METHOD > PARK

Poseidon for UML
Embedded Edition

ANSI C TUTORIAL

Contact:

method park Software AG

Wetterkreuz 19a

91058 Erlangen

Germany

phone:

+49-9131-97206-0

fax:

+49-9131-97206-200

e-mail:

info@methodpark.de

Contents

1 INTRODUCTION 4

2 GUI 5

2.1 C/C++ PROPERTIES PANEL 5
2.2 C/C++ CODE PREVIEW PANEL 6
2.3 GENERAL SETTINGS FOR GENERATION 6

3 EXAMPLE PROJECT 7

3.1 IMPLEMENTATION OF CLASSES 8
3.2 CLASS DISPLAY 8
3.3 CLASS HIFI 9

4 GENERATION 11

4.1 STRUCTURE OF THE GENERATED FILES 13
4.1.1 Forward Declaration File 13
4.1.2 Declaration File 14
4.1.3 Implementation 17
4.2 USING THE CLASS HIFI 18
4.3 USE OF POLYMORPHISM 19

5 GENERATING CODE FOR STATE MACHINES 22

5.1 HOW TO USE A STATE MACHINE 23
5.2 IMPLEMENTATION OF THE CLASS TUNER 23

6 EXTENDING THE EXAMPLE 28

7 OPTIMIZATIONS 30

7.1 OPTIMIZATIONS OF CLASSES 30

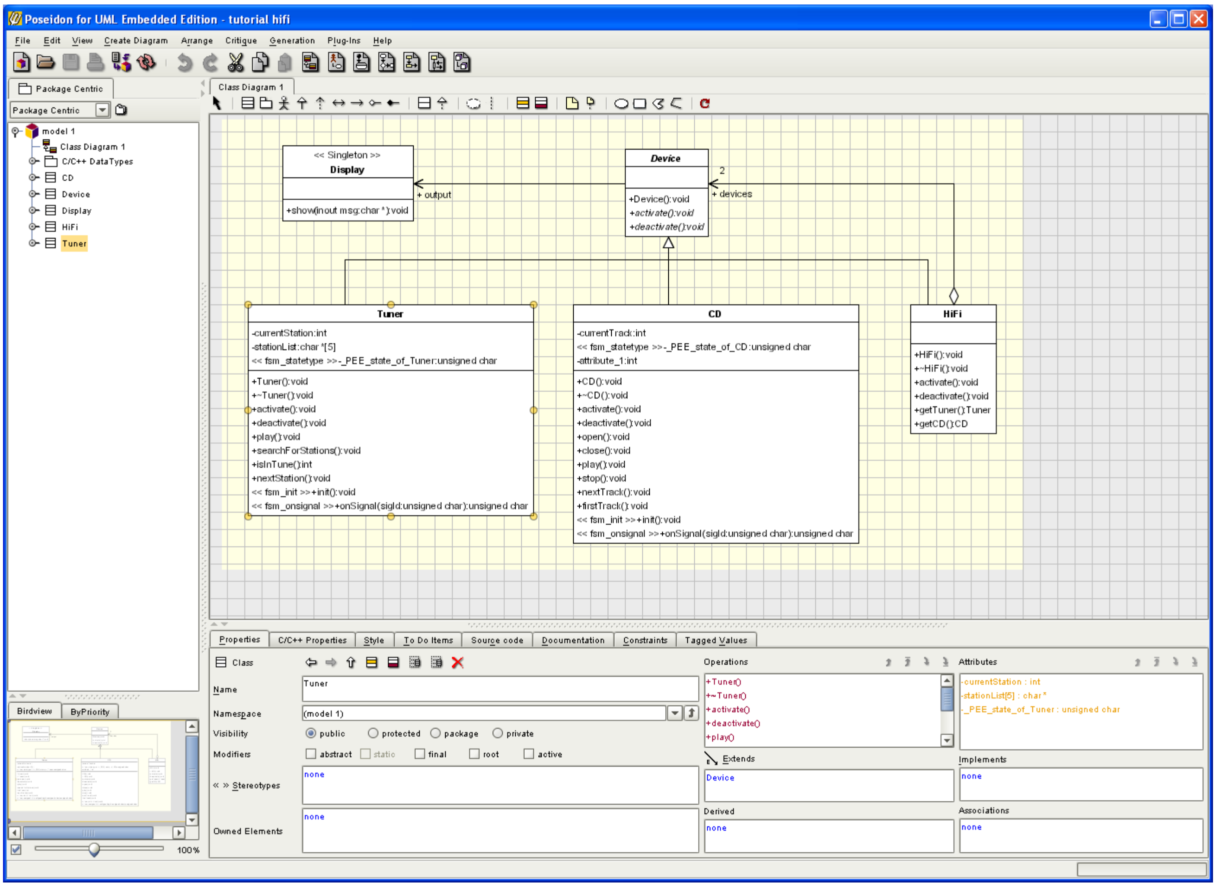
1 Introduction

This tutorial gives the user an understanding of how to work with Poseidon for UML Embedded Edition and ANSI C as target language. Examples are used to walk the user through a typical project with Poseidon for UML Embedded Edition. The tutorial serves as a guideline and shows the most important steps necessary for working with Poseidon for UML Embedded Edition.

For a detailed description of the ANSI C/C++ code generation please refer to the Embedded Edition User Guide. The installation and registration of Poseidon for UML is explained in the general Poseidon User Guide.

2 GUI

The work area of Poseidon is separated in five parts. At the top of the window, there is a main menu and a tool bar that give access to the main functionalities. Below this are four panes. The biggest pane is called the diagram pane where the various UML diagrams are displayed. To the left of it, you find the so-called navigation pane. The two areas on the bottom are the overview pane and the details pane.



The details pane is composed of a number of different panels that can be selected through corresponding tabs at the top of the pane. Please refer to the Poseidon User Guide for more information about the Poseidon GUI.

The Embedded Edition introduces two new detail panels: the C/C++ Properties panel and the C/C++ Code Preview panel. The C/C++ Properties panel allows you to modify language-dependent properties of the currently selected model element. The Code Preview panel shows the code that will be generated for an element. If an operation is currently selected, it is also possible to edit its body.

