Your Modeling Tool and Code Generator for Object-Oriented Embedded Systems
Kontakt:

method park Software AG
Wetterkreuz 19a
91058 Erlangen
Germany

Telefon:
+49-9131-97206-0
Fax:
+49-9131-97206-200
E-Mail:
info@methodpark.de

Version: 1.1
Date: 26-Jan-04
Author: Michael Lerch
# TABLE OF Contents

1  **INTRODUCTION** ................................................................. 9
   1.1  About this Edition .......................................................... 9
   1.2  About this Document ....................................................... 9

2  **USING POSEIDON FOR UML EMBEDDED EDITION** ................. 10
   2.1  The Work Area of Poseidon ............................................. 10
   2.2  C/C++ Properties Panel .................................................. 11
   2.3  Code Preview Panel ........................................................ 12
   2.4  Code Generation ............................................................ 13

3  **CODE GENERATION FOR CLASSES IN ANSI C** ................... 14
   3.1  Basic Mapping Scheme .................................................... 14
   3.2  Generating and using a simple class ................................ 14
      3.2.1  Using a generated class .......................................... 15
      3.2.2  The Forward Declaration of a class (fdef-file) ............... 18
      3.2.3  The declaration of a class (h-file) ............................. 19
      3.2.4  The implementation of a class (c-file) ......................... 25
   3.3  Inheritance ......................................................................... 29
      3.3.1  Using Derived Classes .............................................. 30
      3.3.2  Virtual Method Tables .............................................. 31
   3.4  Classes .............................................................................. 34
   3.5  Packages ............................................................................ 35
   3.6  Attributes ........................................................................... 36
      3.6.1  Attribute Name ......................................................... 36
      3.6.2  Multiplicity ................................................................ 36
      3.6.3  Visibility .................................................................... 36
      3.6.4  Type and Containment .............................................. 36
      3.6.5  Initial value ............................................................. 39
      3.6.6  Class (Static) Attributes .......................................... 39
      3.6.7  Constant class attributes .......................................... 40
      3.6.8  Bit Attributes ............................................................ 40
   3.7  Operations ........................................................................... 42
      3.7.1  Operation Name ......................................................... 42
      3.7.2  Visibility .................................................................... 42
      3.7.3  Parameters ............................................................... 42
6.1 General Settings.................................................................72
6.2 Fileheader Settings .................................................................73
6.3 Tracing Settings .................................................................74
6.4 Memory Settings .................................................................75
6.5 Synchronization Settings.........................................................76
LIST OF FIGURES

