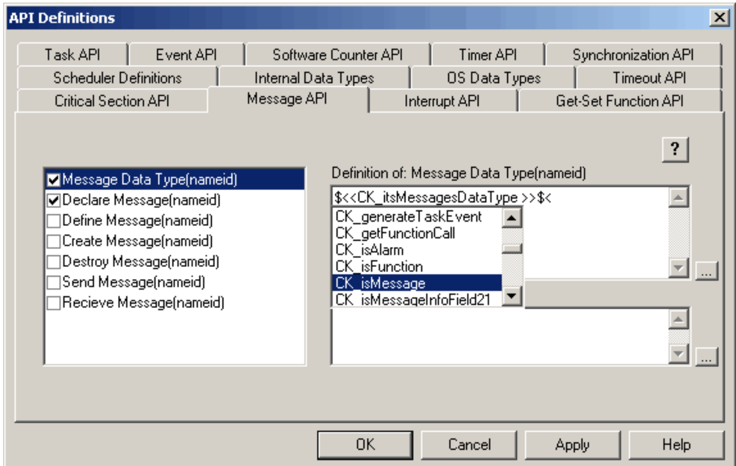
When defining APIs in the OSDT, you can easily access tokens available for use in the definitions.

To see the available tokens, type *$*. The list of available values will be displayed.

The list of available tokens includes the API's formal arguments, and all the design attributes defined for the various elements.

Note

Design attributes have element-level scope (i.e, they are relevant only for the element for which they are defined). Therefore, if a design attribute from a different element is used in an API definition, its value will be an empty string.



Using Parameters for the Generated Code

To represent parameters in the API definitions you provide, you add the prefix “$<” and the suffix “>” to the parameter name, for example:

API name:

Terminate Task(*nameid*)

API Definition in OSDT:

```
TerminateThread ( t_$<nameid>.hndl , 0 );
```

Code that will be generated for a task named T1:

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

A second way to use parameters in the API definition is to use the design attribute value of the element itself. 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
API Name:
Create Task(nameid)
```

API Definition in OSDT:

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

Code that will be generated for a task named T1:

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

A third way to use parameters in the API definition is to use the property value of the element as the API definition. 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 in OSDT:

```
$<<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.

Conditional Expressions in API Definitions

The OSDT also allows the use of conditional expressions in API definitions. This feature allows the inclusion of a certain string if the condition is met and the inclusion of an alternative string if the condition is not met. Basically, the feature mimics the C conditional expression, "? :", although the syntax is slightly different.

The basic syntax is as follows:

?<begin> expression 1 **?<?>** expression 2 **?<:>** expression 3 **?<end>**

If expression 1 evaluates to true, then expression 2 will be used in the API definition; otherwise, expression 3 will be used.

Example of using conditional expressions:

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

In the above example, the string used in the API definition will be expression when yes if **$<prop1>** evaluates to prop1val. Otherwise, the string used will be expression when no.

When defining the condition (expression 1), the following symbols can be used:

?<==> (equal strings)

?<!=> (not equal strings)

?<&&> (logical AND)

?<||> (logical OR)

Expression 2 and expression 3 referred to above can consist of any expression that is legal in the API definition, including additional conditional expressions.

The following is a more complex example, which uses nested conditional expressions.

```
Some prefix 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>
```

Start with the inner conditional expression:

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

This expression will evaluate to `exp 1.1 when yes` if either **$<prop2>** evaluates to "prop2val" or **$<prop2.1>** evaluates to "prop2.1val". If neither of these conditions are met, then the expression will evaluate to `exp 1.2 when no`.

Now look at the outer conditional expression, replacing the result of the inner expression with the string "result of inner conditional expression":

```
?<begin> $<prop1> ?<==> prop1val ?<&&> $<prop1.1> ?<==>

prop1.1val ?<?> result of inner conditional expression ?<:> exp 2 when no
?<end>
```

This expression will evaluate to the result of the inner conditional expression if **$<prop1>** evaluates to prop1val and **$<prop1.1>** evaluates to prop1.1val. Otherwise, the expression will evaluate to `exp 2 when no`.

So, assuming that:

$<prop1> = prop1val

$<prop1.1> = prop1.1val

$<prop2> <> prop2val

$<prop2.1> <> prop2.1val

$<prop3> <> prop3val

The API result will be:

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

General API Definitions

The OSDT allows you to define how code should be generated for the various model elements. These APIs represent the code that will be generated when you use specific elements in your charts.

For example, if you add an activity to an activity diagram, and select “Task” as the type, the code that is generated for activities related to this task will be determined by the APIs you have defined.

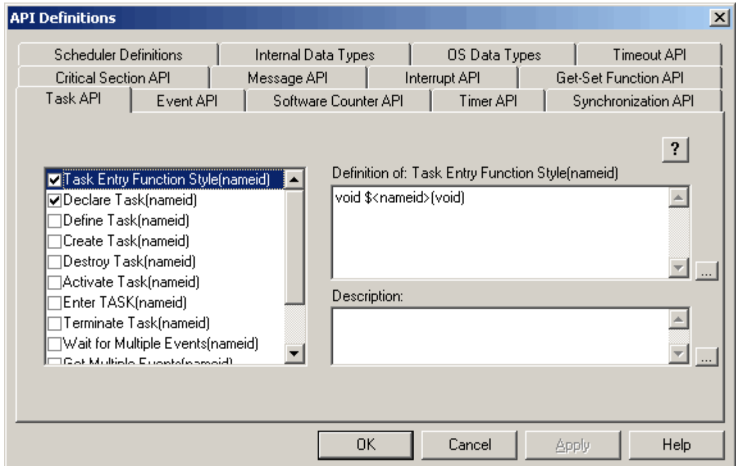
In the case of a Task, you can define the code for activities such as:

- creating the Task
- activating the Task
- destroying the Task

Using the OSDT, you can define the code to generate in connection with the following categories:

- OS data types
- timeouts
- Tasks
- events
- software counters
- timers
- synchronization
- critical sections
- messages
- interrupts
- get-set functions
- queues
- scheduler

In the tables containing the API details, the generated code is based on the definition that appears in the column *Sample Definition*, unless specified otherwise.



OS Data Type APIs

Name	Sample Definition	Description	Where Used	Code Generated

Task Descriptor Name	TASK_H	Defines the Task's (OS Object) Type name.	The definition of this API is used when a Task type name is needed.
			For example, with the tracing function (controlled by a Code Generator option), the type of the parameter that identifies the Task being traced uses this API definition.
				API Definition:
				TASK_H
				Generated Code (type_def.h):
				`#ifdef TRACE_TASK`
				`void traceTask(TASK_H t, char indx);`
				`#endif`

Task Descriptor Data Type	typedef struct TASK_H {HANDLE hndl; DWORD tid; int indx; int evCount; HANDLE eventArray[16];} TASK_H;	Defines the Task's (OS Object) Data Type.	The definition of this API is generated in the file *type_def.h*	The generated code is the API's definition suffixed with an end-line.

Name	Sample Definition	Description	Where Used	Code Generated
Event Descriptor Name	EVENT_H	Defines the Event's (OS Object) Type name.	The definition of this API is used when a Task-Event type name is needed. For example, when using Task-Events in OSEK, a data that holds the Task-Event is allocated with the type name defined by this API.	*API Definition:* EventMaskType *Generated code (type_def.h):* `EventMaskType eventsBuff;`
Event Descriptor Data Type		Defines the Event's (OS Object) Data Type.	The definition of this API is generated in the file *type_def.h*	The generated code is the API's definition suffixed with an end-line.
Software Counter Descriptor Name	SW_COUNTER_H	Defines the Software Counter's (OS Object) Type name.	The definition of this API is used when a Software-Counter type name is needed. For example, when using OSEK OS and Timeouts then this API's definition is used in the Counter Overflow Task to install a timer to invoke this Task again.	*API Definition:* `TickType` *Generated Code (glob_func.c):* `((TickType)6000 * 0.5);`
Software Counter Descriptor Data Type	typedef uint16 SW_COUNTER_H	Defines the Software Counter's (OS Object) Data Type.	The definition of this API is generated in the file *type_def.h*	The generated code is the API's definition suffixed with an end-line.
Timer Descriptor Name	TIMER_H	Defines the Timer's (OS Object) Type name.	NOT USED	
Timer Descriptor Data Type		Defines the Timer's (OS Object) Data Type.	The definition of this API is generated in the file *type_def.h*	The generated code is the API's definition suffixed with an end-line.
Semaphore Descriptor Name	SEM_H	Defines the Semaphore's (OS Object) Type name.	NOT USED	

Name	Sample Definition	Description	Where Used	Code Generated
Semaphore Descriptor Data Type		Defines the Semaphore’s (OS Object) Data Type.	The definition of this API is generated in the file *type_def.h*	The generated code is the API's definition suffixed with an end-line.
Message Descriptor Name	MESSAGE_H	Defines the Message’s (OS Object) Type name.	NOT USED	
Message Descriptor Data Type		Defines the Message’s (OS Object) Data Type.	The definition of this API is generated in the file *type_def.h*	The generated code is the API's definition suffixed with an end-line.
ISR Descriptor Name	ISR_H	Defines the ISR’s (OS Object) Type name.	NOT USED	
ISR Descriptor Data Type	typedef @interrupt void (* ISR_H)(void)	Defines the ISR’s (OS Object) Data Type.	The definition of this API is generated in the file *type_def.h*	The generated code is the API's definition suffixed with an end-line.

Name	Sample Definition	Description	Where Used	Code Generated

Queue Descriptor Data Type(nameid, qType)	typedef struct $<nameid>_qtype{
	uint16 head;
	uint16 tail;
	$<qType>
	q[DEFAULT_QUEUE_SIZE];
	}
	$<nameid>_qtype;
		Defines the data type of a queue.	This API is used when there is data of type Queue in the model.
			The type is generated in the file *type_def.h*
				For Data-Item defined as Queue:
				Name: DI_QUEUE
				Formal Parameter: q_type = unsigned long int
				API Definition:
				`typedef struct $<nameid>_qtype{`
				`uint16 head;`
				`uint16 tail;`
				`$<qType> q[DEFAULT_QUEUE_SIZE];`
				`}`
				`$<nameid>_qtype;`
				Generated code:
				`typedef struct DI_QUEUE_qtype{`
				`uint16 head;`
				`uint16 tail;`
				`unsigned long int q[DEFAULT_QUEUE_SIZE];`
				`}`
				`DI_QUEUE_qtype;`

Name	Sample Definition	Description	Where Used	Code Generated

Queue Data Type Static Init(nameid, qType)	{ 0, /* head */
	0, /* tail */
	{0} /* q */
	}
		Defines the code for the static initialization of the Queue type.	This API is used when statically initializing a Queue typed data.
			The initialization can be in a generic's data allocation, or in a User's data initialization (controlled by a Code Generator option)
				For Data-Item defined as Queue:
				Name: DI_QUEUE
				Formal Parameter: q_type = unsigned long int
				API Definition:
				`{`
				`  0,  /* head */`
				`  0,  /* tail */`
				`  {0} /* q */`
				`}`
				
				Generated code:
				`struct DI_QUEUE_qtype DI_QUEUE = {`
				`  0,  /* head */`
				`  0,  /* tail */`
				`  {0} /* q */`
				`};`

Timeout APIs

<table>
<tr><th>Name</th><th>Sample Definition</th><th>Description</th><th>Where Used</th><th>Code Generated</th></tr>
<tr>
<td>Timeout Install Define (nameid, tmMaskName, tmMaskVal, tmVariableName, tmMaskVarName, tmVariableType, tmCurrentTickName, tmCounterVarName, tmCounterName)</td>
<td>?<begin> $<tmMaskVal> ?<==> ?<?>?<:>#define $<tmMaskName> $<tmMaskVal>
?<end>#define INSTALL_TM_$<nameid>(D, C) \

$<tmMaskVarName> |= $<tmMaskName>; \

$<tmVariableName> = ($<tmVariableType>)$<tmCurrentTickName> + (D); ?<begin> $<tmCounterVarName> ?<==> ?<?>?<:>\

$<tmCounterVarName> = (C);?<end></td>
<td>Definition of the Timeout installation, in the file macro_def.h</td>
<td>This API is used for each of the Timeouts in the model, when generating the macro for installing the Timeout.
The macro is generated in the file macro_def.h</td>
<td>For a Timeout with predefined name: tm_999999998
API's Parameters:
tmMaskName = tm_999999998_TM_MASK
tmMaskVal = 0x01
nameid = tm_999999998
tmMaskVarName = cgTimeoutsMask
tmVariableName = tm_999999998_TIME
tmVariableType = uint32
tmCurrentTickName = currentTick
tmCounterVarName = <empty>
API Definition:
?<begin> $<tmMaskVal> ?<==> ?<?>?<:>#define $<tmMaskName> $<tmMaskVal>
?<end>#define INSTALL_TM_$<nameid>(D, C) \</td>
</tr>
</table>

<table>
<tr><th>Name</th><th>Sample Definition</th><th>Description</th><th>Where Used</th><th>Code Generated</th></tr>
<tr><td>Timeout Install Define
(nameid,
tmMaskName,
tmMaskVal,
tmVariableName,
tmMaskVarName,
tmVariableType,
tmCurrentTickName,
tmCounterVarName,
tmCounterName)

(Continued)</td><td></td><td></td><td></td><td>$<tmMaskVarName> |= $<tmMaskName>; \
$<tmVariableName> = ($<tmVariableType>)$<tmCurrentTickName> + (D); ?<begin> $<tmCounterVarName> ?<==> ?<?>?<:>\
$<tmCounterVarName> = (C);?<end>
Generated Code:
#define tm_999999998_TM_MASK (UNSIGNED_MASK_TYPE)(0x01)
#define INSTALL_TM_tm_999999998(D, C) \

cgTimeoutsMask |= tm_999999998_TM_MASK; \
tm_999999998_TIME = (uint32)currentTick + (D)</td></tr>
</table>

Name	Sample Definition	Description	Where Used	Code Generated

Timeout Install Call(nameid, time, counterIndex, tmCounterName, tmCurrentTickName)	$<tmCurrentTickName> = $<tmCounterName>;
	INSTALL_TIMEOUT($<nameid>, $<time>, $<counterIndex>)
		Definition of the call to the install of a Timeout, in file *<module>.c*	This API is used for each of the Timeouts in the model, when generating the code for installing the Timeout.
			The code for the installation is generated in the file *<module>.c*
				For a Timeout with predefined name: tm_999999998
				API's Parameters:
				tmCurrentTickName = currentTick
				tmCounterName = ms_counter
				nameid = tm_999999998
				time = 1
				counterIndex = ms_counter_INDEX
				API Definition:
				$<tmCurrentTickName> = $<tmCounterName>;
				INSTALL_TIMEOUT($<nameid>, $<time>, $<counterIndex>)
				Generated Code:
				`currentTick = ms_counter;`
				`INSTALL_TIMEOUT(tm_999999998,1,ms_counter_INDEX);`

Name	Sample Definition	Description	Where Used	Code Generated

Timeout Test on Expiration Call(nameid, tmCurrentTickName, tmCounterName, tmEventBuffer, tmCounterIndex, genContextVar)	$<tmCurrentTickName> = $<tmCounterName>;
	$<nameid>($<tmCounterName>, &$<tmEventBuffer>, $<tmCounterIndex>?<begin>$<genContextVar> ?<!=> ?<?>, $<genContextVar>?<:>?<end>)
		Definition of the call to the Timeouts Dispatch function, in the file *<module>.c*	This API is used in the Task's code frame.
			The API defines the code that calls to the Timeouts Dispatch Function, in *<module>.c*
				For a Timeout with predefined name: tm_999999998
				API's Parameters:
				tmCurrentTickName = currentTick
				tmCounterName = ms_counter
				nameid = tm_999999998
				time = 1
				counterIndex = ms_counter_INDEX
				tmEventBuffer = cgDoubleBufferOld_T1
				genContextVar = <empty>
				API Definition:
				$<tmCurrentTickName> = $<tmCounterName>;
				$<nameid>($<tmCounterName>, &$<tmEventBuffer>, $<tmCounterIndex>?<begin> $<genContextVar> ?<!=> ?<?>, $<genContextVar>?<:>?<end>);
				Generated Code:
				`currentTick = ms_counter;`
				`genTmEvent_T1(currentTick, &cgDoubleBufferOld_T1, ms_counter_INDEX);`

Name	Sample Definition	Description	Where Used	Code Generated

Timeout Test on Expiration Define(nameid, tmCurrentTickType, tmCurrentTickName, tmEventBuffType, tmEventBuffName, tmCounterIndexType, tmCounterIndex, genContextVar, timeoutList)	@<for>
	@<timeoutList>
	@<begin>
	if(?<begin> $<$<timeoutList>_counterIndex>?<==>?<?>?<:>$<tmCounterIndex> == $<$<timeoutList>_counterIndex> &&
	
	?<end>$<$<timeoutList>_buffMask> & $<timeoutList>_TM_MASK && $<tmCurrentTickName> >= $<$<timeoutList>_timeVar>) {
	
	GEN_IN_BUFF($<timeoutList>, $<tmEventBuffName>);
	
	$<$<timeoutList>_buffMask> &= ~$<timeoutList>_TM_MASK;
	};
	@<end>}
		Definition of the Timeouts Dispatch function, in the file *<module>.c*	This API is used when generating the Timeout Dispatch Function.
			A separate Timeout Dispatch Function is generated for each of the Tasks, ISR's and Generic Activity generated as function.
			This API uses a syntax capability that handles lists of Design-Attributes.
			Lists of Design-Attributes are simply a set of Design-Attributes with identical name, in the same element.
			The syntax uses the character "@" and the "begin" and "end" tokens to define the boundaries of the definition related to the list of Design-Attributes.
				For an Activity defined to be a Task:
				Name: T1
				With Timeouts: tm_999999998, tm_999999997
				API Definition: Same as the example definition
				API's Parameters:
				nameid = T1
				timeoutList = tm_999999998, tm_999999997
				tmCurrentTickType = uint16
				tmCurrentTickName = currentTickVar
				tmEventBuffType = cgDoubleBufferType_T1*
				tmEventBuffName = buff
				tmCounterIndexType = uint8
				tmCounterIndex = counterIndex
				genContextVar = <empty>
				Generated Code:
				`void`
				`genTmEvent_T1(uint16 currentTickVar, cgDoubleBufferType_T1 * buff, uint8 counterIndex)`
				`{`
				`if(counterIndex == ms_counter_INDEX &&`
				`cgTimeoutsMask & tm_999999998_TM_MASK && currentTickVar >= tm_999999998_TIME) {`