2.1 C/C++ Properties Panel

The C/C++ Properties panel allows making further target-language specific settings for the following model elements:

- Class (Settings for target language, file names, and optimizations)

- Attribute (Settings for additional modifiers)
- Operation (Settings for binding and additional modifiers)
- Parameter (Settings for passing the parameter by value, pointer, or reference)
- Package (Settings for namespace and target path for generation)
- Association end (Settings for the containment of the association end)

2.2 C/C++ Code Preview Panel

The code preview always displays the implementation of the currently selected model element in the corresponding target language. As mentioned above, you can edit the body of operations in the code preview when an operation is selected.

If you would like to look at the implementation of a class, select the class and then the Code Preview details panel. Three files are generated for each class:

- Forward declaration
- Declaration
- Implementation

By default, the preview displays the forward declaration file. In order to view the declaration or implementation file you can switch to the next file by pressing ALT + N or selecting the *NextSource* entry in the View menu.

2.3 General Settings for Generation

In order to open the dialog for the general settings, select the Generation/*Generate Code for Classes of Model* menu entry. Click on the dialog’s “Settings” button (Please make sure that the “Kind of generation” radio button is set to “Embedded”.)

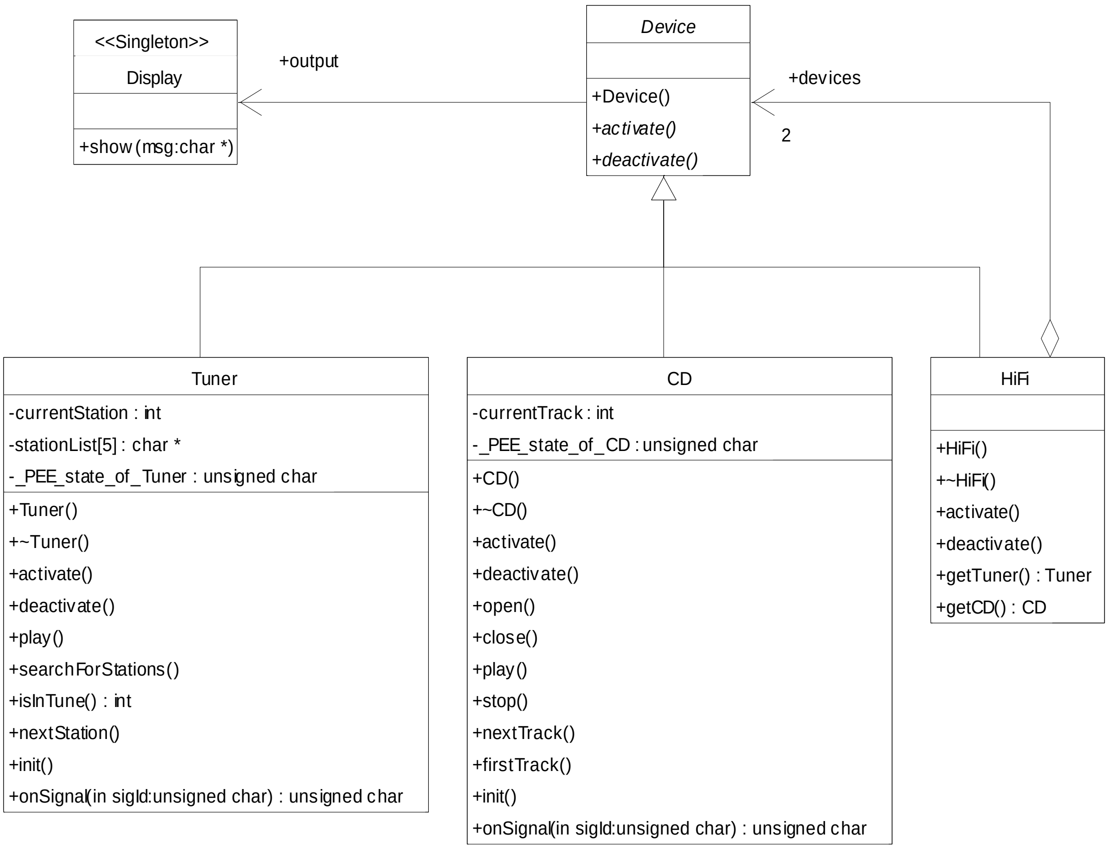
In the general settings dialog you can make settings for

- style,
- file headers,
- tracing,
- memory management, and
- synchronization.

3 Example Project

The following example model shows how to work with Poseidon for UML Embedded Edition and generate code.

This example is a simplified hi-fi system. The model of the hi-fi system is included with Poseidon for UML Embedded Edition. In order to open the "HiFi" project select *File/Open Project* in the main menu. (The project can be found in the *examples/hifi* subdirectory of your Poseidon Embedded Edition installation directory.)



The classes *Tuner*, *CD,* and *HiFi* are all inherits of class *Device* and share certain qualities, in this case the operations **`activate()`** and **`deactivate()`**. The device *HiFi* itself consists of exactly two devices (in this example the Tuner and CD player). The operations **`activate()`** and **`deactivate()`** are declared as abstract operations in class *Device*, thus each device has a common interface for activating and deactivating. The classes *Tuner*, *CD,* and *HiFi* overwrite these operations.

Each device has an association to a display. In this case all devices share the same display and only one display is necessary. Therefore, the display is realized as a singleton class using the <<Singleton>> stereotype.

3.1 Implementation of Classes

The operations of the classes have not been implemented yet. Their implementation is shown in the following sections.

3.2 Class Display

The class *Display* is realized as a <<Singleton>>, which means that only one instance exists during the whole runtime of the system. Poseidon Embedded Edition supports the singleton pattern and automatically generates the operations

- **`getInstance() : class *`**
- **`destroyInstance() : void`**

for singleton classes.

The generated code for singleton classes has been especially optimized for embedded systems.

The class *Display* has only one operation **`+show(msg:char *)`**. Select this operation and choose the Operation Preview property panel, which shows the implementation of that operation.