Figure 2-1. The work area of Poseidon.................................................................10
Figure 2-2. General Properties panel for a class................................................11
Figure 2-3. C/C++ Properties panel for a class....................................................11
Figure 2-4. Code Preview for an operation (ANSI C)........................................12
Figure 2-5. Code Generation dialog.................................................................13
Figure 3-1. Basic mapping principle.................................................................14
Figure 3-2. A simple class modeling segment displays......................................15
Figure 3-3. Files generated for a UML class (ANSI C)......................................15
Figure 3-4. Using a class with the dynamic object model.................................16
Figure 3-5. Using a class with the static object model.......................................17
Figure 3-6. The fdef-file of a class..................................................................18
Figure 3-7. FileComment section of the h-file....................................................19
Figure 3-8. IncludeGuard section of the h-file...................................................20
Figure 3-9. Include section of the h-file.............................................................20
Figure 3-10. Definitions section of the h-file: class documentation......................21
Figure 3-11. Definitions section of the h-file: declaration of operations..............22
Figure 3-12. Definitions section of the h-file: declaration of the class structure.....22
Figure 3-13. Definitions section of the h-file: macros for object creation and destruction....23
Figure 3-14. Definitions section of the h-file: façade message macros..................24
Figure 3-15. IncludeGuardEnd section of the h-file..........................................24
Figure 3-16. FileComment section of the c-file..................................................25
Figure 3-17. Include section of the c-file..........................................................25
Figure 3-18. Definitions section of the c-file.......................................................26
Figure 3-19. Constructor section of the c-file.....................................................27
Figure 3-20. Destructor section of the c-file.......................................................27
Figure 3-21. Operation sections of the c-file.....................................................28
Figure 3-22. Class Sub inherits from class Super............................................29
Figure 3-23. Inheritance with redefinition.......................................................29
Figure 3-24. Using a derived class.................................................................30
Figure 3-25. Polymorphism and dynamic binding..........................................30
Figure 3-26. Virtual method table structure for class Switch..........................31
Figure 3-27. Virtual method table structure for class DimSwitch........................31
Figure 3-28. VTable definition of class Switch..............................................32
Figure 3-29. VTable definition of class DimSwitch..........................................32
Figure 3-30. Properties panel for classes (ANSI C) ................................................................. 34
Figure 3-31. A package diagram ............................................................................................... 35
Figure 3-32. Properties panel for packages (ANSI C) .............................................................. 35
Figure 3-33. Properties panel for attributes (ANSI C) ............................................................ 36
Figure 3-34. Objects of class A contain objects of class B ....................................................... 37
Figure 3-35. Structure for class A in case of by value containment .......................................... 37
Figure 3-36. Constructor and destructor in case of by value containment ............................... 37
Figure 3-37. Structure for class A in case of by pointer containment ...................................... 38
Figure 3-38. Constructor and destructor in case of by pointer containment ........................... 38
Figure 3-39. A temperature controller with a static attribute ............................................... 39
Figure 3-40. Timer class with bit attributes .............................................................................. 41
Figure 3-41. Properties panel for parameters (ANSI C) ........................................................ 42
Figure 3-42. Properties panel for operations (ANSI C) .......................................................... 43
Figure 3-43. Class TempControl with a static operation ....................................................... 43
Figure 3-44. Implementation of an inline operation ............................................................... 44
Figure 3-45. Façade message macro generated for an inline operation ................................. 44
Figure 3-46. Classes with constructors and destructors ....................................................... 45
Figure 3-47. Association between classes A and B ............................................................... 47
Figure 3-48. Properties panel for association ends (ANSI C) ................................................. 48
Figure 3-49. Dependency between two classes ................................................................. 49
Figure 3-50. Using an object of another class as a local variable ....................................... 49
Figure 3-51. The singleton class MessageQueue ................................................................. 50
Figure 3-52. Using the singleton class MessageQueue ........................................................ 50
Figure 4-1. Files generated for a UML class (ANSI C++) ...................................................... 51
Figure 4-2. The declaration file of a class in ANSI C++. ..................................................... 53
Figure 4-3. The implementation file of a class in ANSI C++ ............................................... 55
Figure 4-4. Properties panel for classes (ANSI C++) .......................................................... 56
Figure 4-5. Properties panel for packages (ANSI C++) ....................................................... 56
Figure 4-6. A nested package diagram .................................................................................... 57
Figure 4-7. C++ class defined in nested namespaces ............................................................ 57
Figure 4-8. Properties panel for attributes (ANSI C++) ......................................................... 59
Figure 4-9. Properties panel for operations (ANSI C++) ...................................................... 60
Figure 4-10. Properties panel for parameters (ANSI C++) ................................................... 61
Figure 4-11. Properties panel for constructors (ANSI C++) ................................................ 62
Figure 4-12. Association between classes A and B ............................................................. 62
Figure 4-13. Properties panel for association ends (ANSI C++) .......................................... 63
Figure 5-1. A simple class Player.................................................................64
Figure 5-2. Statechart diagram for class Player.................................................................64
Figure 5-3. The player class with additional state machine features.................................65
Figure 5-4. Using a state machine in ANSI C.................................................................65
Figure 5-5. Tracing output from the state machine.................................................................66
Figure 5-6. Using a state machine in ANSI C++.................................................................66
Figure 5-7. Player statechart with actions.................................................................67
Figure 5-8. Transitions with guard conditions.................................................................67
Figure 5-9. Statechart with coice point.................................................................68
Figure 5-10. Player statechart with history state.................................................................68
Figure 5-11. Statechart with final sub-state and completion transition.................................69
Figure 6-1. Generation dialog with the button for opening the Generation Settings dialog....71
Figure 6-2. General Settings dialog.................................................................72
Figure 6-3. File header settings dialog.................................................................73
Figure 6-4. Tracing settings dialog.................................................................74
Figure 6-5. Memory settings dialog.................................................................75
Figure 6-6. Synchronizations settings dialog.................................................................76
1 Introduction