Name	Sample Definition	Description	Where Used	Code Generated

Timeout Test on Expiration Define(nameid, tmCurrentTickType, tmCurrentTickName, tmEventBuffType, tmEventBuffName, tmCounterIndexType , tmCounterIndex, genContextVar, timeoutList)
 	
(Continued)
				`GEN_IN_BUFF(tm_999999998, buff);`
				`    cgTimeoutsMask &= ~tm_999999998_TM_MASK;`
				`  };`
				`  if(counterIndex == sec_counter_INDEX &&`
				`    cgTimeoutsMask & tm_999999997_TM_MASK && currentTickVar >= tm_999999997_TIME) {`
				`GEN_IN_BUFF(tm_999999997, buff);`
				`    cgTimeoutsMask &= ~tm_999999997_TM_MASK;`
				`  };`
				`}`

Name	Sample Definition	Description	Where Used	Code Generated

Timeout Test on Expiration Declare(nameid, tmCurrentTickType, tmCurrentTickName, tmEventBuffType, tmEventBuffName, tmCounterIndexType , tmCounterIndex, genContextVar)	void $<nameid>($<tmCurrentTickType> $<tmCurrentTickName>, $<tmEventBuffType>* $<tmEventBuffName>, $<tmCounterIndexType> $<tmCounterIndex>?<begin>$<genContextVar> ?<!=> ?<?>, $<genContextVar>?<:>?<end>);	The forward declaration of the Timeouts Dispatch function, in the file *type_def.h*	This API is used when generating the Timeout Dispatch Function's forward declaration.
			The forward declaration is generated in the file *type_def.h*
				For an Activity defined to be a Task:
				Name: T1
				API's Parameters:
				tmCurrentTickType = uint16
				tmCurrentTickName = currentTickVar
				tmEventBuffType = cgDoubleBufferType_T1*
				tmEventBuffName = buff
				tmCounterIndexType = uint8
				tmCounterIndex = counterIndex
				API definition:
				void $<nameid>($<tmCurrentTickType> $<tmCurrentTickName>, $<tmEventBuffType>* $<tmEventBuffName>, $<tmCounterIndexType> $<tmCounterIndex>?<begin> $<genContextVar> ?<!=> ?<?>, $<genContextVar>?<:>?<end>);
				Generated Code:
				`void genTmEvent_T1(uint16 currentTickVar, cgDoubleBufferType_T1 * buff, uint8 counterIndex);`

Name	Sample Definition	Description	Where Used	Code Generated

Timeout Overflow Code Per Task(tmMasks, tmDispatchFunc, counterMaxAllowdVal, tmEventBuffName, counterIndex, genContextVar, timeoutList, timeoutVarType, counterValueType)	if($<tmMasks>)
	{
	
	$<tmDispatchFunc>($<counterMaxAllowdVal>, &($<tmEventBuffName>), $<counterIndex>?<begin>$<genContextVar> ?<!=> ?<?>, $<genContextVar>?<:>?<end>);
	
	@<for>@<timeoutList>
	@<begin>?<begin>
	$<$<timeoutList>_counterIndex>?<!=> ?<?>
	if($<$<timeoutList>_counterIndex> == $<counterIndex>)
	?<:>?<end>
	$<$<timeoutList>_timeoutVar> -= ($<timeoutVarType>) 1 + (($<counterValueType>)$<counterMaxAllowdVal>);
	@<end>
	}
		The overflow code related to a specific task with timeouts, put in the Overflow-Task, in the file *glob_func.c*	This API is used in the Counter's overflow task, in the section where the overflow task calls the Timeout Dispatch Functions for each of the related Timeouts.
			This API defines the code for calling the Timeout Dispatch Function for Timeouts that related to the same Task, ISR or generic activity instance.
			This code is generated in the file *glob_func.c*
				For an Activity defined to be a Task:
				Name: T1
				API's Parameters:
				tmMasks = cgTimeoutsMask != 0
				tmDispatchFunc = genTmEvent_T1
				counterMaxAllowdVal = (uint16) -1
				tmEventBuffName = cgDoubleBufferNew_T1
				counterIndex = ms_counter_INDEX
				genContextVar = <empty>
				timeoutList = tm_999999998 (list with only 1 Timeout)
				timeoutVarType = uint32
				counterValueType = (uint16)
				counterMaxAllowdVal = (uint16) -1
				API definition:
				Same as the example definition
				Generated Code:
				`if(cgTimeoutsMask != 0) {`
				
				`genTmEvent_T1((uint16) -1, &(cgDoubleBufferNew_T1), ms_counter_INDEX);`
				
				`tm_999999998_TIME -= (uint32) 1 + ((uint16)(uint16) -1);`
				`}`

Task APIs

Name	Sample Definition	Description	Where Used	Code Generated
Task Entry Function Style(nameid)	void $<nameid>(void)	Defines the style in which a Task's (OS Object) function entry is generated.	This API is used when generating the code for the definition of a Task's code frame. It defines the Task's code frame function's prototype and return type. The code is generated in the file <module>.c	For an Activity defined to be a Task: Name: T1 API Definition: `void $<nameid>(void)` Generated code: `void T1(void)`
Declare Task(nameid)	extern void $<nameid>(void);	Definition of the code for declaring a Task (OS Object).	This API is used when generating the code for declaring a Task. The code is generated in the file *type_def.h*	For an Activity defined to be a Task: Name: T1 API Definition: `extern void $<nameid>(void);` Generated code: `extern void T1(void);`
Define Task(nameid)		Definition of the code for defining a Task (OS Object).	This API is used when generating the code for defining a Task. The code is generated in the file *glob_dat.c*	For an Activity defined to be a Task: Name: T1 API Definition: `DEFINE_TASK($<nameid>);` Generated code: `DEFINE_TASK(T1);`

Name	Sample Definition	Description	Where Used	Code Generated
Create Task(nameid)		Definition of the code for creating a Task (OS Object).	This API is used when generating the code for creating a Task. The code is generated in the file *glob_dat.c*, in the function: `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For an Activity defined to be a Task: Name: T1 API Definition: `CREATE_TASK($nameid);` Generated code: `CREATE_TASK(T1);`
Destroy Task(nameid)		Definition of the code for destroying a Task (OS Object).	This API is used when generating the code for destroying a Task. The code is generated in the file *glob_dat.c*, in the function: `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For an Activity defined to be a Task: Name: T1 API Definition: `DESTROY($nameid);` Generated code: `DESTROY(T1);`

Name	Sample Definition	Description	Where Used	Code Generated
Activate Task(nameid)		Defines the code for activating a Task (OS Object)	This API is used when a Task needs to be activated. For example, each Activity has a set of macros defined in the file *macro_def.h*. On of this macro is the start_activity_<act-name> macro which, in case that the Activity is defined as a Task, includes the code for Activating the Task, defined by this API. The definition of this API will be generated in the file <module>.c	For an Activity defined to be a Task: Name: T1 API Definition: `ActivateTask($nameid);` Generated code: `#define start_activity_T1 { cgGlobalFlags |= BITAC_T1; ctivateTask(T1); }`

Enter TASK(nameid)		Defines the code for entering a Task (OS Object).	This API defines the code that is put at the beginning of the Task's code frame.
			The code is generated just after the code from the API: Task/ISR Beginning Code(nameid, profileName)
			The definition of this API will be generated in the file <module>.c
				For an Activity defined to be a Task:
				Name: T1
				API Definition: `EnteredTask($nameid);`
				Generated code:
				`void T1(void)`
				`{`
				`EnteredTask(T1);`
				...

Name	Sample Definition	Description	Where Used	Code Generated

Terminate Task(nameid)		Defines the code for terminating a task (OS Object).	This API defines the code that is put at the end of the Task's code frame.
			The definition of this API will be generated in the file <module>.c
				For an Activity defined to be a Task:
				Name: T1
				API Definition: `TerminateTask($nameid);`
				Generated code:
				`void T1(void)`
				`{`
				`...`
				`TerminateTask(T1);`
				`}`

Wait for Multiple Events(nameid)		Defines the code for waiting for multiple Events by a Task (OS Object).	This API defines the code for waiting on multiple Task Events.
			For Tasks that have Task-Event related to them, the code will be generated in the Task's code frame, after the code for the Task's logic.
			The definition of this API will be generated in the file <module>.c
				For an Activity defined to be a Task:
				Name: T1
				API Definition: `WaitMultipleEvent($nameid);`
				Generated code:
				`void T1(void)`
				`{`
				…
				`<Tasklogic>`
				`WaitMultipleEvent(T1);`
				`}`

<table>
<tr><th>Name</th><th>Sample Definition</th><th>Description</th><th>Where Used</th><th>Code Generated</th></tr>
<tr><td>Get Multiple Events(nameid)</td><td></td><td>Defines the code for getting multiple Events by a Task (OS Object).</td><td>This API defines the code for getting the generated Task Events for multiple Task Events.
For Tasks that have Task-Event related to them, the code of this API will be generated in the Task's code frame, after the code for the API: WaitMultipleEvent()
The definition of this API will be generated in the file <module>.c</td><td>For an Activity defined to be a Task:
Name: T1
API Definition: <code>GetMultipleEvent($nameid);</code>
Generated code:
<code>void T1(void)</code>
<code>{</code>
<code>…</code>
<code><Tasklogic></code>
<code>WaitMultipleEvent(T1);</code>
<code>GetMultipleEvent(T1);</code>
<code>...</code>
<code>}</code></td></tr>
</table>

Name	Sample Definition	Description	Where Used	Code Generated

Clear Multiple Events(nameid)		Defines the code for clearing multiple Events by a Task (OS Object).	This API defines the code for clearing multiple Task Events.
			For Tasks that have Task-Event related to them, the code of this API will be generated in the Task's code frame, after the code for the API: Get Multiple Events().
			The definition of this API will be generated in the file <module>.c
				For an Activity defined to be a Task:
				Name: T1
				API Definition: ClearMultipleEvent($nameid);
				Generated code:
				void T1(void)
				{
				…
				<Tasklogic>
				WaitMultipleEvent(T1);
				GetMultipleEvent(T1);
				ClearMultipleEvent(T1);
				...
				}

Get Task ID(nameid)	$<nameid>	Defines the code for getting a Task's (OS Object) ID.	NOT USED	

Event APIs

Name	Sample Description	Definition	Where Used	Code Generated
Declare Event(nameid, itstaskid)		Defines the code for declaring an Event (OS Object).	This API is used when generating the code for declaring a Task Event The code is generated in the file *type_def.h*	For an Event defined to be a Task-Event. Name: EV1 API definition: `DECLARE_EVENT($nameid);` Generated Code: `DECLARE_EVENT(EV1);`
Define Event(nameid, itstaskid)		Defines the code for defining an Event (OS Object).	This API is used when generating the code for defining a Task Event. The code is generated in the file *glob_dat.c*	For an Event defined to be a Task-Event. Name: EV1 API definition: `DEFINE_EVENT($nameid);` Generated Code: `DEFINE_EVENT(EV1);`
Create Event(nameid, itstaskid)		Defines the code for creating an Event (OS Object).	This API is used when generating the code for creating a Task Event. The code is generated in the file *glob_dat.c*, in the function: `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For an Event defined to be a Task-Event. Name: EV1 API definition: `CREATE_EVENT($nameid);` Generated Code: `CREATE_EVENT(EV1);`

Name	Sample Description	Definition	Where Used	Code Generated
Destroy Event(nameid, itstaskid)		Defines the code for destroying an Event (OS Object).	This API is used when generating the code for destroying a Task Event. The code is generated in the file *glob_dat.c*, in the function `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For an Event defined to be a Task-Event. Name: EV1 API definition: `DESTROY_EVENT($nameid);` Generated Code: `DESTROY_EVENT(EV1);`
Clear Event(nameid, itstaskid)		Defines the code for clearing an Event (OS object).	This API defines the code for clearing a single Task Event. The definition of this API will be used only if there is no definition for ClearMultipleEvent () For Tasks that have Task-Event related to them, the code of this API will be generated in the Task's code frame, after the code for the API: Get Event(). The definition of this API will be generated in the file <module>.c	For an Event defined to be a Task-Event. Name: EV1 API definition: `CLEAR_EVENT($nameid);` Generated Code: `CLEAR_EVENT(EV1);`

Name	Sample Description	Definition	Where Used	Code Generated
Get Event(nameid, itstaskid)		Defines the code for getting an Event (OS Object).	This API defines the code for clearing a single Task Event. The definition of this API will be used only if there is no definition for GetMultipleEvent() For Tasks that have Task-Event related to them, the code of this API will be generated in the Task's code frame, after the code for the API: Wait Event(). The definition of this API will be generated in the file <module>.c	For an Event defined to be a Task-Event. Name: EV1 API definition: `GET_EVENT($nameid);` Generated Code: `GET_EVENT(EV1);`
Set Event(nameid, itstaskid)		Defines the code for setting an Event (OS Object).	This API defines the code for setting a single Task Event. The definition of this API will be used in the definition of the GENERATE_EVENT macro for Events that are defined to be a Task Event. The macro is generated in the file *macro_def.h*	For an Event defined to be a Task-Event. Name: EV1 API definition: `SET_EVENT($nameid);` Generated Code: `SET_EVENT(EV1);`
Wait Event(nameid, itstaskid)		Defines the code for waiting for an Event (OS Object).	NOT USED	

Name	Sample Description	Definition	Where Used	Code Generated
Test for Event(nameid, itstaskid)		Defines the code for testing an Event (OS Object).	This API defines the code for testing if a single Task Event was generated. For Tasks that have Task-Event related to them, the code of this API will be generated in the Task's code frame, after the code for the API: Get Event().	For an Event defined to be a Task-Event. Name: EV1 API definition: `TEST_EVENT( $nameid);` Generated Code: `TEST_EVENT( EV1);`

Software Counter APIs

Name	Sample Definition	Description	Where Used	Code Generated

Software Counter Value Type	uint16	Defines the type of a variable that holds a Software Counter value.	This API is used when the type of a Software Counter is required.
			For example, the Timeouts Dispatch Function requires the “Time" that was read from the counter. The value of the "Time" is passed to the function using a "Software Counter Value Type".
				For an Activity defined to be a Task:
				Name: T1
				API Definition: `uint16`
				Generated code:
				`void`
				`genTmEvent_T1(uint16 currentTickVar, cgDoubleBufferType_T1* buff, uint8 counterIndex)`
				`{`
				…

Software Counter Max Value(nameid)	(uint16) -1	Defines the maximum allowed value for a Software Counter, after which it overflows.	The API is used when handling the overflow of counters, used with Timeouts.
			In the Timeouts Dispatch Function, generated in the file *glob_func.c*, there is a code section that subtracts the max value of the software counter from the value stored in the Timeout's time variable.
				For counter named: ms_counter
				API Definition: `(uint16) -1`
				`void onms_counter_OVERFLOW(void)`
				`{`
				`...`
				`tm_999999998_TIME -= (uint32) 1 + ((uint16)(uint16) -1);`
				`...`
				`}`

Name	Sample Definition	Description	Where Used	Code Generated

Get Software Counter Value(nameid, value)	$<value> = $<nameid>;	Defines the code for retrieving the current value of a Software Counter.	This API is used when the value of a Software counter is needed.
			For example, the Timeouts Dispatch Function requires the current "Time" of the "Software Counter" to test for the expired Timeouts.
			This API is used to retrieve the "current value", just before calling the ""Timeouts Dispatch Function"", in a Task's ISR's or Generic Activity generated as Function, Code frame, in the file: <module>.c.
				For an Activity defined to be a Task:
				Name: T1
				API's Parameters:
				value = currentTick
				nameid = T1
				API Definition:
				$<value> = $<nameid>;
				Generated code:
				currentTick = ms_counter;
				genTmEvent_T1(currentTick, &/ *Old*/ cgDoubleBufferOld_T1, ms_counter_INDEX);

Declare Software Counter(nameid)	extern SW_COUNTER_H $<nameid>;	Defines the code for declaring a Software Counter (OS Object).	This API is used when generating the code for declaring a Software Counter The code is generated in the file *type_def.h*	For a Software Counter: Name: ms_counter API definition: DECLARE_S_COUNTER($nameid); Generated Code: DECLARE_S_COUNTER(ms_counter);

Name	Sample Definition	Description	Where Used	Code Generated
Define Software Counter(nameid)	SW_COUNTER_H $<nameid>;	Defines the code for defining a Software Counter (OS Object).	This API is used when generating the code for defining a Software Counter. This code is generated in the file *glob_dat.c*	For a Software Counter: Name: ms_counter API definition: `DEFINE_S_COUNTER($nameid);` Generated Code: `DEFINE_S_COUNTER(ms_counter);`
Create Software Counter(nameid)		Defines the code for creating a Software Counter (OS Object).	This API is used when generating the code for creating a Software Counter. This code is generated in the file *glob_dat.c*, in the function `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For a Software Counter: Name: ms_counter API definition: `CREATE_S_COUNTER($nameid);` Generated Code: `CREATE_S_COUNTER(ms_counter);`
Destroy Software Counter(nameid)		Defines the code for destroying a Software Counter (OS Object).	This API is used when generating the code for destroying a Software Counter. This code is generated in the file *glob_dat.c*, in the function `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For a Software Counter: Name: ms_counter API definition: `DESTROY_S_COUNTER($nameid);` Generated Code: `DESTROY_S_COUNTER(ms_counter);`

Timer APIs

Name	Sample Definition	Description	Where Used	Code Generated
Timer Value Type		Defines the type of a variable that holds a Timer value.	NOT USED	
Timer Max Value(nameid)	(uint8) -1	Defines the maximum allowed value of a Timer, after which it overflows.	NOT USED	
Get Timer Value(nameid, value, itstaskid)		Defines the code for retrieving the current value of a Timer.	NOT USED	
Declare Timer(nameid, itstaskid)	?<begin>$<CK_defineThisAlarm> ?<==> yes ?<?>extern void tim_hndlr_$<nameid>(); ?<:> ?<end>	Defines the code for declaring a Timer (OS Object).	This API is used when generating the code for declaring a Timer The code is generated in the file *type_def.h*	For a Timer: Name: timer_10ms API definition: `DECLARE_TIMER($nameid);` Generated Code: `DECLARE_TIMER(timer_10ms);`