- ***Implement the operation as shown below***

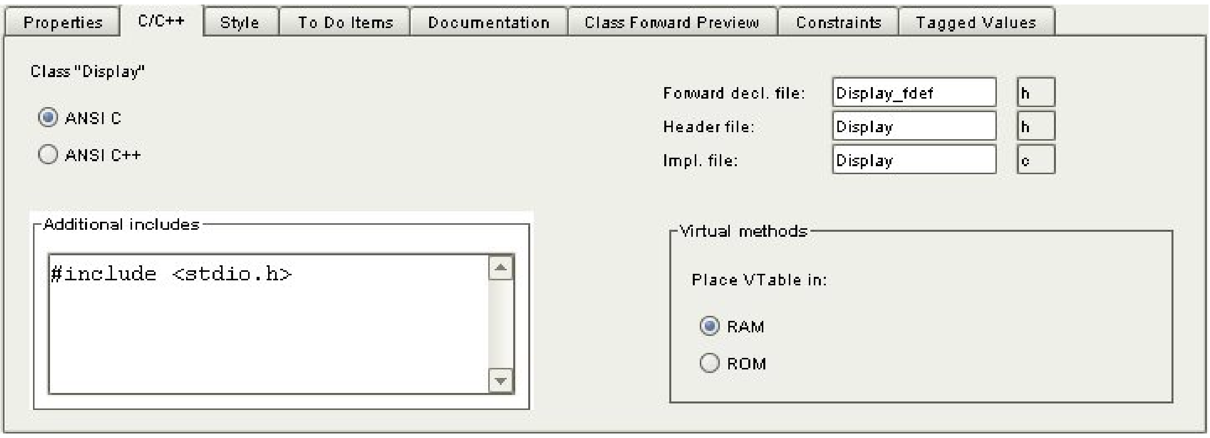
```
void Display_show(char * msg) {
   DEFINESELF  /** lock-end */

   /* -> add your code here */
   printf("%s\n", msg);

} /** lock-begin */
```

The operation **`show(msg : char *)`** uses the standard library's **`printf()`** function for I/O, hence the header file *stdio.h* has to be included. In order to include such additional headers you can use the **Additional Includes** text field in the C/C++ property panel for classes.

- ***Select the class Display and choose the C/C++ property panel.***
- ***Add* #include <stdio.h> *to the "Additional Includes" text field as shown below.***



3.3 Class HiFi

The class *HiFi* has the following operations:

- **`activate() : void`**
- **`deactivate() : void`**
- **`Tuner() : Tuner*`**
- **`CD() : CD*`**

The implementation of these operations can also be done with the operation preview.

- ***Select the operation activate().***
- ***Implement the operation as shown below.***

Calls the method show() of class Display.

```
void HiFi_activate(void * _self) {
   DEFINESELF  /** lock-end */

   /* -> add your code here */
   MSG_Display_show(self_output, "HiFi is activated.");

} /** lock-begin */
```

- ***In the same way, add the following line of code to the implementation of the operation deactivate().***

```
MSG_Display_show(self_output, “HiFi is deactivated.”);
```

The first parameter of each operation is the **self pointer** (**`void * _self`**). The **`DEFINESELF`** macro at the first line of each operation body casts the self pointer to the corresponding class type. The self pointer **`self`** can be used the same way as in other object-oriented languages.

`MSG_Display_show()` is a facade macro for calling the operation **`show()`** of the class *Display*. Such macros are automatically generated for all operations. They hide the various ways of calling the operation (either as a global function or from the virtual method table; the operation can also be an inline macro).

The first parameter **`self_output`** of the operation **`show()`** is the object of the class *Display*. Due to the association from class *Device* to class *Display*, where *Display* plays the role of *output* in the class *Device*, an attribute named *output* is added to the class *Display*. Associations that have a specified role are always generated in this way. If an association does not have a role, it is not be considered for generation.

The “attribute” output has to be initialized with the instance of the class *Display* in the constructor of the class *Device*.

- ***Select the constructor Device() of the class Device and append the initialization of the attribute output as shown below.***

```
Device_Device(void * _self) {
   DEFINESELF  /** lock-end */
   /* -> add your code here */

   /*@ <Init> @*/
   /* Setup of the virtual method table */
   if (_PE_IsFirstObject == 1 /*true*/) {
      _PE_IsFirstObject = 0 /*false*/;
      _PE_Device_MT._PE_destroy = _PE_Device_Destroy;
      _PE_Device_MT.activate = NULL;
      _PE_Device_MT.deactivate = NULL;
   }
   self->_PE_vMethods = &_PE_Device_MT;
   /*@ </Init> @*/

   /* -> add your code here */
   self_output = Display_getInstance();

} /** lock-begin */
```

Setup of the virtual method table. Do not add or change code between the tags <Init> and </Init>, since it is generated automatically.

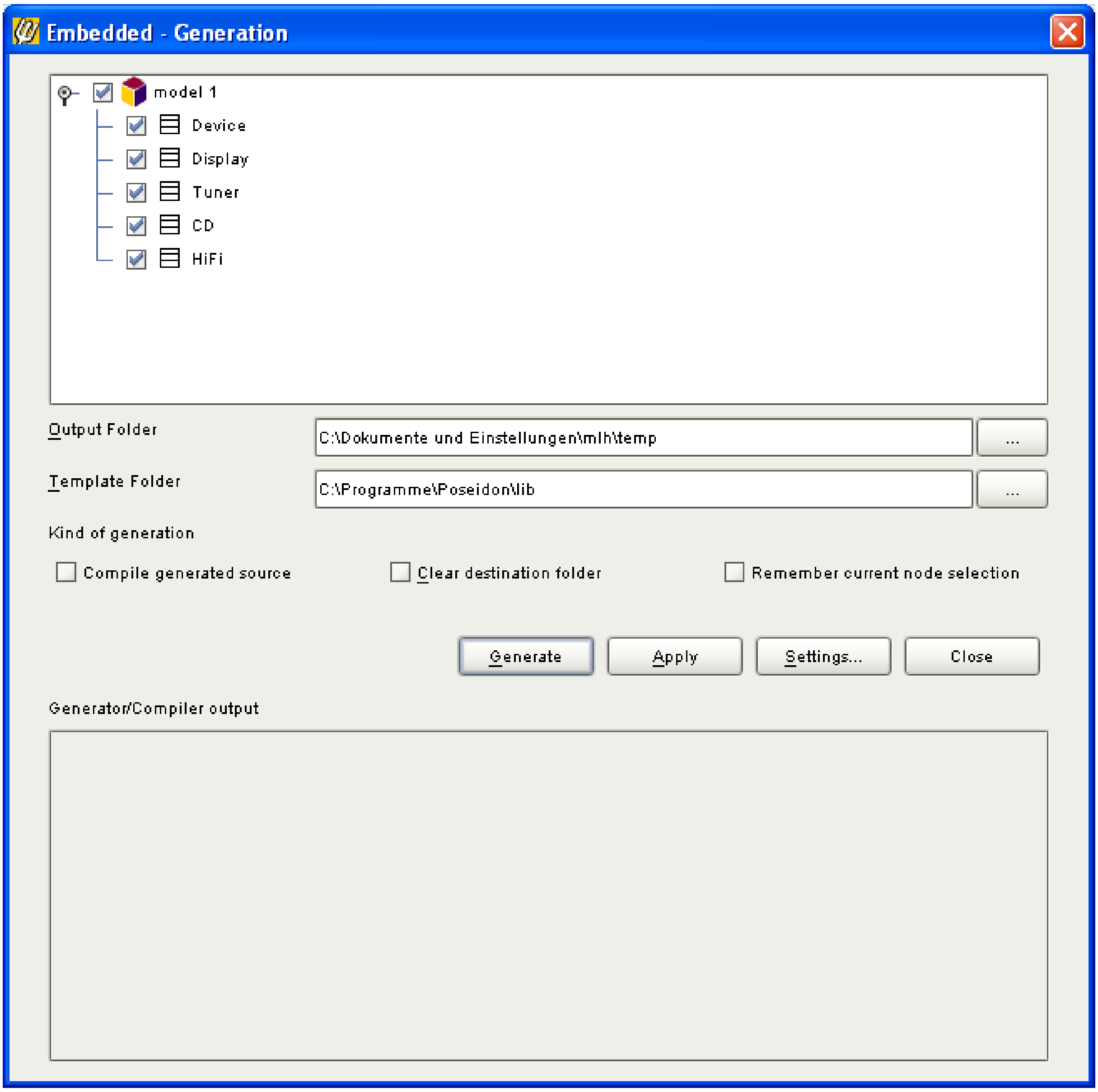
Display_getInstance() returns the instance of class Display. If no instance exists, it will be created.

Initialization of the attribute output with the instance of class Display.

4 Generation

We have implemented a part of our model and would now like to generate ANSI C source code.

- ***To generate the model, select "Generation>Embedded " in the main menu. The following dialog comes up:***



The list box at the top represents the structure of your model and lets you select the packages or classes for generation.

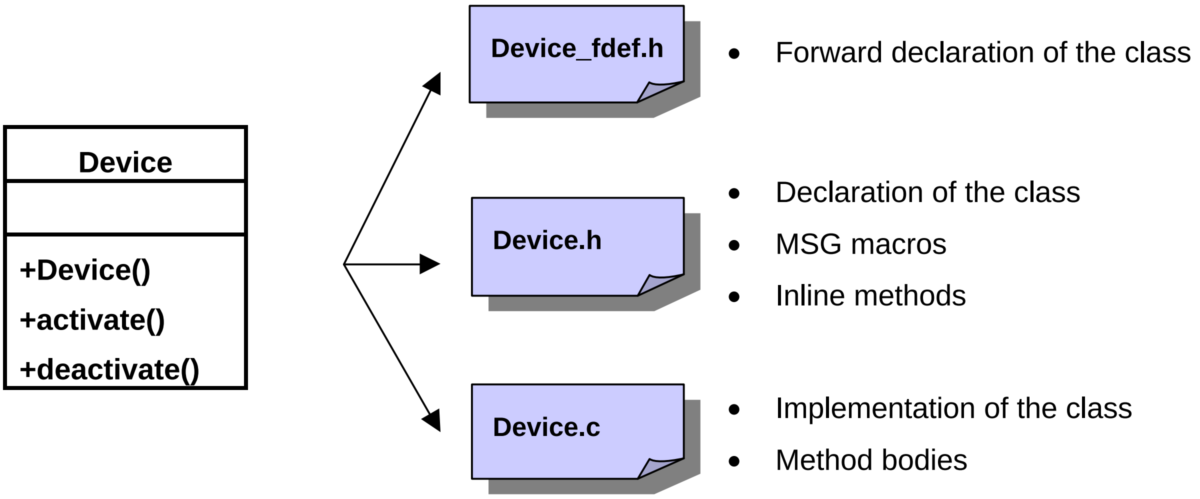
- ***Select all classes.***
- ***Set your target path in the "Output Folder" text field.***
- ***Click the "Generate" button .***

The model has been generated. The target directory now contains the following files:

CD	c
CD	h
CD_fdef	h
Device	c
Device	h
Device_fdef	h
Display	c
Display	h
Display_fdef	h
HiFi	c
HiFi	h
HiFi_fdef	h
Tuner	c
Tuner	h
Tuner_fdef	h

4.1 Structure of the Generated Files

Poseidon generates three files for each class:



4.1.1 Forward Declaration File

Open the file *HiFi_fdef.h* in a text editor or an IDE for C/C++.

You can see the following structure:

```
/*@ <FileComment ID=1048775481296> @*/
/*************************************************
* Class           : Device
* File            : Device_fdef.h
* Generated with : Poseidon for UML PE 1.6
* Last generation: Thu Mar 27 15:31:21 CET 2003
*************************************************/
/*@ </FileComment ID=1048775481296> @*/

/*@ <IncludeGuard> @*/
#ifndef Device_10_10_1__99_1be9101_f42fadf6b9__7fff_H_FDEF_H
#define Device_10_10_1__99_1be9101_f42fadf6b9__7fff_H_FDEF_H
/*@ </IncludeGuard> @*/

/*@ <Definitions> @*/
#define Device struct _PE_Device
/*@ </Definitions> @*/

/*@ <IncludeGuardEnd> @*/
#endif  /* Device_10_10_1__99_1be910                         */
/*@ </IncludeGuardEnd> @*/
```

Sections generated by Poseidon are enclosed in tags. Do not modify this code because it will be overwritten by the next generation. You can add your code outside the protected sections.

Forward declaration of the class Device

The sections generated by Poseidon Embedded Edition are enclosed in tags. The code between these tags is re-generated with each generation. You can add your own code outside the sections. It will be kept when you activate the synchronization between files and model.