Define Timer(nameid, itstaskid)	"?<begin>$<CK_defineThisAlarm>
	?<==> yes
	?<?>void tim_hndlr_$<nameid> ()
	{
	GENERATE_EVENT($<nameid>);
	}
	?<:> ?<end>
		Defines the code for defining a Timer (OS Object).	This API is used when generating the code for defining a Software Timer.
			The code is generated in the file *glob_dat.c*
				For a Timer:
				Name: timer_10ms
				API definition: `DEFINE_TIMER($nameid);`
				Generated Code: `DEFINE_TIMER(timer_10ms);`

Name	Sample Definition	Description	Where Used	Code Generated
Create Timer(nameid, itstaskid)		Defines the code for creating a Timer (OS Object).	This API is used when generating the code for creating a Software Timer. This code is generated in the file *glob_dat.c*, in the function `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For a Timer: Name: timer_10ms API definition: `CREATE_TIMER($nameid);` Generated Code: `CREATE_TIMER(timer_10ms);`
Destroy Timer(nameid, itstaskid)		Defines the code for destroying a Timer (OS Object).	This API is used when generating the code for destroying a Software Timer. This code is generated in the file *glob_dat.c*, in the function `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For a Timer: Name: timer_10ms API definition: `DESTROY_TIMER($nameid);` Generated Code: `DESTROY_TIMER(timer_10ms);`

Name	Sample Definition	Description	Where Used	Code Generated

Install Relative Timer(nameid, value, cycle, counter, itstaskid)	install_mainloop_timer((unsigned long int) ($<value>) + (?<begin>$<counter> ?<==> ?<?>ms_counter?<:>$<counter>?<end>), tim_hndlr_$<nameid>);	Defines the code for installing a Timer (OS Object) with a relative value.	This API is used when installing a Timer using relative time.
			For example, when using a scheduling operation with an Event, that uses a Timer for this operation. The generated code will use the definition of this API to install the timer.
			The code for this operation will be generated in the file <module>.c
				For an Event defined to be a Task-Event.
				Name: EV
				API's Parameters:
				value = 100
				counter = ms_counter
				nameid = EV
				API definition:
				`install_mainloop_rel_timer((unsigned long int) ($<value>) + (?<begin>$<counter> ?<==> ?<?>ms_counter?<:>$<counter>?<end>), tim_hndlr_$<nameid>);`
				Generated Code:
				`install_mainloop_rel_timer((unsigned long int) (100) + (ms_counter), tim_hndlr_EV);`

Name	Sample Definition	Description	Where Used	Code Generated

Install Absolute Timer(nameid, value, cycle, counter, itstaskid)		Defines the code for installing a Timer (OS Object) with an absolute value.	This API is used when installing a Timer using absolute time.
			For example, when using a scheduling operation with an Event that uses a Timer for this operation. The generated code will use the definition of this API to install the timer.
			The code for this operation will be generated in the file <module>.c
				For an Event defined to be a Task-Event.
				Name: EV
				API's Parameters:
				value = 100
				counter = ms_counter
				nameid = EV
				API definition:
				`install_mainloop_abs_timer((unsigned long int)($<value>) + (?<begin>$<counter>?<==>?<?>ms_counter?<:>$<counter>?<end>), tim_hndlr_$<nameid>);`
				Generated Code:
				`install_mainloop_abs_timer((unsigned long int) (100) + (ms_counter), tim_hndlr_EV);`

Name	Sample Definition	Description	Where Used	Code Generated
Clear Timer(nameid, itstaskid)		Defines the code for clearing a Timer (OS Object).	This API is used in the code section that handles the Timer expiration. If the Timer was tested to be expired, the Event related to this Timer is generated. Just before generating the Event, the timer is cleared.	For an Event defined to be a Task-Event, using a Timer. Name: EV API's Parameters: nameid = EV API definition: `Clear_Timer (tim_hndlr_ $nameid)` Generated Code: `Clear_Timer (tim_hndlr_ EV)`
Test for Timer Expiration(nameid, itstaskid)		Defines the code for testing a Timer (OS Object) for expiration.	This API is used in the code section that handles the Timer expiration. This API definition is used in order to find out if a timer has expired. The code related to this API is generated in the file <module>.c	For an Event defined to be a Task-Event, using a Timer. Name: EV API's Parameters: nameid = EV API definition: `TEST_TIMER( tim_hndlr_$ nameid)` Generated Code: `TEST_TIMER( tim_hndlr_E V)`

Name	Sample Definition	Description	Where Used	Code Generated
Cancel Timer(nameid, itstaskid)		Defines the code for canceling a Timer.	This API is used just before installing the Timer. If the Event related to the timer has the Design-Attribute: "CK_cancelBefore Set" with the value "yes", then the definition of this API is put before the timer installation code.	For an Event defined to be a Task-Event, using a Timer. Name: EV API's Parameters: nameid = EV API definition: `TEST_TIMER(tim_hndlr_$nameid)` Generated Code: `TEST_TIMER(tim_hndlr_EV)`

Synchronization APIs

Name	Sample Definition	Description	Where Used	Code Generated
Declare Semaphore(nameid)	extern SEM_H $<nameid>;	Defines the code for declaring a Semaphore (OS Object).	This API is used when generating the code for declaring a Timer This code is generated in the file *type_def.h*	For a Condition defined to be a Semaphore: Name: COND1 API definition: `extern SEM_H $<nameid>;` Generated Code: `extern SEM_H COND1;`
Define Semaphore(nameid)	SEM_H $<nameid>;	Defines the code for defining a Semaphore (OS Object).	This API is used when generating the code for defining a Software Timer. This code is generated in the file *glob_dat.c*	For a Condition defined to be a Semaphore: Name: COND1 API definition: `extern SEM_H $<nameid>;` Generated Code: `DECLARE_TIMER(timer_10ms);`
Create Semaphore(nameid)	$<nameid> = CreateSemaphore();	Defines the code for creating a Semaphore (OS Object).	This API is used when generating the code for creating a Software Timer. This code is generated in the file *glob_dat.c*, in the function: `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For a Condition defined to be a Semaphore: Name: COND1 API definition: `$<nameid> = CreateSemaphore();` Generated Code: `COND1 = CreateSemaphore();`

Name	Sample Definition	Description	Where Used	Code Generated
Destroy Semaphore(nameid)	DestroySemaphore ($<nameid>);	Defines the code for destroying a Semaphore (OS Object).	This API is used when generating the code for destroying a Software Timer. This code is generated in the file *glob_dat.c*, in the function: `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For a Condition defined to be a Semaphore: Name: COND1 API definition: `DestroySemaphore($<nameid>);` Generated Code: `DestroySemaphore(COND1);`
Wait Semaphore(nameid)	WaitSemaphore($<nameid>);	Defines the code for waiting on a Semaphore (OS Object).	This API's definition is used as the generated code for the operator: "get" on a Condition defined to be a Semaphore.	For a Condition defined to be a Semaphore: Name: COND1 API definition: `WaitSemaphore($<nameid>);` Generated Code: `WaitSemaphore(COND1 );`
Release Semaphore(nameid)	ReleaseSemaphore($<nameid>);	Defines the code for releasing a Semaphore (OS Object).	This API's definition is used as the generated code for the operator: "release" on a Condition defined to be a Semaphore.	For a Condition defined to be a Semaphore: Name: COND1 API definition: `ReleaseSemaphore($<nameid>);` Generated Code: `ReleaseSemaphore(COND1);`

Critical Section APIs

Name	Sample Definition	Description	Where Used	Code Generated
Declare Critical Section(nameid)	extern CRITICAL_SECTION $<nameid>;	Defines the code for declaring a Critical Section (OS Object).	This API is used when generating the code for declaring a Critical Section. This code is generated in the file *type_def.h*	For a Critical Section named: critical_section: `extern CRITICAL_SECTION critical_section;`
Define Critical Section(nameid)	CRITICAL_SECTION $<nameid>;	Defines the code for defining a Critical Section (OS Object).	This API is used when generating the code for defining a Critical Section. This code is generated in the file *glob_dat.c*	For a Critical Section named: critical_section: `CRITICAL_SECTION critical_section;`
Create Critical Section(nameid)	$<nameid> = CreateCriticalSection();	Defines the code for creating a Critical Section (OS Object).	This API is used when generating the code for creating a Critical Section. This code is generated in the file *glob_dat.c*, in the function `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For a Critical Section named: critical_section: `critical_section = CreateCriticalSection();`

Name	Sample Definition	Description	Where Used	Code Generated
Destroy Critical Section(nameid)	DestroyCriticalSection($<nameid>);	Defines the code for destroying a Critical Section (OS Object).	This API is used when generating the code for destroying a Critical Section. This code is generated in the file *glob_dat.c*, in the function `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For a Critical Section named: critical_section: `DestroyCriticalSection(critical_section );`
Enter Critical Section(nameid)	EnterCriticalSection($<nameid>);	Defines the code for entering a Critical Section (OS Object).	Used when entering a critical section. For example, swapping the double buffered buffers of a Task, may require guarding this code section. The code generator uses this API just before swapping the buffers.	For a Critical Section named: critical_section: `EnterCriticalSection(critical_section );`
End Critical Section(nameid)	EndCriticalSection($<nameid>);	Defines the code for ending a Critical Section (OS Object).	Used when exiting a critical section. For example, swapping the double buffered buffers of a Task, may require guarding this code section. The code generator uses this API just after the swapping the buffers.	For a Critical Section named: critical_section: `EndCriticalSection(critical_section );`

Message APIs

Name	Sample Definition	Description	Where Used	Code Generated
Message Data Type(nameid)	typedef $<CK_itsMessagesDataType > MESSAGE_$<nameid>;	Defines the code for the Type of a Message (OS Object).	Used for Data-Item which is defined to be a Message, to generate the type definition for the message. Generated in the file *type_def.h*	For a Data-Item named DI_MSG, defined as a Message: Design Attribute: CK_itsMessagesDataType = long int: `typedef long int MESSAGE_DI_MSG;`
Declare Message(nameid)	DECLARE_MSG($<nameid>)	Defines the code for declaring a Message (OS Object).	Used when generating the code for declaring a Message. The code is generated in the file *type_def.h*	For a Data-Item named DI_MSG, defined as a Message: `DECLARE_MSG(DI_MSG)`
Define Message(nameid)	DEFINE_MSG($<nameid>)	Defines the code for defining a Message (OS Object).	Used when generating the code for defining a Message. The code is generated in the file *glob_dat.c*	For a Data-Item named DI_MSG, defined as a Message: `DEFINE_MSG(DI_MSG)`
Create Message(nameid)	CREATE_MSG($<nameid>)	Defines the code for creating a Message (OS Object).	Used when generating the code for creating a Message The code is generated in the file *glob_dat.c*, in the function: `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For a Data-Item named DI_MSG, defined as a Message: `CREATE_MSG(DI_MSG)`

Name	Sample Definition	Description	Where Used	Code Generated
Destroy Message(nameid)	DESTROY_MSG($<nameid>)	Defines the code for destroying a Message (OS Object).	Used when generating the code for destroying a Message The code is generated in the file *glob_dat.c*, in the function `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For a Data-Item named DI_MSG, defined as a Message: `DESTROY_MSG(DI_MSG)`
Send Message(nameid)	SendMessage(<$nameid>)	Defines the code for sending a Message (OS Object).	This API's definition is used as the generated code for the operator "send" on a Data-Item defined to be a Message.	For a Data-Item named DI_MSG, defined as a Message: `SendMessage(DI_MSG)`
Receive Message(nameid)	ReceiveMessage(<$nameid>)	Defines the code for receiving a Message (OS Object).	This API's definition is used as the generated code for the operator "receive" on a Data-Item defined to be a Message	For a Data-Item named DI_MSG, defined as a Message: `ReceiveMessage(DI_MSG)`

Interrupt APIs

Name	Sample Definition	Description	Where Used	Code Generated
ISR Entry Function Style(nameid)	@interrupt void $<nameid>()	Defines the style in which an ISR's (OS Object) entry function is generated.	Used when generating the code for the definition of an ISR's code frame. Defines the ISR's code frame function's prototype and return type. This code is generated in the file <module>.c	For an Activity defined to be an ISR, named T1: `@interrupt void $<T1>()`
Declare ISR(nameid)	extern void $<nameid>();	Defines the code for declaring an ISR (OS Object).	Used when generating the code for declaring an ISR. This code is generated in the file *type_def.h*	For an Activity named T1, defined to be an ISR: `extern void $<T1>();`
Define ISR(nameid)	DEFINE_ISR($nameid);	Defines the code for defining an ISR (OS Object).	Used when generating the code for defining an ISR. This code is generated in the file *glob_dat.c*	For an Activity named T1, defined to be an ISR: `DEFINE_ISR(T1);`
Create ISR(nameid)	CREATE_ISR($nameid);	Defines the code for creating an ISR (OS Object).	Used when generating the code for creating an ISR. This code is generated in the file *glob_dat.c*, in the function `on_startup_code`. The function `on_startup_code` is called at the startup of the generated application.	For an Activity named T1, defined to be an ISR: `CREATE_ISR(T1);`

Name	Sample Definition	Description	Where Used	Code Generated
Destroy ISR(nameid)	DESTROY_ISR($nameid);	Defines the code for destroying an ISR (OS Object).	Used when generating the code for destroying an ISR. This code is generated in the file *glob_dat.c*, in the function `on_exit_code`. The function `on_exit_code` is called at the end of the generated application.	For an Activity named T1, defined to be an ISR: `DESTROY_ISR(T1);`

Enter ISR(nameid)	EnterIsr(<$nameid>	Defines the code for entering an ISR (OS Object).	Defines the code that is put at the beginning of the ISR's code frame.
			This code is generated just after the code from the API: Task/ISR Beginning Code(nameid, profileName)
			The definition of this API will be generated in the file <module>.c
				For an Activity named T1, defined to be an ISR:
				`void T1(void)`
				`{`
				`EnterIsr(T1);`
				...

Leave ISR(nameid)	LeaveIsr(<$nameid>	Defines the code for leaving an ISR (OS Object).	This API defines the code that is put at the end of the ISR's code frame.
			The definition of this API will be generated in the file <module>.c
				For an Activity named T1, defined to be an ISR:
				Generated code:
				`void T1(void)`
				`{`
				`...`
				`LeaveIsr(T1);`
				`}`

Disable Interrupt(mask)	DisableIsr(<$nameid>	Defines the code for disabling an ISR (OS Object).	NOT USED	
Enable Interrupt(mask)	EnableIsr(<$nameid>	Defines the code for Enabling an ISR (OS Object).	NOT USED	

Name	Sample Definition	Description	Where Used	Code Generated
Interrupt Mask Data Type	uint8	Defines the type for an ISR's interrupt mask.	NOT USED	

Scheduler Definition APIs

A scheduler file is a file that includes the Tasks defined in the application.

If this option is selected, the code generator will look for the specified scheduler file, and insert the list of Tasks to be performed.

- **File Name**

  By default, the tasks are added to the file <Profile Name>.c. If you want the generator to use a different file, enter the name of the file in the text box or use the **...** button to select the file.

- **Scheduler Key Words**
  - These are the words that demarcate the beginning and end of the task list. The code generator requires the end keyword as well because it removes the tasks that were previously included in the list.
  - The keywords used can include tokens enclosed with "$<" and ">" (API definition notation).
  - When using "$<token>" as part of the keyword, the name of the keyword can vary between groups of tasks, depending on the data of the task. The tokens that can be used are:
    - $<nameid> - The name of the task
    - $<Design-Attribute-Name> - Any design attribute defined for the task.
  - For example:

    If a task has a design attribute named "CK_timeSlice" with three possible values ("10ms", "40ms" and "100ms"), and the begin and end keywords are defined as */* User $<CK_timeSlice> Tasks Begin */*, and */* User $<CK_timeSlice> Tasks End */*respectively, then tasks with CK_timeSlice = 40ms will be put in the scheduler file between the keywords */* User 40msTasks Begin */* and the keyword */* User 40ms Tasks End */*

- **Task Separator**

  The delimiter to use to separate the individual tasks in the list. When using a custom delimiter, you can use "\n" to specify a new line.