4.1.2 Declaration File

The declaration file (*.h) has the following structure:

Forward declaration of operations

Declaration of the class

Facade macros for operations

4.1.2.1 Facade Macros

The macros encapsulate the calling mechanism of an operation. These macros have the following structure:

`MSG_<Class name>_<Operation name>(<self pointer>, <parameter list>)`

Please use only these macros for calling operations. The following code shows you the facade macros for the operations `activate()`, `deactivate()`, `getTuner()`, and `getCD()` of the class *HiFi*.

```
#define MSG_HiFi_activate(obj)  \
    ((obj)->_PE_vMethods->activate(obj))

#define MSG_HiFi_deactivate(obj)  \
    ((obj)->_PE_vMethods->deactivate(obj))

#define MSG_HiFi_getTuner(obj)  \
    HiFi_getTuner(obj)

#define MSG_HiFi_getCD(obj)  \
    HiFi_getCD(obj)
```

Operation call using the virtual method table.

Operation call via regular C function.

4.1.2.2 Macros for Dynamic or Static Creation or Destruction of Objects

In addition to the facade macros for operations there are macros for the dynamic and static creation or destruction of an object:

dynamic:

`NEW_<classname>(obj)`	Creates a new instance of the class and calls the constructor of the class.
`VIRTUALNEW_<classname>(type, obj)`	Works like NEW_<classname>(obj) but casts the instance to the type specified by `type`.
`DELETE_<classname>(obj)`	Destroys an instance of the class.

static:

`INIT_<classname>(obj)`	Initializes a static instance of the class.
`DEINIT_<classname>(obj)`	Deinitializes a static instance of the class.

4.1.2.3 Facade Macros for Attributes

Each class has facade macros for access to its own and inherited attributes except for class attributes. These macros have the following structure:

```
self_<attributename>
```

4.1.3 Implementation

The implementation file (*.c) contains the implementation of the operations of a class. The following is an excerpt of the file Tuner.c:

```
/*@ <Operation ID=10-10-1--99-1be9101:f42fadf6b9:-7fa1> @*/
void HiFi_activate(void * _self) {
    DEFINESELF

    /* -> add your code here */

}
/*@ </Operation ID=10-10-1--99-1be9101:f42fadf6b9:-7


/*@ <Operation ID=10-10-1--99-1be9101:f42fadf6b9:-7f9e> @*/
void HiFi_deactivate(void * _self) {
    DEFINESELF

    /* -> add your code here */

}
/*@ </Operation ID=10-10-1--99-1be9101:f42fadf6b9:-7f9e


/*@ <Operation ID=10-10-1--99-1be9101:f42fadf6b9:-7f9b> @*/
Tuner * HiFi_getTuner(void * _self) {
    DEFINESELF

    /* -> add your code here */

    return 0;
}
/*@ </Operation ID=10-10-1--99-1be9101:f42fadf6b9:-7f9b> @*/

...
```

You can add the implemen-tation of your method here.

The first parameter of each operation is always the **`self`** pointer.

Macro for the declaration of the self pointer and cast to the proper type.

4.2 Using the Class HiFi

The classes *HiFi*, *Device,* and *Display* have now been implemented to print the message “HiFi is activated“ or “HiFi is deactivated“ on activation and deactivation, respectively .

The files of the classes have been generated. Create a C/C++ project and add the generated files to this project. In order for the project to be compiled correctly, it is necessary to create a `main()` function.

➢ ***Implement the `main()`-function as follows :***

```
#include "HiFi.h"                    // Include the declaration of
                                     // class HiFi


void main(void) {


       HiFi *myHiFi;                 // Pointer of type class HiFi


       NEW_HiFi(myHiFi);             // Creates an instance of class HiFi


       MSG_HiFi_activate(myHiFi);    // Calls the operation activate()
       MSG_HiFi_deactivate(myHiFi);  // Calls the operation deactivate()


       DELETE_HiFi(myHiFi);          // Calls the destructor of HiFi
}
```

We generate the model and try it out, now.

➢ ***Compile the project and execute it.***

You get the following output:

```
HiFi is activated
HiFi is deactivated
```

4.3 Use of Polymorphism

We now extend our example such that the devices (*Tuner, CD*) are activated on calling the operation **`activate()`** of class *HiFi*.

Like the class *HiFi,* the classes *Tuner* and *CD* are also inherited from class *Device* and implement the interface **`activate()`** and **`deactivate()`**. Implement the operation **`activate()`** of class *Tuner* as described in the following:

- ***Select the operation `activate()` of class Tuner and add the following line of code in the operation preview:***

```
MSG_Display_show(self_output, "Tuner is activated\n");
```

- ***Edit the operation `deactivate()` of class Tuner analogously:***

```
MSG_Display_show(self_output, "Tuner is deactivated\n");
```

- ***Repeat this for the class CD:***

Implementation of operation `activate()`:

```
MSG_Display_show(self_output, "CD is activated\n");
```

Implementation of operation `deactivate()`:

```
MSG_Display_show(self_output, "CD is deactivated\n");
```

Now, we extend the operation **`activate()`** of class *HiFi*, so that on activating the HiFi the Tuner and CD are activated as well. This means that we create an instance of class *Tuner* and class *CD* with the operation **`activate()`** of class *HiFi* and destroy these instances on the operation **`deactivate()`** of class *HiFi*. The instances of Tuner and CD are stored in the aggregation of class HiFi to the Device.

➢ ***Select the operation `activate()` of the class HiFi and change the implementation like this:***

```
/* -> add your code here */

MSG_Display_show(self_output, "HiFi is activated\n");

VIRTUALNEW_Tuner(Device, self_devices[0]);
VIRTUALNEW_CD(Device, self_devices[1]);

MSG_Device_activate(self_devices[0]);
MSG_Device_activate(self_devices[1]);
```

Create an instance of the classes Tuner and CD and cast it to the type of class Device.

Calling the methods activate() on the devices. Due to the polymorphism the method activate() of the class Tuner and CD is called.

On calling the operation `deactivate()` of class HiFi, the devices will be destroyed.

➢ ***Change the implementation of operation `deactivate()` of class HiFi as shown below:***

Calling the methods deactivate() on the devices. Due to the polymorphism the method deactivate() of the class Tuner and CD is called.

```
/* -> add your code here */
MSG_Device_deactivate(self_devices[0]);
MSG_Device_deactivate(self_devices[1]);

DELETE_Device(self_devices[0]);
DELETE_Device(self_devices[1]);

MSG_Display_show(self_output, "HiFi is deactivated\n");
```

Calling the destructors of class Tuner and CD.

- ***Again generate the model.***

- ***Compile the project and execute it.***

You get the following output:

```
HiFi is activated
Tuner is activated
CD is activated
Tuner is deactivated
CD is deactivated
HiFi is deactivated
```

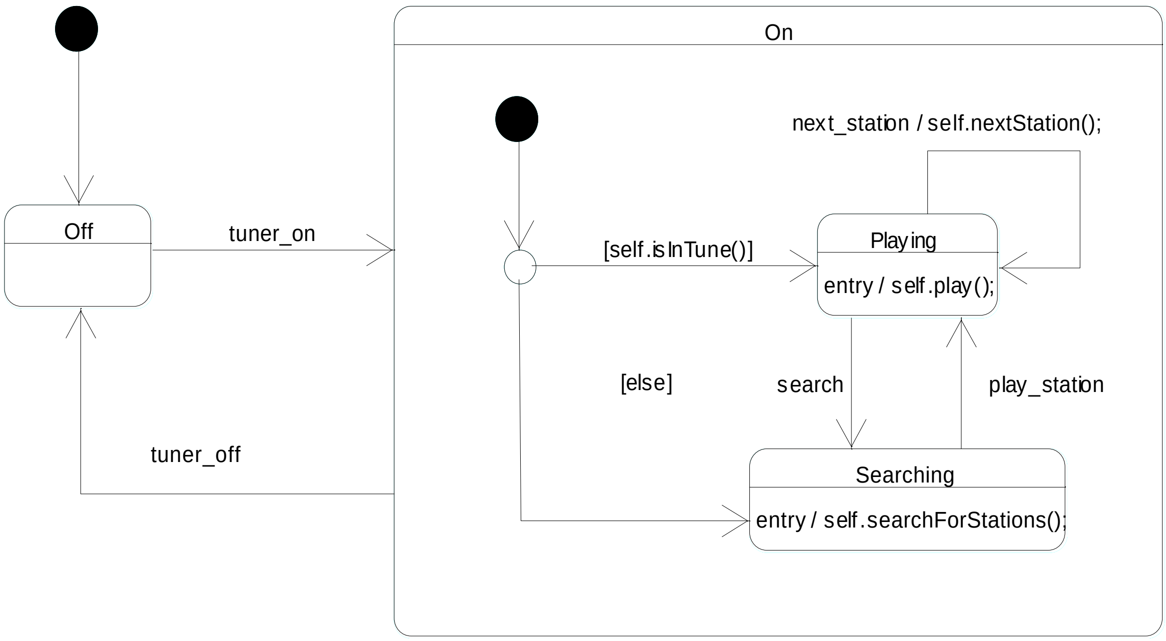
5 Generating Code for State Machines

State machines are a powerful instrument for describing the behavior of a class. Event-driven classes can be described easily and fast. Poseidon allows you to generate code from state machines. Because of the formal specification of state machines, they can be completely generated and the user does not have to further implement the generated code.

In our example two classes have state machines:

- ✓ *Tuner*
- ✓ *CD*

The following is the state machine of the class *Tuner*



The *Tuner* can be in two conditions: *Off* or *On*. If the Tuner is in state *On*, it is either in state *Playing* or in state *Searching*.

If the Tuner is playing, it can be switched to the next station by sending the event “next_station”, or to the state *Searching*. In state *Searching* the Tuner searches for stations and goes back to state *Playing* when it receives the event “play_station”.

On entering the state *On* the operation **`isInTune()`** is called to decide if the Tuner must enter the state *Playing* or the state *Searching.*

5.1 How to Use a State Machine

There are two operations for communicating with the state machine. These operations are public operations of the class that contains the state machine and are generated automatically. The following shows the facade macros of these operations for the class Tuner:

```
MSG_Tuner_init(<object>)
MSG_Tuner_onSignal(<object>, <signal>)
```

The operation `init()` initializes the state machine and the initial transition is done. The operation `onSignal()` sends a signal to the state machine.

5.2 Implementation of the Class Tuner

The class *Tuner* has two attributes: `stationList` and `currentStation`. The initialization of these attributes can be done in the constructor of the class *Tuner*:

➢ ***Change the implementation of the class Tuner to the following:***

```
Tuner_Tuner(void * _self) {
        DEFINESELF  /** lock-end */
        int i;
        /* Call super class constructor */
        _PE_Device_Create(self);
        /* -> add your code here */
        /*@ <Init> @*/
                /* Setup of the virtual method table */
                ...
        /*@ </Init> @*/
        /* -> add your code here */
        for (i = 0; i < 5; self_stationList[i++] = NULL);
        self_currentStation = 0;

        MSG_Tuner_init(self);
} /** lock-begin */
```

Declaration of a local variable for the loop.

Initialization of the attributes stationList and currentStation.

Initializatin of the state machine. The initial transition is taken.

The initialization of the attributes is necessary to make sure that they contain a valid value. The attribute **`stationList`** is the list of stations and contains pointers to the station names.

The attribute **`currentStation`** contains the position of the current station in the **`stationList`**.

In order to enter the state *On* when calling the operation **`activate()`** of the class Tuner, we must send the signal “tuner_on” to the state machine.

- ***Add the following line to the operation `activate()` of the class Tuner:***

```
MSG_Tuner_onSignal(self, tuner_on);
```

- ***Add the following line to the operation `deactivate()` of the class Tuner:***

```
MSG_Tuner_onSignal(self, tuner_off);
```

The Tuner switches from the state *Off* to the state *On* when the operation **`activate()`** is called.

On entering the state *On* it is decided, if the Tuner is to enter to the state *Playing* or the state *Searching*. This decision is done in the operation **`isInTune()`** of the class Tuner.

- ***Implement the operation `isInTune()` as described here:***

```
/* -> add your code here */
// check, if the current station is in the station list
if (self_stationList[self_currentStation] != NULL) {
      // current station is valid
      return 1;
}
// current station is not valid
return 0;
```

- ***Add the following line to the operation `searchForStations()` of class Tuner:***

```
MSG_Display_show(self_output, “Tuner is searching.”);
```

- ***Add the following line to the operation `play()` of class Tuner:***

```
MSG_Display_show(self_output, “Tuner is playing.”);
```

- ***Add the following line to the operation `nextStation()` of class Tuner:***

```
MSG_Display_show(self_output, “switch to next station.”);
```

- ***Again generate the model.***

Compile the project and execute it.

You get the following output:

```
HiFi is activated
Tuner is activated
Tuner is searching
CD is activated
Tuner is deactivated
CD is deactivated
HiFi is deactivated
```

Calling **`activate()`** on the class *Tuner* sends the signal “tuner_on” to the state machine and the Tuner switches to the state *On*. The decision whether the Tuner enters the state *Playing* or the state *Searching* on entering the state On is made in the operation **`isInTune()`**.