Get-Set Function APIs

Name	Sample Definition	Description	Where Used	Code Generated
Get Function Declare(nameid, returntype, argType, argName)	$<returntype> get_$<nameid>_CB($<argType> $<argName>);	Forward declaration of the 'Get' function (placed in type_def.h)	Defines the forward declaration of the "Get" function for a Statemate element, like a Data-Item or a Condition. The API's definition is generated in the file *type_def.h*	For Data-Item named DI, of type Real: `double get_DI_CB( void);`

Get Function Define(nameid, returntype, argType, argName, getElemCode)	"$<returntype> get_$<nameid>_CB($<argType> $<argName>){ $<getElemCode> }"	Definition of the 'Get' function (placed in glob_func.c)	Defines the definition of the "Get" function for a Statemate element, like a Data-Item or a Condition.
			This API's definition is generated in the file *glob_func.c*
				For Data-Item named DI, of type Real:
				`double`
				`get_DI_CB(void){`
				`return DI;`
				`}`

Get Function Name(nameid)	get_$<nameid>_CB	Name of the 'Get' function, used for the Panel/Test-Driver Bindings (placed in glob_func.c)	Used when the name of the "Get" function is required. For example, when using the Test-Driver instrumentation, and the "Get Value" option, this API is used to initialize the Test-Driver data.	For Data-Item named DI of type integer: `testDriver_addKey(3, "DI", INTEGER_DATA_ITEM_T, DI_CB, get_DI_CB);`
Get Array Element Function Declare(nameid, returntype, argType, argName)	$<returntype> get_$<nameid>_CB($<argType> $<argName>);	Forward declaration of the 'Get' function (placed in type_def.h)	For elements of type array, this is used in the file *type_def.h* to generate the forward declaration of the "Get" function for the array's element.	For Data-Item named DI_ARR of type array of integers: `int get_DI_ARR_ArrElm_CB(int index);`

Name	Sample Definition	Description	Where Used	Code Generated

Get Array Element Function Define(nameid, returntype, argType, argName, getElemCode)	"$<returntype> get_$<nameid>_CB($<argType> $<argName>){
	$<getElemCode>
	}
		Definition of the 'Get' function for array's element(placed in glob_func.c)	For elements of type array, this is used in the file *glob_func.c* to generate the definition of the "Get" function for the array's element.	For Data-Item named DI_ARR of type array of integers:
				`int`
				`get_DI_ARR_CB_ArrElm(int index)`
				`{`
				`return(DI_ARR[index - DI_ARR_INDEX_SHIFT]);`
				`}`

Set Function Declare(nameid, returntype, argType, argName)	$<returntype> $<nameid>_CB($<argType> $<argName>);	Forward declaration of the 'Set' function (placed in type_def.h)	Defines the forward declaration of the "Set" function for a Statemate element, like a Data-Item or a Condition.
			The API's definition is generated in the file *type_def.h*
				For Data-Item named DI, of type Real:
				`void`
				`DI_CB(doubel di_val);`

Set Function Define(nameid, returntype, argType, argName, tstDrvInst, setElemCode)	"$<returntype> $<nameid>_CB($<argType> $<argName>){
	$<tstDrvInst>
	$<setElemCode>
	}
		Definition of the 'Set' function (placed in glob_func.c)	Defines the definition of the "Set" function for a Statemate element, like a Data-Item or a Condition.
			This API's definition is generated in the file `glob_func.c`
				For Data-Item named DI, of type Real:
				`void`
				`DI_CB(double di_val){`
				`DI = di_val;`
				`}`

Set Function Name(nameid)	$<nameid>_CB	Name of the 'Set' function, used for the Panel/Test-Driver Bindings (placed in glob_func.c)	Used when the name of the "Set" function is required. For example, when using the Test-Driver instrumentation, this API is used to initialize the Test-Driver data.	For Data-Item named DI of type integer: `testDriver_addKey(3, "DI", INTEGER_DATA_ITEM_T, DI_CB, get_DI_CB);`

Name	Sample Definition	Description	Where Used	Code Generated
Set Array Element Function Declare(nameid, returntype, argType, arrIndexArgType, arrIndexArgName, argName)	$<returntype> $<nameid>_CB($<argType> $<argName>, $<arrIndexArgType> $<arrIndexArgName>);	Forward declaration of the 'Set' function (placed in type_def.h)	For elements of type array, this is used in the file *type_def.h* to generate the forward declaration of the "Set" function for the array's element.	For Data-Item named DI_ARR of type array of integers: `void DI_ARR_CB_ArrElm(double di_val, int index);`

Set Array Element Function Define(nameid, returntype, argType, argName, arrIndexArgType, arrIndexArgName, tstDrvInst, setElemCode)	"$<returntype> $<nameid>_CB($<argType> $<argName>, $<arrIndexArgType> $<arrIndexArgName>){
	$<tstDrvInst>
	$<setElemCode>
	}
		Definition of the 'Set' function (placed in glob_func.c)	For elements of type array, this is used in the file *glob_func.c* to generate the definition of the "Set" function for the array's element.	For Data-Item named DI_ARR of type array of integers:
				`void`
				`DI_ARR_CB_ArrElm(double di_val, int index){`
				`DI_ARR[index - DI_ARR_INDEX_SHIFT] = di_val;`
				`}`

Queue APIs

Name	Sample Definition	Description	Where Used	Code Generated

Queue Put(nameid, elName)	{
	int tmpH =
	($<nameid>.tail+1)
	%
	DEFAULT_QUEUE
	_SIZE;
	if(tmpH !=
	$<nameid>.head){
	$<nameid>.tail =
	tmpH;
	$<nameid>.q[$<na
	meid>.tail] =
	$<elName>;
	};
	}
		Put <elName> at the end of queue <nameid>.	The generated code for the operation put!() for elements of type Queue.
			Generated in the file <module>.c, or *glob_func.c* for user's functions.
				For Data-Item DI_Q of type Queue, and DI_IN to put into the queue:
				`{`
				`int tmpH =`
				`(DI_Q.tail+1)`
				`%`
				`DEFAULT_QUEUE_`
				`SIZE;`
				`if(tmpH !=`
				`DI_Q.head){`
				`DI_Q.tail =`
				`tmpH;`
				`DI_Q.q[DI_Q] =`
				`DI_IN;`
				`};`
				`}`

Queue Urgent Put(nameid, elName)	{
	int tmpH =
	($<nameid>.head -
	1 +
	DEFAULT_QUEUE
	_SIZE) %
	DEFAULT_QUEUE
	_SIZE;
	if(tmpH !=
	$<nameid>.tail){
	$<nameid>.q[$<na
	meid>.head] =
	$<elName>;
	$<nameid>.head =
	tmpH;
	};
	}
		Put <elName> at the beginning of queue <nameid>.	The generated code for the operation uput!() for elements of type Queue.
			Generated in the file <module>.c, or *glob_func.c* for user's functions.
				For Data-Item DI_Q of type Queue, and DI_IN to put into the queue:
				`{`
				`int tmpH =`
				`(DI_Q.head - 1`
				`+`
				`DEFAULT_QUEUE_`
				`SIZE) %`
				`DEFAULT_QUEUE_`
				`SIZE;`
				`if(tmpH !=`
				`DI_Q.tail){`
				`DI_Q.q[DI_Q.he`
				`ad] = DI_IN;`
				`DI_Q.head =`
				`tmpH;`
				`};`
				`}`

Name	Sample Definition	Description	Where Used	Code Generated

Queue Get(nameid, elName, statElName)	?<begin> $<statElName> ?<==> ?<?>{
	if($<nameid>.tail != $<nameid>.head){
	$<nameid>.head = ($<nameid>.head + 1) % DEFAULT_QUEUE_SIZE;
	$<elName> = $<nameid>.q[$<nameid>.head];
	};
	}?<:>{
	if($<nameid>.tail != $<nameid>.head){
	$<nameid>.head = ($<nameid>.head + 1) % DEFAULT_QUEUE_SIZE;
	$<elName> = $<nameid>.q[$<nameid>.head];
	MAKE_TRUE($<statElName>);
	}
	else{ MAKE_FALSE($<statElName>);
	};
	}?<end>
		Remove value from front of queue <nameid>, return value in <elName>, and set <statElName> successful.	The generated code for the operation get!() for elements of type Queue.
			Generated in the file <module>.c, or *glob_func.c* for user's functions.
				For Data-Item DI_Q of type Queue, and DI_OUT to set with the operation's value, and STAT as the status variable name:
				{
				if(DI_Q.tail != DI_Q.head){
				DI_Q.head = (DI_Q.head + 1) % DEFAULT_QUEUE_SIZE;
				DI_OUT = $<nameid>.q[DI_Q.head];
				MAKE_TRUE(STAT);
				}
				else{
				MAKE_FALSE(STAT);
				};
				}

Name	Sample Definition	Description	Where Used	Code Generated

Queue Peek(nameid, elName, statElName)	%=%
	?<begin> $<statElName> ?<==> ?<?>{
	if($<nameid>.tail != $<nameid>.head){
	$<elName> = $<nameid>.q[($<nameid>.head+1) % DEFAULT_QUEUE_SIZE];
	};
	}?<:>{
	if($<nameid>.tail != $<nameid>.head){
	$<elName> = $<nameid>.q[($<nameid>.head+1) % DEFAULT_QUEUE_SIZE];
	MAKE_TRUE($<statElName>);
	}
	else{
	MAKE_FALSE($<statElName>);
	};
	}?<end>
		Copy value from front of queue <nameid>, return value in <elName>, and set <statElName> successful.	The generated code for the operation peek!() for elements of type Queue.
			Generated in the file <module>.c, or *glob_func.c* for user's functions.
				For Data-Item DI_Q of type Queue, and DI_OUT to set with the operation's value, and STAT as the status variable name:
				`{`
				`if(DI_Q.tail != DI_Q.head){`
				`DI_OUT = DI_Q.q[(DI_Q.head+1) % DEFAULT_QUEUE_SIZE];`
				`MAKE_TRUE(STAT);`
				`}`
				`else{`
				`MAKE_FALSE(STAT);`
				`};`
				`}`

Queue Flush(nameid)	$<nameid>.head = 0;
	$<nameid>.tail = 0;
		Remove all elements from queue <nameid>	The generated code for the operation fl!() for elements of type Queue.
			Generated in the file <module>.c, or *glob_func.c* for user's functions.
				For Data-Item DI_Q of type Queue:
				`DI_Q.head = 0;`
				`DI_Q.tail = 0;`

Name	Sample Definition	Description	Where Used	Code Generated
Queue Length(nameid)	($<nameid>.head <= $<nameid>.tail ? $<nameid>.tail - $<nameid>.head : DEFAULT_QUEUE_SIZE - ($<nameid>.head - $<nameid>.tail))	Return length of Queue <nameid>	The generated code for the operation q_length() for elements of type Queue. Generated in the file <module>.c, or *glob_func.c* for user's functions.	For Data-Item DI_Q of type Queue: `(DI_Q.head <= DI_Q.tail ? DI_Q.tail - DI_Q.head : DEFAULT_QUEUE_SIZE - (DI_Q.head - DI_Q.tail))`

Internal Data Types APIs

Name	Sample Definition	Description	Where Used	Code Generated
Condition Buffer User-Type Type	uint8	Defines the type of User-Define-Type used when generating the conditions in 'Buffer per Condition' mode.	Conditions that are generated in a separate buffer each, use the type which is defined by this API. The definition of this API is generated in the file *type_def.h* - for Conditions related to a Task/ISR or a Generic Activity generated as function. The definition of this API is generated in the files *glob_dat* (forward declaration) and *glob_dat.c* (declaration) - for Condition in the "global" scope.	For Condition named COND1, using a separate buffer: `uint8 COND1;`

Name	Sample Definition	Description	Where Used	Code Generated
Event Buffer Type()	uint8	Defines the type of buffer used when generating the events in 'Buffer per Event' mode.	Events that are generated in a separate buffer each, use the type which is defined by this API. The definition of this API is used in the file *type_def.h*	For Event named EV1, using a separate buffer: `uint8 EV1;`

Default Signed Integer Type()	int	Defines the default integer signed type. This overrides the definition in the Code-Generator property-sheet.	Element defined to be of type Integer or Bit array will have a definition of its 'signed type' in the generated code. The definition may be a macro in the file *macro_def.h*, or a typedef in the file *type_def.h* (depending on a profile setting).
			This type is used with the Arithmetical bitwise operations (ASHL, ASHR), to cast the value of the element to the Element's "signed type".
			The definition of this API is used for an Element only if there are no specific definitions of the Integer signed type already defined in its Design-Attributes.
				For a Data-Item named DI, of type Integer
				Design-Attribute "Integer Signed Type": signed long int
				`#define STYPE_DI signed long int (in macro_def.h)`
				`typedef signed long int STYPE_DI (in type_def.h)`

Name	Sample Definition	Description	Where Used	Code Generated

Default Unsigned Integer Type()	unsigned int	Defines the default integer unsigned type.
		This overrides the definition in the Code-Generator property-sheet.
			Element defined to be of type Integer or Bit array will have a definition of its 'unsigned type' in the generated code. The definition may be a macro in the file *macro_def.h*, or a typedef in the file *type_def.h* (depending on a profile setting).
			This type is used with the Arithmetical bitwise operations (LSHL, LSHR), to cast the value of the element to the Element's "unsigned type".
			The definition of this API is used for an Element only if there are no specific definitions of the Integer unsigned type already defined in its Design-Attributes.
				For a Data-Item named DI, of type Integer
				Design-Attribute "Integer Unsigned Type": unsigned long int
				
				`#define STYPE_DI unsigned long int (in macro_def.h )`
				`typedef unsigned long int STYPE_DI (in type_def.h)`

Name	Sample Definition	Description	Where Used	Code Generated
Default Floating Point Type()	double	Defines the default floating point type. This overrides the definition in the Code-Generator property-sheet.	"Element defined to be of type Real, and which is not defining a specific data type in its Design-Attributes will use the default floating data type. The default floating data type defined by this API overrides the default floating point data type defined in the Code Generator's profile. The definition of this API is used when declaring the Elements data, in glob_dat.c (declaration) and glob_dat.h (extern declaration)."	"For a Data-Item named DI_REAL, of type Real: double DI_REAL; (in glob_dat.c) extern double DI_REAL; (in glob_dat.h)"

Name	Sample Definition	Description	Where Used	Code Generated

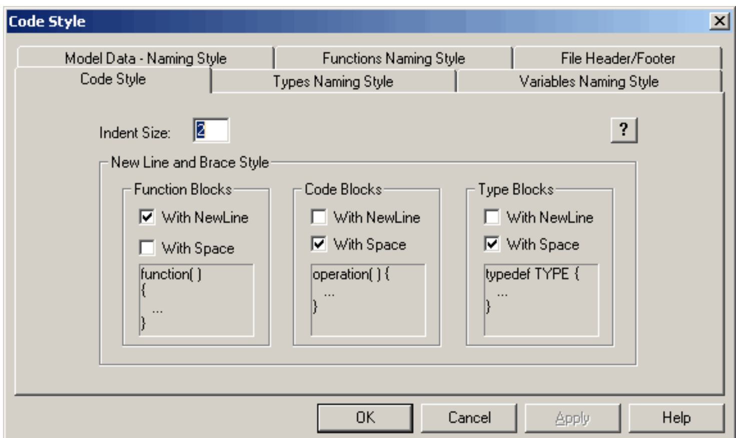
Bit Field Type()	unsigned int	Defines the data-type used for conditions and bit data-items, when generating conditions/bits without macros. This overrides the definition in the Code-Generator property-sheet.	Conditions and Data-Items of type Bit are combined into buffers.
			When using the Code Generator's option for generating Conditions in expressions without using macros ("Use Macros for > Conditions" - unchecked), the type definition of the Conditions/Bits buffers uses a bit field data type.
			Each of the fields in the bit fields is of type which can be defined by this API.
			The data type is defined in the file *type_def.h* for Bits/ Conditions that relate to a TASK/ ISR or a User-Defined-Type, and in the file *glob_dat.c* and *glob_dat.h* for Bits/ Conditions that relate to the global scope.
				For Conditions COND1 and COND2:
				`struct {`
				`unsigned int cond1_:1; /* GENERAL_CONTROL:cond1 */`
				`unsigned int cond2_:1; /* GENERAL_CONTROL:cond2 */`
				`} cg_BitsConditions;`

Default char Type()	char	Defines the default ""char"" type. *Example*: The tool will generate the following data allocation for Data-Item DI_STR defined as string with length=23: <Default char Type() Definition> di_str_[23];	Defines the default data type for elements of type char or char* (strings). For example a Data-Item defined to be a string will use this API to allocate its data.	For Data-Item named DI_STR or type string, with length of 10: `char DI_STR[10];`

Customizing Code Style

The OSDT allows you to modify a large number of settings that affect the code style. These settings are accessed by clicking **Code Style...** in the main OSDT window, and they are categorized as follows:

- Code Style
- Types Naming Style
- Variables Naming Style
- Model Data - Naming Style
- Functions Naming Style
- File Header/Footer



Code Style

This category includes the following settings, which affect the visual appearance of the code:

- Indent Size
- New Line and Brace Style

  These settings determine whether the opening brace appears on the same line as the preceding code or on a new line. There are separate settings for:

  - Function Blocks
  - Code Blocks
  - Type Blocks

Types Naming Style

Name	Sample Definition	Description	Where Used	Code Generated

Single Buffer Type Prefix(activityNameid)	cgSingleBufferType_	Defines the prefix of the Single Buffer type, for Task/ISR	All data which is defined as "Single Buffered" and that relates to a Task/ ISR is combined into a structure. The structure's name is combined from the prefix defined by this API, and from the name of the Task/ISR.
			The type is defined in the file *type_def.h*
				For an Activity defined to be a Task:
				Name: T1
				API Definition: `cgSingleBufferType_`
				Generated Code:
				`struct cgSingleBufferType_T1_type {`
				...
				`};`

Double Buffer Type Prefix(activityNameid)	cgDoubleBufferType_	Defines the prefix of the Double Buffer type, for Task/ISR	All data which is defined as "Double Buffered" and that relates to a Task/ ISR is combined into a structure. The structure's name is combined from the prefix defined by this API, and from the name of the Task/ISR.
			The type is defined in the file *type_def.h*
				For an Activity defined to be a Task:
				Name: T1
				API Definition: `cgDoubleBufferType_`
				Generated Code:
				`struct cgDoubleBufferType_T1_type {`
				...
				`};`

Name	Sample Definition	Description	Where Used	Code Generated
State Variable Prefix(activityNameid)	StateInfo_	Defines the prefix of the State Variable type, for statechart hierarchy	A control Activity, which represent a Statechart hierarchy is implemented using a single function. The state in which the state machine is in is stored in a variable. For each such Statechart hierarchy a separate type is generated. The name of the type is combined of the prefix defined by this API, and from the name of the Control Activity. The type id defined in the file *type_def.h*	For a Control Activity: Name: FAN_CTRL API Definition: `StateInfo_` Generated code (in type_def.h): `typedef uint8 StateInfo_FAN_CTRL;`

Name	Sample Definition	Description	Where Used	Code Generated

Bits Buffer Type Prefix(udt_or_activity_nameid)	bitsConditionsStruct	Defines the prefix of the Bits and Conditions buffer.
		The formal parameter *udt_or_activity_nameid* is the name of the context element in which the buffer is defined.
		The context element can be either an Activity or a User-Defined-Type.
			Conditions and Data-Items of type Bit are combined into buffers. The number of Bits/ Condition per buffer comes from the selected Word-Size.
			In case there are more Bits/ Conditions than the size of the Word-Size an additional buffer is allocated with the prefix defined in this API and postfixed with an incrementing number.
			The buffers are defined in the file *type_def.h* for Bits/ Conditions that relates to a TASK/ ISR or a User-Defined-Type, and in the files *glob_dat.c* and *glob_dat.h* for Bits/ Conditions that relate to the global scope.
				For Conditions CO1 - CO12 and Data-Items of type "bit" BIT1-BIT12, using word size of 8-Bits:
				API Definition: `bitsConditionsStruct`
				There are total of 26 elements, that require 4 8-Bits buffers.
				Therefore the generated code would be:
				`uint8 bitsConditionsStruct;`
				`uint8 bitsConditionsStruct1;`
				`uint8 bitsConditionsStruct2;`
				`uint8 bitsConditionsStruct3`

Name	Sample Definition	Description	Where Used	Code Generated

Integer Signed Type Macro Prefix()	STYPE_	Defines the prefix of the 'signed type' macro for integer Data-Item , used with bit shifting predefined functions, or other user-defined usages.	Element which is defined to be of type Integer or Bitarray will have a definition of its 'signed type' in the generated code. The definition may be a macro in the file *macro_def.h*, or a typedef in the file *type_def.h* (depending on a profile setting).
			This type is used with the Arithmetical bitwise operations (ASHL, ASHR), to cast the value of the element to the Element's "signed type".
			The definition of the "signed type" is defined in the Element's Design-Attributes.
				For a Data-Item of type: Integer
				Name: DI
				Design-Attribute "Integer Signed Type": signed long int
				Expression: DI = ASHL(DI, 5)
				API Definition: `STYPE_`
				Generated code:
				`#define STYPE_DI signed long int (in macro_def.h)`
				`DI = ((STYPE_DI) DI)<<5; (in <module>.c)`

Name	Sample Definition	Description	Where Used	Code Generated

Integer Unsigned Type Macro Prefix()	TYPE_	Defines the prefix of the 'unsigned type' macro for integer Data-Item , used with bit shifting predefined functions, o r other user-defined usages.	Element which is defined to be of type Integer or Bit array will have a definition of its 'unsigned type' in the generated code. The definition may be a macro in the file *macro_def.h*, or a typedef in the file *type_def.h* (depending on profile settings).
			This type is used with the Logical bitwise operations (LSHL, LSHR), to cast the value of the element to the Element's "unsigned type".
			The definition of the "unsigned type" is defined in the Element's Design-Attributes.
				For a Data-Item of type: Integer
				Name: DI
				Design-Attribute "Integer Unsigned Type": unsigned long int
				Expression: DI = LSHL(DI, 5)
				API Definition: `TYPE_`
				Generated code:
				`#define STYPE_DI signed long int (in macro_def.h)`
				`DI = ((TYPE_DI)DI)<<5; (in <module>.c)`

Name	Sample Definition	Description	Where Used	Code Generated

User Defined Type Record Name Postfix()	_T	Controls the postfix of the name of the structure of a User-Defined-Type or Data-Items defined as Record/Union.
		It also controls the postfix of the structures:
		TestDriver_PreviousValues
		PanelBindings_PreviousValues
		Example:
		The following typedef will be generated for Data-Item DI_REC defined as record with the fields: G1 and G2.
		typedef struct DI_REC_type<Postfix> {
		int G1;
		int G2;
		} DI_REC_type;
			Elements of type Record/Union may have a postfix in the name of their generated type.
			The type is defined in the file *type_def.h*.
			The postfix is used only on the name of the type and not on the typedef'ed name.
				For Data-Item defined as record:
				Name: DI_REC
				Fields : G1 and G2, defined as integers.
				API Definition: _T
				Generated code:
				`typedef struct DI_REC_type_T {`
				`int G1;`
				`int G2;`
				`} DI_REC_type;`

Variables Naming Style

Name	Example Definition	Description	Where Used	Code Generated
Global Flags Buffer	cgGlobalFlags	Controls the naming of the global flags buffers. If more than one buffer is required, the name of the next buffer is postfixed with an incrementing number, starting with 1. (cgGlobalFlags1, cgGlobalFlags2, etc.)	The generated application uses some bit sized data to handle various parts of the implementation (like Bit activation for tasks). All those bits are combined into buffers and are generated in the files *glob_dat.c* (declaration) and *glob_dat.h* (extern declaration).	For an Application using 14 global bits. API definition: `cgGlobalFlags` Word Size: 16-Bits Generated Code: `uint16 cgGlobalFlags;`

Timeouts Mask	cgTimeoutsMask	Controls the naming of the timeout mask buffers. If more than one buffer is required, the name of the next buffer is postfixed with an incrementing number, starting with 1. (cgTimeoutMask1, cgTimeoutMask2, etc.)	The generated application uses bits sized data to indicate that a Timeout is pending.
			All those bits are combined into buffers and are generated in the files *glob_dat.c* (declaration) and *glob_dat.h* (extern declaration)
				For an Application using 14 Timeouts.
				API definition: `cgTimeoutsMask`
				Word Size: 8-Bits
				Generated Code:
				`uint8 cgTimeoutsMask;`
				`uint8 cgTimeoutsMask1;`

Name	Example Definition	Description	Where Used	Code Generated
Timeout Event Name(nameid)	$<nameid>	Controls the naming of the timeout event. The profile name can be added here. *Example:* $<nameid>_$<profileName>	Defines the name that identifies the event that relates to a Timeout. The code generator creates a predefined name for the event, which the user can customize.	For a Timeout with predefined name: tm_999999998 In profile: MY_PROF API Definition: `$<nameid>_$<profileName>` Generated event name: `tm_999999998_MY_PROF`
Timeout Time Variable Name(nameid)	$<nameid>_TIME	Controls the naming of the timeout time variable. The profile name can be added here. *Example:* $<nameid>_TIME_$<profileName>	A Timeout uses a Timeout-Time-Variable to hold the time in which it expires. The name of the Timeout-Time-Variable is used when installing the Timeout, and when testing for Timeout's expiration.	For a Timeout with predefined name: tm_999999998 API Definition: `$<nameid>_TIME` Generated event name: `tm_999999998_TIME`
Timeout Mask Name(nameid)	$<nameid>_TM_MASK	Controls the naming of the timeout mask. The profile name can be added here. *Example:* $<nameid>_TM_MASK_$<profileName>	A timeout uses a bit sized data to indicate that the Timeout was installed and is now pending. All the bits for all the timeouts are combined into buffers. In order to access the relevant bit for a Timeout, the application uses a Mask. The name of the Mask is defined by this API, and is generated in the file *macro_def.h*	For a Timeout with predefined name: tm_999999998 API Definition: `$<nameid>_TM_MASK` Generated code for the mask: `#define tm_999999998_TM_MASK 0x01`

Name	Example Definition	Description	Where Used	Code Generated
Timeout Counter Index Name(nameid)	$<nameid>_counter	Controls the naming of the timeout's related counter index. The profile name can be added here. *Example:* $<nameid>_counter_$<profileName>	When using the optimization "Reuse Timeout Variable Where Possible", more than one Timeout may share the same Timeout-Time-Variable. If those Timeouts are using a different Counter. then it is required to store the actual counter on which the timeout was installed. For this purpose, a variable is allocated that stores the index of the counter. The name of this variable is defined by this API, and is generated in the file *glob_dat.c* (declaration) and *glob_dat.h* (extern declaration).	For a Timeout with predefined name: tm_999999998 API Definition: `$<nameid>_counter` Generated code for the counter's index variable: `uint8 tm_999999998_counter;`
Current Time Variable	currentTick	Controls the naming of the variable that holds the "current time" retrieved from a counter; variable is later passed to functions that require this information.	When retrieving the value of a counter, this API is used to define the name of the variable used to hold this value. The retrieval of the counter's value is done before calling a method that requires the current time of a counter, like when testing for Timeout's expiration.	API definition: `currentTick` Generated Code: `currentTick`

Name	Example Definition	Description	Where Used	Code Generated
Single Buffer Variable Prefix(activityName id)	cgSingleBuffer_	Defines the prefix of the Single Buffer type, for Task/ISR.	Single buffered elements that are related to a Task/ ISR are generated in a structure. This API is used when generating the name of the 'Single Buffer' variable of this Single Buffer type, in the file *glob_dat.c* (declaration) and *glob_dat.h* (extern declaration)	For an Activity defined to be a Task: Name: T1 API Definition: `myCgSingleBuffer_` Generated code: `cgSingleBufferType_T1 myCgSingleBuffer_T1;`

Double Buffer Typedef Prefix(activityName id)	cgDoubleBuffer_	Defines the prefix of the Double Buffer typedef'ed type, for Task/ISR.	Double buffered elements that are related to a Task/ ISR are generated in a structure.
			This API is used when generating the name of the 'Double Buffer' type, in the file *type_def.c*
				For an Activity defined to be a Task:
				Name: T1
				API Definition: `myCgDoubleBuffer_`
				Generated code:
				`struct myCgDoubleBuffer_T1_type {`
				`… (Double buffered data)`
				`};`
				`typedef struct myCgDoubleBuffer_T1_type cgDoubleBufferType_T1;`

Name	Example Definition	Description	Where Used	Code Generated
Double Buffer Next Variable Prefix(activityName id)	cgDoubleBufferNew_	Defines the prefix of the Double Buffer variable which holds the current value, for Task/ISR.	Double buffered elements that are related to a Task/ ISR are generated in a structure. This API is used when generating the name of the 'Double Buffer' variable of this Double Buffer type, in the file *glob_dat.c* (declaration) and *glob_dat.h* (extern declaration). The double buffering of an Element is achieved by using two instances of the Double Buffer type; one is "new" and one is "old". This API is used for prefixing the name of the buffer of the "new" values.	For an Activity defined to be a Task: Name: T1 API Definition: `myCgDoubleBuffer_` Generated code: `extern cgDoubleBufferType_T1 /* New*/ cgDoubleBufferNew_T1;`

Name	Example Definition	Description	Where Used	Code Generated
Double Buffer Current Variable Prefix(activityName id)	cgDoubleBufferOld_	Defines the prefix of the Double Buffer variable which holds the new value, for Task/ISR.	Double buffered elements that are related to a Task/ ISR are generated in a structure. This API is used when generating the name of the 'Double Buffer' variable of this Double Buffer type, in the file *glob_dat.c* (declaration) and ""glob_dat.h"" (extern declaration). The double buffering of an Element is achieved by using two instances of the Double Buffer; type one is "new" and one is "old". This API is used for prefixing the name of the buffer of the "old" values.	For an Activity defined to be a Task: Name: T1 API Definition: `myCgDoubleBuffer_` Generated code: `extern cgDoubleBufferType_T1 /*Old*/ cgDoubleBufferOld_T1;`

Name	Example Definition	Description	Where Used	Code Generated

Events Buffer Prefix(udt_or_activity_nameid)	cg_Events	Controls the naming of the Events buffers. If more than one buffer is required, the name of the next buffer is postfixed with an incrementing number, starting with 1 (cg_Events1, cg_Events2, etc.). The formal parameter *udt_or_activity_nameid* is that name of the context element in which the buffer is defined. The context element can be either an Activity of a User-defined type.	Events use Bit sized data, which are combined into buffers.
			The name of these buffers is defined using this API, and is generated in the file *type_def.h*, inside the Double-Buffer type structure.
				For an Activity defined to be a Task:
				Name: T1
				With 10 Events assigned to this Task, and Word Size defined to be 8-Bits
				API Definition: `cg_Events`
				Generated code:
				`struct cgDoubleBuffer_T1_type {`
				`  ...`
				`  uint8 cg_Events;`
				`  ...`
				`};`

Bits Buffer Prefix(udt_or_activity_nameid)	cg_BitsConditions	Controls the naming of the Bits and Conditions buffer in a Task/ISR or user-defined-type structure, and for globals.
		The formal parameter *udt_or_activity_nameid* is the name of the context element in which the buffer is defined.
		The context element can be either an Activity or a User-defined-type.
			Conditions and Data-Item of type "Bit" use Bit sized data, which are combined into buffers.
			The name of these buffers is defined using this API, and is generated in the file *type_def.h*" inside the Task's/ISR's data type structure (if related to it), or in the files *glob_dat.c* (declaration) and `glob_dat.h` (extern declaration).
				For 10 Conditions that are defined in the "global" scope, and Word Size defined to be 8-Bit.
				API definition: `cg_BitsConditions`
				The generated code in the file *glob_dat.c*:
				`AMCBitsStruct8 cg_BitsConditions;`
				`AMCBitsStruct8 cg_BitsConditions1;`

Name	Example Definition	Description	Where Used	Code Generated
Current State Info Variable Prefix(activityName id)	currentState_	Defines the prefix of the Current State variable, for statecharts.	A Statechart Hierarchy (under a Control-Activity) uses a single variable to hold the state of the state machine. This API defines the prefix of the State variable that holds the current state of the state machine.	For a Control Activity: Name: FAN_CTRL API Definition: `currentState_` Generated code (in glob_dat.c): `StateInfo_FAN_CTRL currentState_FAN_CTRL;`

Stay Same State Info Variable Prefix(activityName id)	staySame_	Defines the prefix of the Stay Same variable, for statecharts.	A Statechart Hierarchy (under a Control-Activity) uses a single variable to hold the state of the state machine.
			This API defines the prefix of the State variable that holds the state of the state machine in favor of the Entering and Exiting reaction implementation.
			This variable is generated in the Statechart's function.
				For a Control Activity:
				Name: FAN_CTRL
				API Definition: `staySame_`
				Generated code (in <module>.c):
				`void`
				`cgDo_FAN_CTRL(void)`
				`{`
				…
				`StateInfo_FAN_CTRL staySame_FAN_CTRL = 0;`
				`...`

Name	Example Definition	Description	Where Used	Code Generated

Next State Info Variable Prefix(activityName id)	nextState_	Defines the prefix of the Next State variable, for statecharts.	A Statechart Hierarchy (under a Control-Activity) uses a single variable to hold the state of the state machine.
			This API defines the prefix of the State variable that holds the state of the state machine in favor of the Entering and Exiting reaction and Statechart's implementation.
			This variable is generated in the Statechart's function.
				For a Control Activity:
				Name: FAN_CTRL
				API Definition: `nextState_`
				Generated code (in <module>.c):
				`void`
				`cgDo_FAN_CTRL(void)`
				`{`
				…
				`StateInfo_FAN_CTRL nextState_FAN_CTRL = 0;`
				`...`

Model Data Naming Style

Name	Example Definition	Description	Where Used	Code Generated
Model Data Prefix()	$RimcPre_$<CK_itsSequence>	Prefixes the name of global model data elements—those elements where the field "Its Task" is *global*. It will be added just before the element model name.	When generating the declaration and extern declaration of a global user data, this API is used to add a prefix for the name of the Element.	For a Data-Item of type Integer: Name: DI Design Attribute: "Its Task" defined to be "global" Design Attribute: "CK_itsSequence" defined to be "2" API Definition: `$RimcPre_$<CK_itsSequence>_` Generated code (glob_dat.c): `int $RimcPre_2_DI;`
Model Data Postfix()	_$<CK_postFix>_data	Postfixes the name of global model data elements—those elements where the field "Its Task" is *global*. It will be added just after the element model name.	When generating the declaration and extern declaration of a global user data, this API is used to add a postfix for the name of the Element.	For a Data-Item of type Integer: Name: DI Design Attribute: "Its Task" defined to be "global" Design Attribute: "CK_postFix" defined to be "post" API Definition: `_$<CK_postFix>_data` Generated code (glob_dat.c): `int DI_post_data;`

Functions Naming Style

Name	Example Definition	Description	Where Used	Code Generated
Check for Timeout Function Name(activityNameid)	genTmEvent_$<activityNameid>	Defines the prefix of the timeout's dispatch function, related to a Task/ ISR or generic activity chart generated as a function.	This API defines the name of the name of the Timeouts dispatch function. The function is generated before the Task code function, in the file <module>.c.	For an Activity defined to be a Task: Name: T1 API Definition: genTmEvent_ $<activityNameid> Generated code: void genTmEvent_ T1(...) { …

Activity Function Name(activityNameid)	cgActivity_$<activityNameid>	Defines the prefix for the name of the function/macro implementing an activity.	Each Activity is translated into a Function/Macro (depending on a profile option).
			The name of the Function/Macro is defined by this API, and generated in the file <module>.c
				For an Activity NOT defined to be a Task:
				Name: ACT1
				API Definition:
				cgActivity_ $<activityNameid>
				Generated code:
				1. As macro:
				#define cgActivity_ ACT1()\
				{\
				...
				}
				2. As function:
				void
				cgActivity_ ACT1(void)
				{
				…
				}

Name	Example Definition	Description	Where Used	Code Generated

Statechart Function Name(activityNameid)	cgDo_$<activityNameid>	Defines the prefix for the name of the function/macro implementing a statechart hierarchy.	Each Control-Activity (with the Statechart hierarchy below it) is translated into a Function/Macro (depending on a profile option).
			The name of the Function/Macro is defined by this API, and generated in the file <module>.c
				For a Control-Activity:
				Name: CTRL1
				API Definition: `cgDo_$<activityNameid>`
				Generated code:
				1. As macro:
				`#define cgDo_CTRL1()\`
				`{\`
				` ...`
				`}`
				2. As function:
				`void`
				`cgDo_CTRL1(void)`
				`{`
				…
				`}"`

Entering Reaction Function Name(activityNameid)	cgEnterActions_$<activityNameid>	Defines the prefix for the name of the Entering Reaction function implementing a statechart hierarch.	The Entering-Reactions for a Statechart hierarchy is generated in a single function.
			This API defines the name of this function.
			The function is generated in the file <module>.c
				For a Control-Activity:
				Name: CTRL1
				API Definition: `cgEnterActions_$<activityNameid>`
				Generated code:
				`void`
				`cgExitActions_GENERAL_CONTROL(...)`
				`{`
				…
				`}`

Name	Example Definition	Description	Where Used	Code Generated

Exiting Reaction Function Name(activityNameid)	cgExitActions_$<activityNameid>	Defines the prefix for the name of the Exiting Reaction function implementing a statechart hierarchy.	The Exiting-Reactions for a Statechart hierarchy is generated in a single function.
			This API defines the name of this function.
			The function is generated in the file <module>.c
				For a Control-Activity:
				Name: CTRL1
				API Definition: `cgExitActions_$<activityNameid>`
				Generated code:
				`void`
				`cgExitActions_GENERAL_CONTROL(...)`
				`{`
				…
				`}`

Generic Chart Function Name(activityNameid)	cgGenericFunc_$<activityNameid>	Defines the prefix for the name of the function implementing a generic activity.	Generic Activity which is implemented as function uses an entry function to the generic's code.
			The name of this function is defined by this API.
			This function is generated in the file g_<Generic-As-Function-Name>.
				For a Generic Activity Chart:
				Name: GEN_ACT1
				API Definition: `cgGenericFunc_$<activityNameid>`
				Generated code:
				`void`
				`cgGenericFunc_GEN_ACT1(GEN_ACT1_GRecord_type * rec)`
				`{`
				…
				`}`

File Header/Footer

Name	Example Definition	Description	Where Used	Code Generated
Generated File Header(profileName, fileName, module_name, genDate, genTime, chartsList, profileOptions, project, workarea, profileVersion)	?<begin>$<chartsList> ?<!=> ?<?>$<chartsList>?<:> ?<end>/* Project: $<project> */ /* Workarea: $<workarea> */ /* Profile Name: $<profileName>, Version: $<profileVersion> */ /* File Name: $<fileName> */ /* Date: $<genDate>, $<genTime> */ ?<begin>$<profileOptions> ?<!=> ?<?>$<profileOptions>?<:>?<end>	Determines the format of the various comments added to each generated file, such as profile name, date, file name, information regarding the charts in the module. Will be affected by the code-generation option: *Additional Model Description*.	This API definition is used at the head of each one of the generated files.	Project: MY_PROJ Workarea: C:\myWa Profile Name: PROF1 Version: New File Name: C:\myWa \prt\prof1\<file-name> Date: Sunday, January 15, 2006, 10:05:25 profileOptions: Optimizations Settings: ... API definition: ?<begin>$<chartsList> ?<!=> ?<?>$<chartsList>?<:> ?<end>/* Project: $<project> */ /* Workarea: $<workarea> */ /* Profile Name: $<profileName>, Version: $<profileVersion> */ /* File Name: $<fileName> */ /* Date: $<genDate>, $<genTime> */ ?<begin>$<profileOptions> ?<!=> ?<?>$<profileOptions> ?<:>?<end> Generated Header: /* Project: MY_PROJ */ /* Workarea: C:\myWa */ /* Profile Name: prof1 Version: New */ /* File Name: C:\myWa \prt\prof1\<file-name> */ /* Date: Sunday, January 15, 2006, 10:05:25 */ /* Optimizations Settings: ...*/