We now extend our example in the following way:

➢ ***Change your*** `main()` ***function to the following:***

```
#include "HiFi.h"                              // Include the declaration of
                                               // class HiFi
#include "Tuner.h"                             // Include the declaration of
                                               // classe Tuner


void main(void) {

      HiFi *myHiFi;                            // Pointer of type class HiFi
      Tuner *myTuner;                          // Pointer of type class Tuner

      NEW_HiFi(myHiFi);                        // Create an instance of class HiFi

      MSG_HiFi_activate(myHiFi);               // Call operation activate() of HiFi

      myTuner = MSG_HiFi_getTuner(myHiFi);     // get the instance of Tuner

                                               // send the signal play_station to
                                               // the state machine of class Tuner
      MSG_Tuner_onSignal(myTuner, play_station);

                                               // sende the signal next_station to
                                               // the state machine of class Tuner
      MSG_Tuner_onSignal(myTuner, next_station);

      MSG_HiFi_deactivate(myHiFi);             // Call operation deactivate()

      DELETE_HiFi(myHiFi);                     // Call destrukcors of HiFi
}
```

Please remember to include the declaration file for the class Tuner in the **`main()`** function.

The operations **`getTuner()`** and **`getCD()`** of class Tuner are not implemented yet.

- ***Change the implementation of operation*** `getTuner()` ***to the following:.***

```
return  (Tuner*)self_devices[0];
```

- ***Change the implementation of operatione*** `getCD()` ***to the following:***

```
return  (CD*)self_devices[1];
```

- ***Again generate the model.***
- ***Compile the project and execute it.***

You get the following output:

```
HiFi is activated
Tuner is activated
Tuner is searching
CD is activated
Tuner is playing
Switch to next station
Tuner is playing
Tuner is deactivated
CD is deactivated
HiFi is deactivated
```

6 Extending the Example

Our hi-fi system contains a Tuner and a CD player. The Tuner is implemented in a simple way. We would like to implement searching for station and set up a station list.

- ***Implement the operation `searchForStations()` of class Tuner as shown below:***

```
/* -> add your code here */
int i;

for (i = 0; i < 5; i++) {
	if (self_stationList[i] == NULL) {
		self_stationList[i] = (char*)malloc(10 * sizeof(char));
	}
	sprintf(self_stationList[i], "STATION %d", i);
}
```

This fills the station list with station names.

In order to change the station, the signal “next_station” is sent to the state machine of the class Tuner. This results in the calling of the operation **`nextStation()`** and switches to the next station.

- ***Change the implementation of the operation `nextStation()` of class Tuner:***

```
/* -> add your code here */
if (self_currentStation < 4) {
	self_currentStation++;
}
else self_currentStation = 0;
```

- ***To show the name of the station that is currently playing, we must extend the implementation of the operation `play()` of the class Tuner.***

```
char otxt[] = "Tuner is playing %s\n";
char *msg;
msg = (char*)malloc(strlen(self_stationList[self_currentStation]) + strlen(otxt));

sprintf(msg, otxt, self_stationList[self_currentStation]);

MSG_Display_show(self_output, msg);
```

- ***Again generate the model.***
- ***Compile the project and execute it.***

You get the following output:

```
HiFi is activated
Tuner is activated
Tuner is searching
CD is activated
Tuner is playing STATION 0
Switch to next station
Tuner is playing STATION 1
Tuner is deactivated
CD is deactivated
HiFi is deactivated
```

The implementation of the classes Display, HiFi, Device, and Tuner are complete. You can now try to implement the class CD.

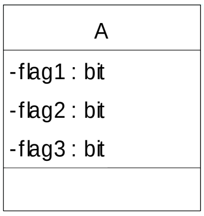
7 Optimizations

7.1 Optimizations of classes

✓ Bit attributes

Classes often have attributes representing a value that is either true or false (so-called flags). Normally, the boolean data type is used for this purpose. However, such attributes consume at least one byte. Therefore, bit coding techniques are used in embedded systems. If a class has several flag attributes, each can be coded in a bit and up to eight status flags can be combined in a single byte. Of course, a certain programming effort is necessary for the management and the access of such status flags. Therefore, Poseidon Embedded Edition provides the bit data type. Poseidon automatically combines bit attributes in bytes and generates inline access operations for each flag. Thus, a substantial amount of memory can be saved.

Example: Class with bit attributes.