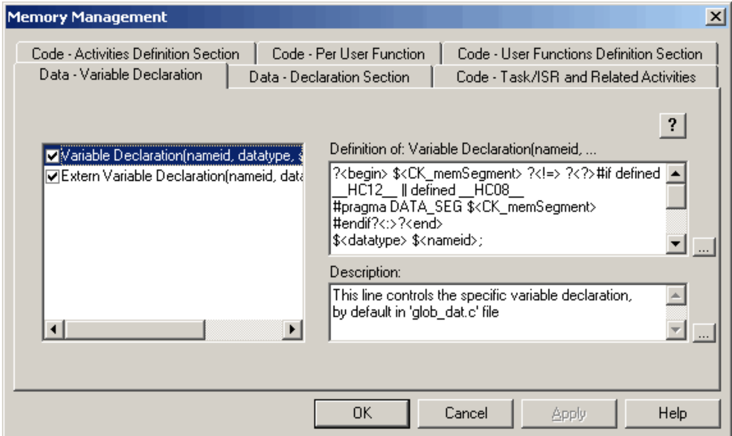
Name	Example Definition	Description	Where Used	Code Generated
Generated File Footer(profileName, fileName, module_name, genDate, genTime, chartsList, profileOptions, Project, Workarea, profileVersion)	/* End of generated file */	Determines the format of the various comments added to each generated file, such as profile name, date, file name, information regarding the charts in the module. Will be affected by the code-generation option: *Additional Model Description*.	This API definition is used at the footer of each one of the generated files.	API definition: `/* End of generated file */` Generated Footer: `/* End of generated file */`
Generated Profile H File Header(profileName, fileName, genDate, genTime, profileOptions, project, workarea, profileVersion)	?<end>/* Project: $<project> */ /* Workarea: $<workarea> */ /* Profile Name: $<profileName>, Version: $<profileVersion> */ /* File Name: $<fileName> */ /* Date: $<genDate>, $<genTime> */ ?<begin>$<profileOptions> ?<!=> ?<?>$<profileOptions> ?<:>?<end>	Determines the format of the various comments added to the generated file <profile-name>.h, such as profile name, date, file name. Will be affected by the code-generation option: *Additional Model Description*.	This API definition is used at the head of the file <profile-name>.h.	See example for API: Generated File Header

Name	Example Definition	Description	Where Used	Code Generated
Generated Profile H File Footer(profileName, fileName, genDate, genTime, profileOptions, Project, Workarea, profileVersion)	"/* End of generated file */	Determines the format of the various comments added to the generated file <profile-name>.h, such as profile name, date, file name. Will be affected by the code-generation option: *Additional Model Description*.	This API definition is used at the footer of the file <profile-name>.h.	See example for API: Generated File Footer
Profile .c File Header(profileName, fileName, genDate, genTime, profileOptions, Project, Workarea, profileVersion)	""#include ""$<profileName>.h	Determines the header of the file: <profile-name>.c.	This API defines the header of the generated file <profile-name>.c	API Definition: #include "$<profileName>.h" Generated Header: #include "$<profileName>.h" ...

Customizing Memory Management

The OSDT allows you to modify settings that affect memory management. These settings are accessed by clicking **Memory Management...** in the main OSDT window. (This option will be grayed out unless the **Memory Management** check box is selected.) These settings are categorized as follows:

- Data—Variable Declaration
- Data—Declaration Section
- Code—Task/ISR and Related Activities
- Code—Activities Definition Section
- Code—Per-User Function
- Code—User Functions Definition Section


Memory Management
Code - Activities Definition Section
Code - Per User Function
Code - User Functions Definition Section
Data - Variable Declaration
Data - Declaration Section
Code - Task/ISR and Related Activities
Variable Declaration(nameid, datatype,
Extern Variable Declaration(nameid, data
Definition of: Variable Declaration(nameid, ...
?<begin> $<CK_memSegment> ?<!=> ?<?>#if defined __HC12__ || defined __HC08__
#pragma DATA_SEG $<CK_memSegment>
#endif?<:>?<end>
$<datatype> $<nameid>;
Description:
This line controls the specific variable declaration, by default in 'glob_dat.c' file
OK
Cancel
Apply
Help

Data—Variable Declaration

<table>
<tr><th>Name</th><th>Sample Definition</th><th>Description</th><th>Where Used</th><th>Code Generated</th></tr>
<tr>
<td>Variable Declaration(nameid, datatype, shortdescription)</td>
<td>?<begin> $<CK_memSegment> ?<!=> ?<?>#if defined __HC12__ || defined __HC08__
#pragma DATA_SEG $<CK_memSegment>
#elif defined(COSMIC12)
#pragma section [$<CK_memSegment>]
#endif
?<:>?<end>$<datatype> $<nameid>;?<begin> $<CK_memSegment> ?<!=> ?<?>
#if defined __HC12__ || defined __HC08__
#pragma DATA_SEG DEFAULT
#elif defined(COSMIC12)
#pragma section []
#endif?<:>?<end></td>
<td>Controls the specific variable declaration, by default in 'glob_dat.c' file.</td>
<td>By default in the file glob_dat.c</td>
<td>For Data-Item DI of type Integer, Design Attribute:
CK_memSegment != <empty-string>
Formal parameters:
datatype = int
nameid = DI

CK_memSegment = CONST_SEG1

#if defined __HC12__ || defined __HC08__
#pragma DATA_SEG CONST_SEG1
#elif defined(COSMIC12)
#pragma section [CONST_SEG1]
#endif
int DI;
#if defined __HC12__ || defined __HC08__
#pragma DATA_SEG DEFAULT
#elif defined(COSMIC12)
#pragma section []
#endif</td>
</tr>
</table>

Name	Sample Definition	Description	Where Used	Code Generated

Extern Variable Declaration(nameid, datatype, shortdescription)	?<begin> $<CK_memSegment> ?<!=> ?<?>#if defined __HC12__ || defined __HC08__
	extern far $<datatype> $<nameid>;
	#else
	?<:>?<end>extern $<datatype> $<nameid>;?<begin> $<CK_memSegment> ?<!=> ?<?>
	#endif?<:>?<end>
		Controls the specific variable extern declaration, by default in 'glob_dat.h' file.	By default in the file *glob_dat.h*	For Data-Item DI of type Integer, Design Attribute:
				
				`#if defined __HC12__ || defined __HC08__`
				`extern far int DI;`
				`#else`
				`extern int DI;`
				`#endif`

Data—Declaration Section

Name	Sample Definition	Description	Where Used	Code Generated
Declaration Section [.c] Header	#if defined __HC12__ || defined __HC08__ #pragma DATA_SEG DEFAULT #pragma CONST_SEG DEFAULT #pragma STRING_SEG DEFAULT #elif defined(COSMIC12) #pragma section [] #endif	This line will be added at the beginning of 'glob_dat.c'	This line will be added at the beginning of 'glob_dat.c'	"#if defined __HC12__ || defined __HC08__ #pragma DATA_SEG DEFAULT #pragma CONST_SEG DEFAULT #pragma STRING_SEG DEFAULT #elif defined(COSMIC12) #pragma section [] #endif
Declaration Section [.c] Footer	#if defined __HC12__ || defined __HC08__ #pragma DATA_SEG DEFAULT #elif defined(COSMIC12) #pragma section [] #endif	This line will be added at the end of 'glob_dat.c'	This line will be added at the end of 'glob_dat.c'	"#if defined __HC12__ || defined __HC08__ #pragma DATA_SEG DEFAULT #elif defined(COSMIC12) #pragma section [] #endif
Extern Declaration Section [.h] Header		This line will be added at the beginning of 'glob_dat.h'	This line will be added at the beginning of 'glob_dat.h'	/* This is the header for extern declaration file */
Extern Declaration Section [.h] Footer		This line will be added at the end of 'glob_dat.h'	This line will be added at the end of 'glob_dat.h'	/* This is the footer for extern declaration file */

Name	Sample Definition	Description	Where Used	Code Generated

8-bit Declaration Begin Section	#pragma PRGM_8_BIT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for 8-bit data declaration begin section */' defined in 'type_declare_order.txt'	These APIs enable insertion of user code into the data declaration and extern declaration section in the files *glob_dat.c* and *glob_dat.h*.
			The order in which the data is generated can be defined by adding a file named *type_declare_order.txt* into the OSI.
			This file is a list of names of types, which defines the order in which they will be generated.
			Data of types not included in this file will be generated last.
			The file *type_declare_order.txt* may also include a set of keywords that identifies the location in which the definition of the corresponding APIs are inserted.
			There are two sets of APIs:
			1. APIs referring to the declaration file (*glob_dat.c*)
			2. APIs referring to the extern declaration file (*glob_dat.h*)
				For file *type_declare_order.txt* with the following content:
				/* Key word: for 8-bit data declaration begin section */
				/* Key word: for 8-bit data extern declaration begin section */
				unsigned char
				char
				uint8
				/* Key word: for 8-bit data declaration end section */
				/* Key word: for 8-bit data extern declaration end section */
				/* Key word: for 16-bit data declaration begin section */
				/* Key word: for 16-bit data extern declaration begin section */
				unsigned short int uint16
				/* Key word: for 16-bit data declaration end section */
				/* Key word: for 16-bit data extern declaration end section */
				Generated glob_dat.c:
				`#pragma PRGM_8_BIT_BEGIN_SEC`
				`unsigned char`
				`char`
				`uint8`
				`#pragma PRGM_8_BIT_END_SEC`
				`#pragma PRGM_16_BIT_BEGIN_SEC`
				`unsigned short int`
				`uint16`
				`#pragma PRGM_16_BIT_END_SEC`
				Generated glob_dat.h:
				#pragma PRGM_8_BIT_EXT_BEGIN_SEC
				unsigned char
				char
				uint8
				#pragma PRGM_8_BIT_EXT_END_SEC
				#pragma PRGM_16_BIT_EXT_BEGIN_SEC
				unsigned short int
				uint16
				#pragma PRGM_16_BIT_EXT_END_SEC

Name	Sample Definition	Description	Where Used	Code Generated
8-bit Declaration End Section	#pragma PRGM_8_BIT_END_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for 8-bit data declaration end section */' defined in 'type_declare_order.txt'		
16-bit Declaration Begin Section	#pragma PRGM_16_BIT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for 16-bit data declaration begin section */' defined in 'type_declare_order.txt'		
16-bit Declaration end Section	#pragma PRGM_16_BIT_END_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for 16-bit data declaration end section */' defined in 'type_declare_order.txt'		

Name	Sample Definition	Description	Where Used	Code Generated
32-bit Declaration Begin Section	#pragma PRGM_32_BIT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for 32-bit data declaration begin section */' defined in 'type_declare_order.txt'		
32-bit Declaration End Section	#pragma PRGM_32_BIT_END_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for 32-bit data declaration end section */' defined in 'type_declare_order.txt'		
Record Declaration Begin Section	#pragma PRGM_REC_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for record data declaration begin section */' defined in 'type_declare_order.txt'		

Name	Sample Definition	Description	Where Used	Code Generated
Record Declaration End Section	#pragma PRGM_REC_END_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for record data declaration end section */' defined in 'type_declare_order.txt'		
Other Types Declaration Begin Section	#pragma PRGM_OTHER_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for other types declaration begin section */' defined in 'type_declare_order.txt'		
Other Types Declaration End Section	#pragma PRGM_OTHER_END_SEC	Defines the code that will be generated to 'glob_dat.c', relating to the key-word '/* Key word: for other types declaration end section */' defined in 'type_declare_order.txt'		

Name	Sample Definition	Description	Where Used	Code Generated
8-bit Extern Declaration Begin Section	#pragma PRGM_8_BIT_EXT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for 8-bit data extern declaration begin section */' defined in 'type_declare_order.txt'		
8-bit Extern Declaration End Section	#pragma PRGM_8_BIT_EXT_END_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for 8-bit data extern declaration end section */' defined in 'type_declare_order.txt'		
16-bit Extern Declaration Begin Section	#pragma PRGM_16_BIT_EXT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for 16-bit data extern declaration begin section */' defined in 'type_declare_order.txt'		

Name	Sample Definition	Description	Where Used	Code Generated
16-bit Extern Declaration End Section	#pragma PRGM_16_BIT_EXT_END_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for 16-bit data extern declaration end section */' defined in 'type_declare_order.txt'		
32-bit Extern Declaration Begin Section	#pragma PRGM_32_BIT_EXT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for 32-bit data extern declaration begin section */' defined in 'type_declare_order.txt'		
32-bit Extern Declaration End Section	#pragma PRGM_32_BIT_EXT_END_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for 32-bit data extern declaration end section */' defined in 'type_declare_order.txt'		

Name	Sample Definition	Description	Where Used	Code Generated
Record Extern Declaration Begin Section	#pragma PRGM_REC_EXT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for record data extern declaration begin section */' defined in 'type_declare_order.txt'		
Record Extern Declaration End Section	#pragma PRGM_REC_EXT_END_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for record data extern declaration end section */' defined in 'type_declare_order.txt'		
Other Types Extern Declaration Begin Section	#pragma PRGM_OTHER_EXT_BEGIN_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for other types extern declaration begin section */' defined in 'type_declare_order.txt'		

Name	Sample Definition	Description	Where Used	Code Generated
Other Types Extern Declaration End Section	#pragma PRGM_OTHER_EXT_END_SEC	Defines the code that will be generated to 'glob_dat.h', relating to the key-word '/* Key word: for other types extern declaration end section */' defined in 'type_declare_order.txt'		

Code—Task/ISR and Related Activities

Name	Sample Definition	Description	Where Used	Code Generated

Task/ISR Opening(nameid)	?<begin>$<CK_memSegment>?<!=>?<?>#if defined __HC12__ || defined __HC08__
	#pragma CODE_SEG $<CK_memSegment>
	#elif defined(COSMIC12)
	#pragma section ($<CK_memSegment>)
	#endif?<:>?<end>
		Controls the specific Task/ISR entry function body definition.
		It will be added just before the function body begins, in <module>.c file. It will be added, as well, to all related activities functions.
			These API's enable insertion of user code before and after the Task's/ ISR's body code function, and before and after every function which is related to the Task/ISR.
			Such functions may be the functions for the Activities in the Task's/ISR's hierarchy.
			The code related to these API's is generated in <module>.c
				For an Activity named T1, defined to be an TASK, with Activity named ACT1 inside:
				Design-Attribute: CK_memSegment = CODE_SEG_1
				`#pragma CODE_SEG CODE_SEG_1`
				`void`
				`cgActivity_ACT1(void)`
				`{`
				`...`
				`}`
				`#pragma CODE_SEG DEFAULT`
				
				`#pragma CODE_SEG CODE_SEG_1`
				`void T1(void)`
				`{`
				`...`
				`cgActivity_ACT1();`
				`...`
				`}`
				`#pragma CODE_SEG DEFAULT`

Task/ISR Closure(nameid)	%=% ?<begin>$<CK_memSegment>?<!=>?<?>#if defined __HC12__ || defined __HC08__ #pragma CODE_SEG DEFAULT #elif defined(COSMIC12) #pragma section () #endif?<:>?<end>	Controls the specific Task/ISR entry function body definition. It will be added just after the function body end, in <module>.c file. It will be added, as well, to all related activities functions.		

Name	Sample Definition	Description	Where Used	Code Generated

Task/ISR Beginning Code(nameid, profileName)	/* Task/ISR Beginning Code */	Controls the code that can be added at the beginning of a Task/ISR body. It will be added at the beginning of the Task/ISR body, in the file <module>.c	These API's enable insertion of user code in specific locations of the Task's/ISR's body code function.
			The code related to these API's is generated in <module>.c
				For an Activity named T1, defined to be a TASK, with Activity named ACT1 inside:
				void T1(void)
				{
				/* Task beginning */
				<Enter Task API definition>
				/* Task beginning 2 */
				...
				cgActivity_AC1();
				...
				/* Task Ending */
				<Terminate Task API definition>
				}

Task/ISR Beginning Code Entry 2(nameid, profileName)	/* Task/ISR Beginning Code Entry 2*/	Controls the code that can be added at the beginning of a Task/ISR body. It will be added at the beginning of the Task/ISR body, after the 'Task/ISR Beginning Code' API definition, in the file <module>.c		
Task/ISR Ending Code(nameid, profileName)	/* Task/ISR Ending Code */	Controls the code that can be added at the end of a Task/ISR body. It will be added just at the end of the Task/ISR body, in the file <module>.c		
Related Function Declaration Style(nameid, returntype, arglist)	$<returntype> $<nameid>($<arglist>)	Controls the declaration style of functions which are related to the Task/ISR and related activities, such as "cgEnterActions_..." and "cgExitActions_...".	Used in the file <module>.c with the functions related to Tasks/ISRs	For an Activity named T1, defined to be a TASK, with a Control Activity named ACT_CTRL: void cgDo_ACT_CTRL(void) { …

Name	Sample Definition	Description	Where Used	Code Generated

Related Function Call(nameid, arglist)	$<nameid>($<arglist>)	Controls the call style of functions which are related to the Task/ISR and related activities, such as "cgEnterActions_..." and "cgExitActions_...".	Used in the file <module>.c with the functions related to Tasks/ ISRs	For an Activity named T1, defined to be a TASK, with a Control Activity named ACT_CTRL:
				`void T1(void)`
				`{`
				`...`
				
				`cgDo_ACT_CTRL();`

Forward Related Function Declaration(nameid, returntype, arglist)	#if defined __HC12__ || defined __HC08__
	extern ?<begin> $<CK_memSegment> ?<!=> ?<?>far ?<:> ?<end>$<returntype> $<nameid>($<arglist>);
	#else
	extern $<returntype> $<nameid>($<arglist>);
	#endif
		Controls the forward (extern) declaration of functions related to the Task/ISR and related activities, such as "cgEnterActions_..." and "cgExitActions_..."; by default, in the file 'type_def.h'	Used in the file *type_def.h* with the functions related to Tasks/ISRs	For an Activity named T1, defined to be a TASK, with a Control Activity named
				ACT_CTRL with entering reactions in it.
				`#if defined __HC12__ || defined __HC08__`
				`extern void cgEnterActions_GENERAL_CONTROL(...);`
				`#else`
				`extern void cgEnterActions_GENERAL_CONTROL(...);`
				`#endif`

Activity Function Opening(nameid)	?<begin> $<CK_compilationFlag> ?<!=> (none) ?<&&> $<CK_compilationFlag> ?<!=> ?<?>#ifdef $<CK_compilationFlag> ?<:>?<end>	Controls the specific function body definition. It will be added just before the function body begins, in the file <module>.c	Used in the file <module>.c with the functions related to Tasks/ ISRs	For Activity named AC1:
				Design-Attribute: CK_compilationFlag = CODE_SEG_1
				`#ifdef CODE_SEG_1`
				`void`
				`cgActivity_AC1(void)`
				`{`
				…

Name	Sample Definition	Description	Where Used	Code Generated

Activity Function Closure(nameid)	"?<begin> $<CK_compilation Flag> ?<!=> (none) ?<&&> $<CK_compilation Flag> ?<!=> ?<?>#endif
	?<:>?<end>
		Controls the specific function body definition. It will be added just after the function body end, in the file <module>.c	Used in the file <module>.c with the functions related to Tasks/ ISRs	For Activity named AC1:
				Design-Attribute: CK_compilationFlag = CODE_SEG_1
				`void`
				`cgActivity_AC1 (void)`
				`{`
				…
				`}`
				`#endif`

Activity Function Call Opening(nameid)	?<begin> $<CK_compilation Flag> ?<!=> (none) ?<&&> $<CK_compilation Flag> ?<!=> ?<?>#ifdef $<CK_compilation Flag> ?<:>?<end>	Controls the specific call to the function implementing the activity. It will be added just before calling the function, in the file <module>.c	Used in the file <module>.c with the functions related to Tasks/ ISRs	For Activity named AC1:
				Design-Attribute: CK_compilationFlag = CODE_SEG_1
				
				`#ifdef CODE_SEG_1`
				
				`cgActivity_AC1 ();`
				`#endif`

Activity Function Call Closure(nameid)	?<begin> $<CK_compilation Flag> ?<!=> (none) ?<&&> $<CK_compilation Flag> ?<!=> ?<?>#endif ?<:>?<end>	Controls the specific call to the function implementing the activity. It will be added just after calling the function, in the file <module>.c	Used in the file <module>.c with the functions related to Tasks/ ISRs	For Activity named AC1:
				Design-Attribute: CK_compilationFlag = CODE_SEG_1
				
				`#ifdef CODE_SEG_1`
				
				`cgActivity_AC1 ();`
				`#endif`

Name	Sample Definition	Description	Where Used	Code Generated

Statechart Beginning Code(nameid)	if((currentState_$<nameid> & $<nameid>_BITS_RANGE) != 0){
	/* Error: $<nameid> State Variable out of Range. */
	};
		Controls the code at the beginning of the Statechart. It will be added as the first executable statement in the Statechart function.	These API's enable insertion of user code in specific locations of the Statechart function (cgDo_…).
			The code related to these API's is generated in <module>.c
				For a Control Activity named ACT_CTRL:
				`void`
				`cgDo_ACT_CTRLvoid)`
				`{`
				`if((currentState_ACT_CTRL & ACT_CTRL_BITS_RANGE) != 0){`
				`/* Error: ACT_CTRL State Variable out of Range. */`
				`};`
				`...`
				`/* Statechart Ending Code */`
				`}`

Statechart Ending Code(nameid)		Controls the code at the end of the Statechart. It will be added as the last executable statement in the Statechart function.		

Code—Activities Definition Section

<table>
<tr><th>Name</th><th>Sample Definition</th><th>Description</th><th>Where Used</th><th>Code Generated</th></tr>
<tr><td>Activities Body Definition File [.c] Header</td><td>#if defined __HC12__ || defined __HC08__
#pragma CODE_SEG DEFAULT
#elif defined(COSMIC12)
#pragma section ()
#endif</td><td>This line will be added at the beginning of each <module>.c file.</td><td></td><td></td></tr>
<tr><td>Activities Body Definition File [.c] Footer</td><td>#if defined __HC12__ || defined __HC08__
#pragma CODE_SEG DEFAULT
#elif defined(COSMIC12)
#pragma section ()
#endif</td><td>This line will be added at the end of each <module>.c file.</td><td></td><td></td></tr>
</table>

Code—Per-User Function

Name	Sample Definition	Description	Where Used	Code Generated

User Function Definition Style(nameid, returntype, arglist, shortdescription)	$<returntype> $<nameid>($<arglist>)	Controls the specific function declaration style.	Used in the file *glob_func.c* for user functions.	For a Subroutine named SUBR1,
				Return type: Integer, Parameter: PRM1 of type Real:
				`int SUBR1(double PRM1)`
				`{`
				`...`

Extern User Function Declaration(nameid, returntype, arglist, shortdescription)	extern ?<begin> $<CK_memSegment> ?<!=> ?<?>far ?<:> ?<end>$<returntype> $<nameid>($<arglist>);	Controls the specific function extern declaration; by default in the file *type_def.h*	Used in the file *type_def.h* for user functions.	For a Subroutine named SUBR1, Return type: Integer, Parameter: PRM1 of type Real, Design-Attribute: CK_memSegment = SEG_1: `extern far int SUBR1(double PRM1);`

User Function Opening()	?<begin>$<CK_compilationFlag>?<!=> (none) ?<&&> $<CK_compilationFlag>?<!=> ?<?>#ifdef $<CK_compilationFlag>?<:>?<end>
	?<begin>$<CK_memSegment>?<!=>?<?>#if defined __HC12__ || defined __HC08__
	#pragma CODE_SEG $<CK_memSegment>
	#elif defined(COSMIC12)
	#pragma section ($<CK_memSegment>)
	#endif?<:>?<end>
		Controls the specific function body definition. It will be added just before the function body begins, by default in the file *glob_func.c*	Used in the file *glob_func.c* for user functions.	For a Subroutine named SUBR1,
				Return type: Integer, Parameter: PRM1 of type Real,
				Design-Attribute: CK_compilationFlag = FEATURE_1
				`#ifdef FEATURE_1`
				`int SUBR1(double PRM1)`
				`...`

<table>
<tr><th>Name</th><th>Sample Definition</th><th>Description</th><th>Where Used</th><th>Code Generated</th></tr>
<tr>
<td>User Function Closure()</td>
<td>?<begin>$<CK_memSegment>?<!=>?<?>#if defined __HC12__ || defined __HC08__
#pragma CODE_SEG DEFAULT
#elif defined(COSMIC12)
#pragma section ($<CK_memSegment>)
#endif?<:>?<end>
?<begin>$<CK_compilationFlag>?<!=> (none) ?<&&> $<CK_compilationFlag>?<!=> ?<?>#endif?<:>?<end></td>
<td>Controls the specific function body definition. It will be added just after the function body end, by default in the file glob_func.c</td>
<td>Used in the file glob_func.c for user functions.</td>
<td>For a Subroutine named SUBR1,
Return type: Integer, Parameter: PRM1 of type Real,
Design-Attribute: CK_compilationFlag = FEATURE_1

<code>int SUBR1(double PRM1)</code>
<code>{</code>
<code>…</code>
<code>}</code>
<code>#endif</code></td>
</tr>
<tr>
<td>User Function Call Style(nameid, arglist, funcPrefixWrapCode, funcPostfixWrapCode, shortdescription)</td>
<td>$<funcPrefixWrapCode>$<nameid>($<arglist>)$<funcPostfixWrapCode></td>
<td>Controls the specific function call style.
Formal Parameters:
- nameid - The name of the function.
- arglist - The code for the arguments, in the function's call.
- funcPrefixWrapCode - Function's prefix code (if any).
- funcPostfixWrapCode - Function's postfix code (if any).
- shortdescription - the function's short description</td>
<td>Used in the files where a User-Function is called: <module>.c, glob_func.c etc.</td>
<td>For a Subroutine named SUBR1,
Actual Parameter: DI_REAL,
Design-Attribute: funcPrefixWrapCode = <empty>
Design-Attribute: funcPostfixWrapCode = <empty>

<code>SUBR1(DI_REAL)</code></td>
</tr>
</table>

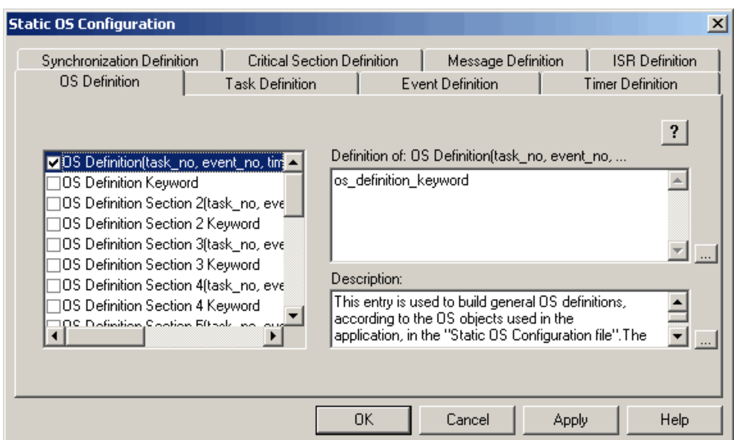
Code—User Functions Definition Section

Name	Sample Definition	Description	Where Used	Code Generated
Functions Body Definition File [.c] Header	#if defined __HC12__ || defined __HC08__ #pragma CODE_SEG DEFAULT #elif defined(COSMIC12) #pragma section () #endif	This line will be added at the beginning of the file *glob_func.c*	This line will be added at the beginning of the file *glob_func.c*	#if defined __HC12__ || defined __HC08__ #pragma CODE_SEG DEFAULT #elif defined(COSMIC12) #pragma section () ‘#endif
Functions Body Definition File [.c] Footer	#if defined __HC12__ || defined __HC08__ #pragma CODE_SEG DEFAULT #elif defined(COSMIC12) #pragma section () #endif	This line will be added at the end of the file *glob_func.c*	This line will be added at the end of the file *glob_func.c*	#if defined __HC12__ || defined __HC08__ #pragma CODE_SEG DEFAULT #elif defined(COSMIC12) #pragma section () #endif
Extern Declaration File [.h] Header	/* Ext Decl. Header */	This line will be added at the beginning of the file *type_def.h*	This line will be added in the file *type_def.h*	/* Ext Decl. Header */
Extern Declaration File [.h] Footer	/* Ext Decl. Footer */	This line will be added at the end of the file *type_def.h*	This line will be added at the end of the file *type_def.h*	/* Ext Decl. Footer */

Customizing the Static OS Configuration

The OSDT allows you to modify a large number of static OS configuration settings. These settings are accessed by clicking **Static OS Configuration...** in the main OSDT window. (This option will be grayed out unless the **Static OS Configuration** check box is selected.) These settings are categorized as follows:

- Task Definition
- Event Definition
- Timer Definition
- Synchronization Definition
- Critical Section Definition
- Message Definition
- ISR Definition



Where Definition is Used, Code Generated

For OSIs that use a static OS configuration file, you provide the names of the following two files, when defining the profile to use for code generation:

- *OS CFG Input*
    - The template file to use for the creation of the OS configuration file.
    - The keywords used in the template file will be replaced with concrete data from the model to create the OS configuration file that reflects the OS objects in the model.
- *OS CFG Output*

  The name to use for the generated static OS configuration file.

The task, event, timer, synchronization, critical section, message, and ISR definitions listed in the tables in this section are used for building the file used for generation of the static OS configuration file.

The following code will be generated for an activity named T1, defined to be a task:, and the input file given below.

Input File:

```
…
task_definition_keyword_1
…
task_definition_keyword_6
semaphore_definition_keyword_9
…
event_definition_keyword_4
...
```

Generated Code:

```
…
task_definition_1
…
task_definition_6
semaphore_definition_9
…
event_definition_4
...
```

Task Definition

Name	Sample Definition	Description
Task Definition(nameid, eventlist, semaphoreList, messageList)	task_definition_1	Used to build TASK definitions, per each TASK in the model, in the "Static OS Configuration file". If only a single API is used, it must be this one.
Task Definition Keyword	task_definition_keyword_1	Defines the appropriate location, in the "Static OS Configuration file", for TASK definitions.
Task Definition Section 2(nameid, eventlist, semaphoreList, messageList)	task_definition_2	Builds Section 2 TASK definitions, per each TASK in the model, in the "Static OS Configuration file".
Task Definition Section 2 Keyword	task_definition_keyword_2	Defines the appropriate location, in the "Static OS Configuration file", for Section 2 TASK definitions
Task Definition Section 3(nameid, eventlist, semaphoreList, messageList)	task_definition_3	Builds Section 3 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 3 Keyword	task_definition_keyword_3	Defines the appropriate location, in the "Static OS Configuration file", for Section 3 TASK definitions.
Task Definition Section 4(nameid, eventlist, semaphoreList, messageList)	task_definition_4	Builds Section 4 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 4 Keyword	task_definition_keyword_4	Defines the appropriate location, in the "Static OS Configuration file", for Section 4 TASK definitions.
Task Definition Section 5(nameid, eventlist, semaphoreList, messageList)	task_definition_5	Builds Section 5 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 5 Keyword	task_definition_keyword_5	Defines the appropriate location, in the "Static OS Configuration file", for Section 5 TASK definitions.
Task Definition Section 6(nameid, eventlist, semaphoreList, messageList)	task_definition_6	Used to build Section 6 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 6 Keyword	task_definition_keyword_6	Defines the appropriate location, in the "Static OS Configuration file", for Section 6 TASK definitions.

Name	Sample Definition	Description
Task Definition Section 7(nameid, eventlist, semaphoreList, messageList)	task_definition_7	Used to build Section 7 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 7 Keyword	task_definition_keyword_7	Defines the appropriate location, in the "Static OS Configuration file", for Section 7 TASK definitions.
Task Definition Section 8(nameid, eventlist, semaphoreList, messageList)	task_definition_8	Used to build Section 8 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 8 Keyword	task_definition_keyword_8	Defines the appropriate location, in the "Static OS Configuration file", for Section 8 TASK definitions.
Task Definition Section 9(nameid, eventlist, semaphoreList, messageList)	task_definition_9	Used to build Section 9 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 9 Keyword	task_definition_keyword_9	Defines the appropriate location, in the "Static OS Configuration file", for Section 9 TASK definitions.
Task Definition Section 10(nameid, eventlist, semaphoreList, messageList)	task_definition_10	Used to build Section 10 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 10 Keyword	task_definition_keyword_10	Defines the appropriate location, in the "Static OS Configuration file", for Section 10 TASK definitions.
Task Definition Section 11(nameid, eventlist, semaphoreList, messageList)	task_definition_11	Used to build Section 11 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 11 Keyword	task_definition_keyword_11	Defines the appropriate location, in the "Static OS Configuration file", for Section 11 TASK definitions.
Task Definition Section 12(nameid, eventlist, semaphoreList, messageList)	task_definition_12	Used to build Section 12 TASK definitions, per each TASK in the model, in the "Static OS Configuration file"
Task Definition Section 12 Keyword	task_definition_keyword_12	Defines the appropriate location, in the "Static OS Configuration file", for Section 12 TASK definitions.

Event Definition

Name	Sample Definition	Description
Event Definition(nameid, itstaskid)	event_definition_1	Used to build Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Keyword	event_definition_keyword_1	Used to define the appropriate location for Event definitions, in the "Static OS Configuration file"
Event Definition Section 2(nameid, itstaskid)	event_definition_2	Used to build Section 2 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 2 Keyword	event_definition_keyword_2	Used to define the appropriate location for Section 2 Event definitions, in the "Static OS Configuration file"
Event Definition Section 3(nameid, itstaskid)	event_definition_3	Used to build Section 3 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 3 Keyword	event_definition_keyword_3	Used to define the appropriate location for Section 3 Event definitions, in the "Static OS Configuration file"
Event Definition Section 4(nameid, itstaskid)	event_definition_4	Used to build Section 4 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 4 Keyword	event_definition_keyword_4	Used to define the appropriate location for Section 4 Event definitions, in the "Static OS Configuration file"
Event Definition Section 5(nameid, itstaskid)	event_definition_5	Used to build Section 5 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 5 Keyword	event_definition_keyword_5	Used to define the appropriate location for Section 5 Event definitions, in the "Static OS Configuration file"

Name	Sample Definition	Description
Event Definition Section 6(nameid, itstaskid)	event_definition_6	Used to build Section 6 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 6 Keyword	event_definition_keyword_6	Used to define the appropriate location for Section 6 Event definitions, in the "Static OS Configuration file"
Event Definition Section 7(nameid, itstaskid)	event_definition_7	Used to build Section 7 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 7 Keyword	event_definition_keyword_7	Used to define the appropriate location for Section 7 Event definitions, in the "Static OS Configuration file"
Event Definition Section 8(nameid, itstaskid)	event_definition_8	Used to build Section 8 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 8 Keyword	event_definition_keyword_8	Used to define the appropriate location for Section 8 Event definitions, in the "Static OS Configuration file"
Event Definition Section 9(nameid, itstaskid)	event_definition_9	Used to build Section 9 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 9 Keyword	event_definition_keyword_9	Used to define the appropriate location for Section 9 Event definitions, in the "Static OS Configuration file"
Event Definition Section 10(nameid, itstaskid)	event_definition_10	Used to build Section 10 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 10 Keyword	event_definition_keyword_10	Used to define the appropriate location for Section 10 Event definitions, in the "Static OS Configuration file"

Name	Sample Definition	Description
Event Definition Section 11(nameid, itstaskid)	event_definition_11	Used to build Section 11 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 11 Keyword	event_definition_keyword_11	Used to define the appropriate location for Section 11 Event definitions, in the "Static OS Configuration file"
Event Definition Section 12(nameid, itstaskid)	event_definition_12	Used to build Section 12 Event definitions, per each Event in the model, in the "Static OS Configuration file"
Event Definition Section 12 Keyword	event_definition_keyword_12	Used to define the appropriate location for Section 12 Event definitions, in the "Static OS Configuration file"