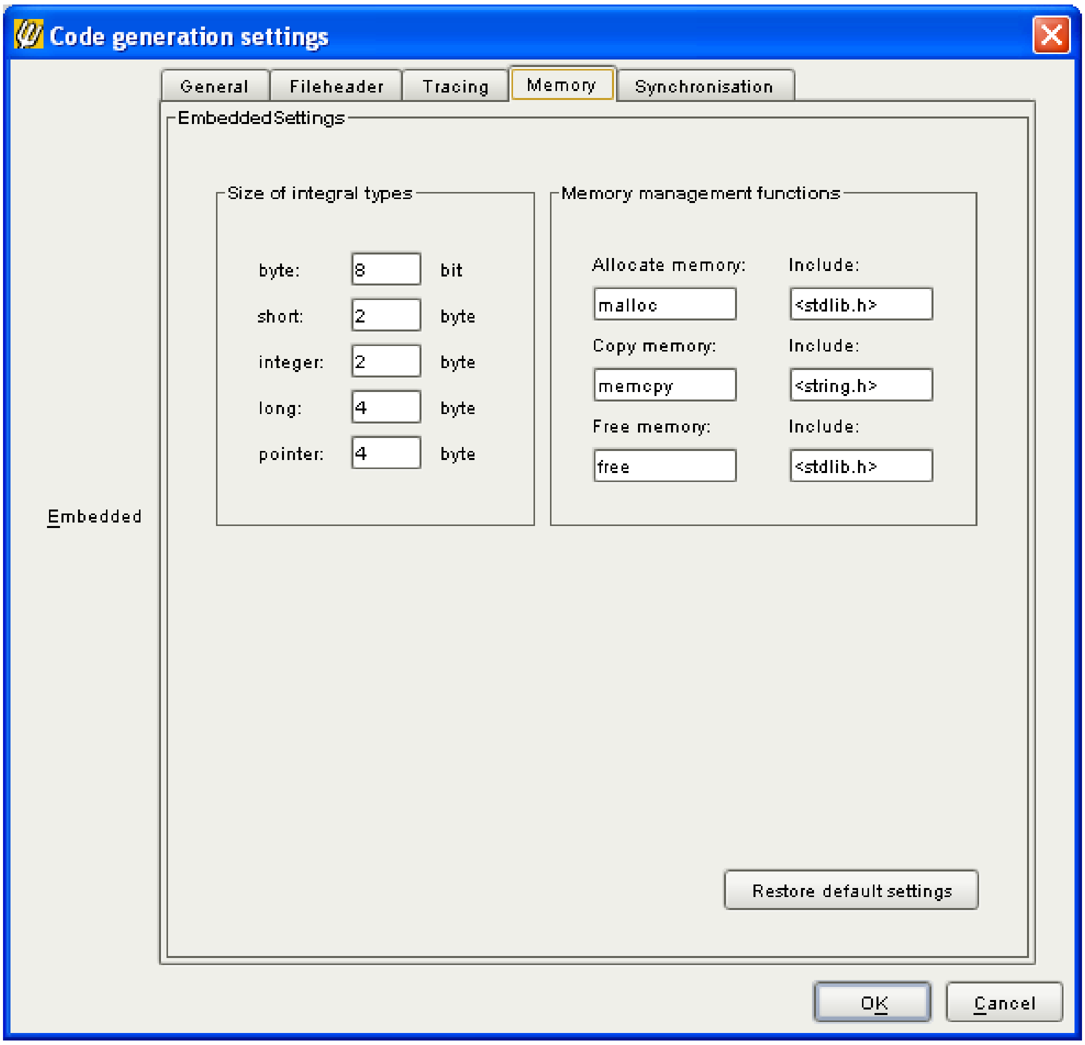


Poseidon automatically generates inline access macros for setting and getting the state of the flag:

```
MSG_A_isFlag1(obj)

MSG_A_setFlag1(obj, arg)
```

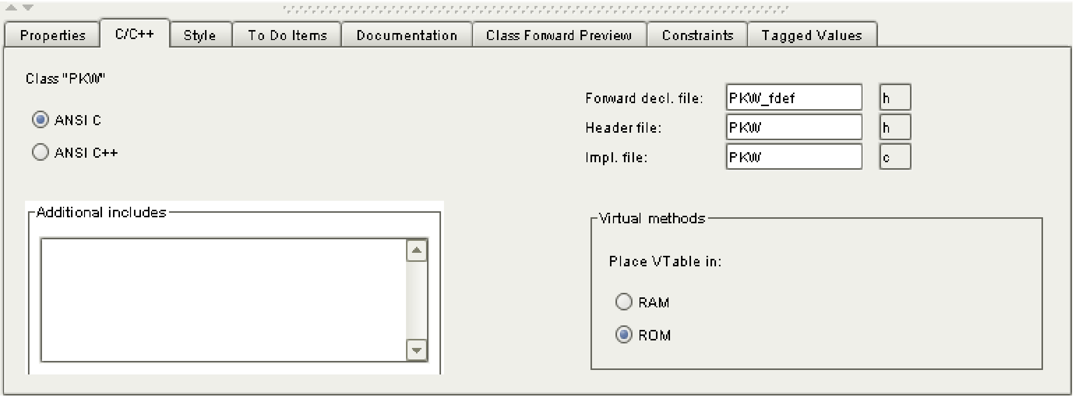
All bit attributes of a class are automatically combined in bytes. You can specify the size of a byte in the general settings of the Poseidon Embedded Edition. Choose the “Generation/ Generate Code for Classes of Model” menu entry. Click the “Settings“ button in the dialog and select the ”Memory“ tab. (Please make sure that the language is set to “Embedded”.)



Here you can specify your desired size of integral types.

✓ **Virtual methods in RAM or ROM**

If you use inheritance, virtual method tables are generated for the classes. By default, this virtual method table is stored in the RAM, but you can also store the virtual method table in the ROM. This can be changed for each class individually in the C/C++ property panel for the class.



✓ **Optimizations for operations**

abstract

These operations do not have an implementation („pure virtual" in C++). An implementation is provided in a subclass. Classes with abstract operations are abstract classes, which means that no objects of this class may be instantiated.

virtual

By default, all operations are virtual operations, i.e. are dynamically bound at runtime. Thus, dynamic polymorphism is enabled in C. The only exception is the constructor of a class, which is always non-virtual. As soon as a class has at least one virtual operation (including inherited operations) a virtual method table is created for this class.

non-virtual

These operations are bound statically. They are not listed in the method table. For the purpose of a clean UML design, such operations should not be redefined in subclasses.

Inline

These operations are generated as inline macros in the header file of the class (h-file). There are two different kinds of inline operations. Inline operation macros are automatically generated for access operations for bit attributes. These operations are not modeled in the class. For macros that are generated by using the inline modifier, the user must supply an implementation in the operation preview or in the h-file of the class. The use of inline operations is necessary in order to realize efficient access operations. However, these inline operations should be used carefully since the debugging of operations is made more difficult. Inline operations are always non-virtual.

Static

These operations are class operations. Class operations are not object-specific but bound to classes.

- ✓ **Optimizations for attributes**

static

Classes can have class attributes and class operations. These are not used via a specific object but via the class itself. All class attributes are generated in the c-file of the class. If an initial value is specified in the model, it will be used in the generated code.

const

The const modifier is not allowed for attributes, only in combination with static. If the modifiers static and const are set, the attribute is implemented as #define and an initial value is expected

- ✓ **Stereotypes for classes**

<<singleton>>

A singleton class is a class for which it is ensured that there will be only one instance object. A global point of access is provided for this object.

The singleton design pattern is often used when designing software for embedded systems. Poseidon Embedded Edition provides a special object layout for singleton classes that results in even less overhead and allows very efficient access.

In Poseidon singleton classes are specified by the <<singleton>> stereotype. There are several restrictions for singleton classes:

– Singleton classes may not have inheritance relationships to other classes.

– The constructor and the destructor of a singleton class are always private, which means that no user objects can be created.

– Singleton classes may not have virtual or abstract operations. All operations of singleton classes are non-virtual by default.

– The model may not contain “by value” associations to singleton classes.

– The model may not contain associations to singleton classes with multiplicity > 1.