Timer Definition

Name	Sample Definition	Description
Timer Definition(nameid, itstaskid, eventnameid)	timer_definition_1	Used to build Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Keyword	timer_definition_keyword_1	Used to define the appropriate location for Timer definitions in the "Static OS Configuration file"
Timer Definition Section 2(nameid, itstaskid, eventnameid)	timer_definition_2	Used to build Section 2 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 2 Keyword	timer_definition_keyword_2	Used to define the appropriate location for Section 2 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 3(nameid, itstaskid, eventnameid)	timer_definition_3	Used to build Section 3 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 3 Keyword	timer_definition_keyword_3	Used to define the appropriate location for Section 3 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 4(nameid, itstaskid, eventnameid)	timer_definition_4	Used to build Section 4 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 4 Keyword	timer_definition_keyword_4	Used to define the appropriate location for Section 4 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 5(nameid, itstaskid, eventnameid)	timer_definition_5	Used to build Section 5 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 5 Keyword	timer_definition_keyword_5	Used to define the appropriate location for Section 5 Timer definitions in the "Static OS Configuration file"

Name	Sample Definition	Description
Timer Definition Section 6(nameid, itstaskid, eventnameid)	timer_definition_6	Used to build Section 6 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 6 Keyword	timer_definition_keyword_6	Used to define the appropriate location for Section 6 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 7(nameid, itstaskid, eventnameid)	timer_definition_7	Used to build Section 7 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 7 Keyword	timer_definition_keyword_7	Used to define the appropriate location for Section 7 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 8(nameid, itstaskid, eventnameid)	timer_definition_8	Used to build Section 8 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 8 Keyword	timer_definition_keyword_8	Used to define the appropriate location for Section 8 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 9(nameid, itstaskid, eventnameid)	timer_definition_9	Used to build Section 9 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 9 Keyword	timer_definition_keyword_9	Used to define the appropriate location for Section 9 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 10(nameid, itstaskid, eventnameid)	timer_definition_10	Used to build Section 10 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 10 Keyword	timer_definition_keyword_10	Used to define the appropriate location for Section 10 Timer definitions in the "Static OS Configuration file"

Name	Sample Definition	Description
Timer Definition Section 11(nameid, itstaskid, eventnameid)	timer_definition_11	Used to build Section 11 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 11 Keyword	timer_definition_keyword_11	Used to define the appropriate location for Section 11 Timer definitions in the "Static OS Configuration file"
Timer Definition Section 12(nameid, itstaskid, eventnameid)	timer_definition_12	Used to build Section 12 Timer definitions, per each Timer in the model, in the "Static OS Configuration file"
Timer Definition Section 12 Keyword	timer_definition_keyword_12	Used to define the appropriate location for Section 12 Timer definitions in the "Static OS Configuration file"

Synchronization Definition

Name	Sample Definition	Description
Semaphore Definition(nameid)	semaphore_definition_1	Used to build Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Keyword	semaphore_definition_keyword_1	Used to define the appropriate location for Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 2(nameid)	semaphore_definition_2	Used to build Section 2 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 2 Keyword	semaphore_definition_keyword_2	Used to define the appropriate location for Section 2 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 3(nameid)	semaphore_definition_3	Used to build Section 3 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 3 Keyword	semaphore_definition_keyword_3	Used to define the appropriate location for Section 3 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 4(nameid)	semaphore_definition_4	Used to build Section 4 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 4 Keyword	semaphore_definition_keyword_4	Used to define the appropriate location for Section 4 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 5(nameid)	semaphore_definition_5	Used to build Section 5 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 5 Keyword	semaphore_definition_keyword_5	Used to define the appropriate location for Section 5 Semaphore definitions in the "Static OS Configuration file"

Name	Sample Definition	Description
Semaphore Definition Section 6(nameid)	semaphore_definition_6	Used to build Section 6 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 6 Keyword	semaphore_definition_keyword_6	Used to define the appropriate location for Section 6 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 7(nameid)	semaphore_definition_7	Used to build Section 7 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 7 Keyword	semaphore_definition_keyword_7	Used to define the appropriate location for Section 7 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 8(nameid)	semaphore_definition_8	Used to build Section 8 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 8 Keyword	semaphore_definition_keyword_8	Used to define the appropriate location for Section 8 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 9(nameid)	semaphore_definition_9	Used to build Section 9 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 9 Keyword	semaphore_definition_keyword_9	Used to define the appropriate location for Section 9 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 10(nameid)	semaphore_definition_10	Used to build Section 10 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 10 Keyword	semaphore_definition_keyword_10	Used to define the appropriate location for Section 10 Semaphore definitions in the "Static OS Configuration file"

Name	Sample Definition	Description
Semaphore Definition Section 11(nameid)	semaphore_definition_11	Used to build Section 11 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 11 Keyword	semaphore_definition_keyword_11	Used to define the appropriate location for Section 11 Semaphore definitions in the "Static OS Configuration file"
Semaphore Definition Section 12(nameid)	semaphore_definition_12	Used to build Section 12 Semaphore definitions, per each Semaphore in the model, in the "Static OS Configuration file"
Semaphore Definition Section 12 Keyword	semaphore_definition_keyword_12	Used to define the appropriate location for Section 12 Semaphore definitions in the "Static OS Configuration file"

Critical Section Definition

Name	Sample Definition	Description
Critical Section Definition(nameid)	critical_section_definition_1	Used to build Critical Section definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Keyword	critical_section_definition_keyword_1	Used to define the appropriate location for Critical Section definitions in the "Static OS Configuration file"
Critical Section Definition Section 2(nameid)	critical_section_definition_2	Used to build Critical Section Section 2 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 2 Keyword	critical_section_definition_keyword_2	Used to define the appropriate location for Critical Section Section 2 definitions in the "Static OS Configuration file"
Critical Section Definition Section 3(nameid)	critical_section_definition_3	Used to build Critical Section Section 3 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 3 Keyword	critical_section_definition_keyword_3	Used to define the appropriate location for Critical Section Section 3 definitions in the "Static OS Configuration file"
Critical Section Definition Section 4(nameid)	critical_section_definition_4	Used to build Critical Section Section 4 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 4 Keyword	critical_section_definition_keyword_4	Used to define the appropriate location for Critical Section Section 4 definitions in the "Static OS Configuration file"
Critical Section Definition Section 5(nameid)	critical_section_definition_5	Used to build Critical Section Section 5 definitions, per each Critical Section in the model, in the "Static OS Configuration file"

Name	Sample Definition	Description
Critical Section Definition Section 5 Keyword	critical_section_definition_keyword_5	Used to define the appropriate location for Critical Section Section 5 definitions in the "Static OS Configuration file"
Critical Section Definition Section 6(nameid)	critical_section_definition_6	Used to build Critical Section Section 6 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 6 Keyword	critical_section_definition_keyword_6	Used to define the appropriate location for Critical Section Section 6 definitions in the "Static OS Configuration file"
Critical Section Definition Section 7(nameid)	critical_section_definition_7	Used to build Critical Section Section 7 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 7 Keyword	critical_section_definition_keyword_7	Used to define the appropriate location for Critical Section Section 7 definitions in the "Static OS Configuration file"
Critical Section Definition Section 8(nameid)	critical_section_definition_8	Used to build Critical Section Section 8 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 8 Keyword	critical_section_definition_keyword_8	Used to define the appropriate location for Critical Section Section 8 definitions in the "Static OS Configuration file"
Critical Section Definition Section 9(nameid)	critical_section_definition_9	Used to build Critical Section Section 9 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 9 Keyword	critical_section_definition_keyword_9	Used to define the appropriate location for Critical Section Section 9 definitions in the "Static OS Configuration file"

Name	Sample Definition	Description
Critical Section Definition Section 10(nameid)	critical_section_definition_10	Used to build Critical Section Section 10 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 10 Keyword	critical_section_definition_keyword_10	Used to define the appropriate location for Critical Section Section 10 definitions in the "Static OS Configuration file"
Critical Section Definition Section 11(nameid)	critical_section_definition_11	Used to build Critical Section Section 11 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 11 Keyword	critical_section_definition_keyword_11	Used to define the appropriate location for Critical Section Section 11 definitions in the "Static OS Configuration file"
Critical Section Definition Section 12(nameid)	critical_section_definition_12	Used to build Critical Section Section 12 definitions, per each Critical Section in the model, in the "Static OS Configuration file"
Critical Section Definition Section 12 Keyword	critical_section_definition_keyword_12	Used to define the appropriate location for Critical Section Section 12 definitions in the "Static OS Configuration file"

Message Definition

Name	Sample Definition	Description
Message Definition(nameid, itstaskid, datatype)	message_section_definition_1	Used to build Message definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Keyword	message_definition_keyword_1	Used to define the appropriate location for Message definitions in the "Static OS Configuration file"
Message Definition Section 2(nameid)	message_section_definition_2	Used to build Message Section 2 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 2 Keyword	message_definition_keyword_2	Used to define the appropriate location for Message Section 2 definitions in the "Static OS Configuration file"
Message Definition Section 3(nameid)	message_section_definition_3	Used to build Message Section 3 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 3 Keyword	message_definition_keyword_3	Used to define the appropriate location for Message Section 3 definitions in the "Static OS Configuration file"
Message Definition Section 4(nameid)	message_section_definition_4	Used to build Message Section 4 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 4 Keyword	message_definition_keyword_4	Used to define the appropriate location for Message Section 4 definitions in the "Static OS Configuration file"
Message Definition Section 5(nameid)	message_section_definition_5	Used to build Message Section 5 definitions, per each Message in the model, in the "Static OS Configuration file"

Name	Sample Definition	Description
Message Definition Section 5 Keyword	message_definition_keyword_5	Used to define the appropriate location for Message Section 5 definitions in the "Static OS Configuration file"
Message Definition Section 6(nameid)	message_section_definition_6	Used to build Message Section 6 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 6 Keyword	message_definition_keyword_6	Used to define the appropriate location for Message Section 6 definitions in the "Static OS Configuration file"
Message Definition Section 7(nameid)	message_section_definition_7	Used to build Message Section 7 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 7 Keyword	message_definition_keyword_7	Used to define the appropriate location for Message Section 7 definitions in the "Static OS Configuration file"
Message Definition Section 8(nameid)	message_section_definition_8	Used to build Message Section 8 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 8 Keyword	message_definition_keyword_8	Used to define the appropriate location for Message Section 8 definitions in the "Static OS Configuration file"
Message Definition Section 9(nameid)	message_section_definition_9	Used to build Message Section 9 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 9 Keyword	message_definition_keyword_9	Used to define the appropriate location for Message Section 9 definitions in the "Static OS Configuration file"

Name	Sample Definition	Description
Message Definition Section 10(nameid)	message_section_definition_10	Used to build Message Section 10 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 10 Keyword	message_definition_keyword_10	Used to define the appropriate location for Message Section 10 definitions in the "Static OS Configuration file"
Message Definition Section 11(nameid)	message_section_definition_11	Used to build Message Section 11 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 11 Keyword	message_definition_keyword_11	Used to define the appropriate location for Message Section 11 definitions in the "Static OS Configuration file"
Message Definition Section 12(nameid)	message_section_definition_12	Used to build Message Section 12 definitions, per each Message in the model, in the "Static OS Configuration file"
Message Definition Section 12 Keyword	message_definition_keyword_12	Used to define the appropriate location for Message Section 12 definitions in the "Static OS Configuration file"

ISR Definition

Name	Sample Definition	Description
ISR Definition(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_1	Used to build ISR definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Keyword	isr_definition_keyword_1	Used to define the appropriate location for ISR definitions in the "Static OS Configuration file"
ISR Definition Section 2(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_2	Used to build ISR Section 2 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 2 Keyword	isr_definition_keyword_2	Used to define the appropriate location for ISR Section 2 definitions in the "Static OS Configuration file"
ISR Definition Section 3(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_3	Used to build ISR Section 3 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 3 Keyword	isr_definition_keyword_3	Used to define the appropriate location for ISR Section 3 definitions in the "Static OS Configuration file"
ISR Definition Section 4(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_4	Used to build ISR Section 4 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 4 Keyword	isr_definition_keyword_4	Used to define the appropriate location for ISR Section 4 definitions in the "Static OS Configuration file"
ISR Definition Section 5(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_5	Used to build ISR Section 5 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 5 Keyword	isr_definition_keyword_5	Used to define the appropriate location for ISR Section 5 definitions in the "Static OS Configuration file"
ISR Definition Section 6(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_6	Used to build ISR Section 6 definitions, per each ISR in the model, in the "Static OS Configuration file"

Name	Sample Definition	Description
ISR Definition Section 6 Keyword	isr_definition_keyword_6	Used to define the appropriate location for ISR Section 6 definitions in the "Static OS Configuration file"
ISR Definition Section 7(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_7	Used to build ISR Section 7 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 7 Keyword	isr_definition_keyword_7	Used to define the appropriate location for ISR Section 7 definitions in the "Static OS Configuration file"
ISR Definition Section 8(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_8	Used to build ISR Section 8 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 8 Keyword	isr_definition_keyword_8	Used to define the appropriate location for ISR Section 8 definitions in the "Static OS Configuration file"
ISR Definition Section 9(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_9	Used to build ISR Section 9 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 9 Keyword	isr_definition_keyword_9	Used to define the appropriate location for ISR Section 9 definitions in the "Static OS Configuration file"
ISR Definition Section 10(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_10	Used to build ISR Section 10 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 10 Keyword	isr_definition_keyword_10	Used to define the appropriate location for ISR Section 10 definitions in the "Static OS Configuration file"
ISR Definition Section 11(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_11	Used to build ISR Section 11 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 11 Keyword	isr_definition_keyword_11	Used to define the appropriate location for ISR Section 11 definitions in the "Static OS Configuration file"

Name	Sample Definition	Description
ISR Definition Section 12(nameid, eventlist, semaphoreList, messageList)	isr_section_definition_12	Used to build ISR Section 12 definitions, per each ISR in the model, in the "Static OS Configuration file"
ISR Definition Section 12 Keyword	isr_definition_keyword_12	Used to define the appropriate location for ISR Section 12 definitions in the "Static OS Configuration file"

OS Definition

API Name	Example Definition	Description
OS Definition(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_1	This entry is used to build general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.The various counts are based on calls to the "Definition" API`s, like "Timer Definition" and "Event Definition". Additional API`s keywords like "Task Definition Section 2" might appear ONLY after the first one in each section.
OS Definition Keyword	os_definition_keyword_1	This entry is used to define the appropriate location, in the Static OS Configuration file, for the OS definitions.
OS Definition Section 2(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_2	This entry is used to build Section 2 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 2 Keyword	os_definition_keyword_2	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 2 of general OS definitions.
OS Definition Section 3(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_3	This entry is used to build Section 3 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.

API Name	Example Definition	Description
OS Definition Section 3 Keyword	os_definition_keyword_3	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 3 of general OS definitions.
OS Definition Section 4(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_4	This entry is used to build Section 4 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 4 Keyword	os_definition_keyword_4	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 4 of general OS definitions.
OS Definition Section 5(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_5	This entry is used to build Section 5 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 5 Keyword	os_definition_keyword_5	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 5 of general OS definitions.
OS Definition Section 6(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_6	This entry is used to build Section 6 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 6 Keyword	os_definition_keyword_6	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 6 of general OS definitions.
OS Definition Section 7(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_7	This entry is used to build Section 7 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 7 Keyword	os_definition_keyword_7	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 7 of general OS definitions.

API Name	Example Definition	Description
OS Definition Section 8(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_8	This entry is used to build Section 8 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 8 Keyword	os_definition_keyword_8	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 8 of general OS definitions.
OS Definition Section 9(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_9	This entry is used to build Section 9 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 9 Keyword	os_definition_keyword_9	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 9 of general OS definitions.
OS Definition Section 10(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_10	This entry is used to build Section 10 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 10 Keyword	os_definition_keyword_10	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 10 of general OS definitions.
OS Definition Section 11(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_11	This entry is used to build Section 11 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 11 Keyword	os_definition_keyword_11	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 11 of general OS definitions.

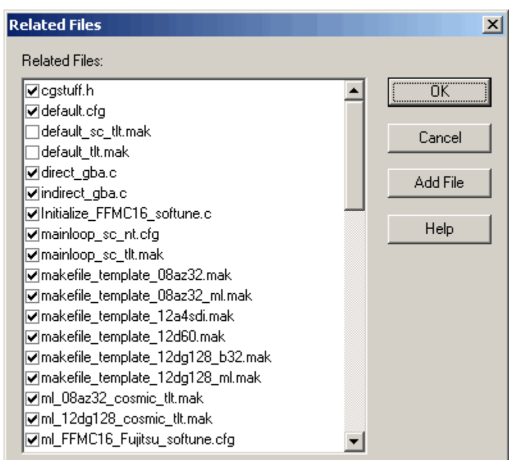
API Name	Example Definition	Description
OS Definition Section 12(task_no, event_no, timer_no, sem_no, crit_sec_no, mess_no, isr_no)	os_section_definition_12	This entry is used to build Section 12 of general OS definitions, according to the OS objects used in the application, in the Static OS Configuration file.
OS Definition Section 12 Keyword	os_definition_keyword_12	This entry is used to define the appropriate location, in the Static OS Configuration file, for Section 12 of general OS definitions.

Specifying Related Files

The main OSDT screen contains a button labeled **Related Files...**. When this button is pressed, a list of files is displayed. These files are files that you will want to have available in your workarea, for example, makefiles, .h files, and .oil files. The specific files listed will depend upon the OSI you are using.

When you create a new workarea, these files are copied to the workarea's *prt* directory.

To remove a file from the files that will be copied to this directory, clear the check box next to the file. To add a file, click the **Add File** button.



Upgrading an OSI

The OSDT is capable of upgrading OSIs that were created with previous versions of the tool.

To upgrade an existing OSI, select **File** > **Update from OSI...** from the main menu.

The upgrade operation carries out the following actions:

- Checks for API definitions that exist in the reference OSI but not in the OSI being upgraded. You will be prompted to approve these additions.
- Checks for obsolete API definitions—those that exist in the OSI being upgraded but not in the reference OSI. You will be prompted to approve the removal of these obsolete definitions.
- Checks the list of Related Files for new, modified, or obsolete files.
- Checks for design attributes that exist in the reference OSI but not in the OSI being upgraded. You will be prompted to approve these additions.
- Checks for obsolete design attributes—those that exist in the OSI being upgraded but not in the reference OSI. You will be prompted to approve the removal of these obsolete design attributes.

All changes made during the upgrade operation are recorded in a log file located in the ctd directory (file is named <OSI NAME>DD/MM/YY.txt).

When the upgrade operation is completed, the OSI will be marked as "modified." The changes will only take effect when the OSI is saved.

Index