1.1 About this Edition

Poseidon for UML is a fully-fledged UML CASE tool. It evolved from the open-source project ArgoUML and turned it to a world class modelling tool. It is one of the industry's most important tools. The strong focus on usability makes it the easiest tool to learn and use. It is delivered in several editions to meet the needs of different users.

Poseidon for UML Embedded Edition makes it possible to use object-oriented software technology even for small embedded systems. The Embedded Edition lets you use object-oriented design with the UML and subsequent generation of ANSI C or C++ code. Thus, you can conveniently switch to the UML even when you have to use C as the target language or when you use C++ but the resources are very limited. When generating code, there is only a minimum amount of overhead – even extremely small systems for 8- and 16-bit controllers can be implemented in a highly efficient way. Since modeling stays the same, it is very easy to start with C and then switch to C++ at a later date.

Poseidon for UML Embedded Edition is a 100% UML-compliant case tool that specifically meets the requirements of embedded systems. This means:

- Support of all types of UML diagrams; very easy to use.
- Generation of highly efficient and compact ANSI C or ANSI C++ code for class and statechart diagrams.
- An integrated editor always displays a preview of the code and lets you modify the implementation of the model.
- Roundtrip engineering makes sure that source code and model are always consistent.
- The critique mechanism helps you check the model and adhere to design rules.

1.2 About this Document

This document explains all features of the C/C++ code generation component of Poseidon for UML Embedded Edition.

For general topics such as installation and registration of Poseidon for UML or editing of diagrams, please refer to the general Poseidon User Guide.
2  Using Poseidon for UML Embedded Edition

This chapter gives an overview of the user interface of Poseidon for UML Embedded Edition.

2.1  The Work Area of Poseidon

The work area of Poseidon is separated into five parts. At the top of the window, there are 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 called the overview pane and the details pane.

![Figure 2-1. The work area of Poseidon.](image)

The details pane is composed of a number of different panels that can be selected through corresponding tabs at the top of the pane. They are used to browse, enter, or change more detailed information of the model. This includes information that might not even be visible in the diagrams. Please refer to the Poseidon Users Guide for more information about the particular panes.

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.
2.2 C/C++ Properties Panel

The UML properties of a model element such as its name or visibility are specified in the general Properties panel of Poseidon. All element properties that are specific for C/C++ code generation are combined in the C/C++ Properties panel.

The C/C++ Properties panel lets you make further target-language specific settings for the following model elements:

- packages
- classes
- attributes
- operations
- parameters
- association ends

The panels are described in detail in chapters 3 and 4.
2.3 Code Preview Panel

The details pane of Poseidon contains two Code Preview panels, one for C/C++ and one for Java. Please refer to the Poseidon Users Guide for information about the Java Preview panel.

The C/C++ Code Preview panel displays the source code that will be generated for the currently selected model element. Every time a new model element is selected – e.g. in the navigation or the diagram pane – the corresponding source code is generated on the fly and updated in the Code Preview pane. The source code is not written to the file system at that time.

The Code Preview is available for classes, operations, and attributes. If an operation is selected, its implementation can be edited in the Code Preview:

![Code Preview](image)

**Figure 2-4. Code Preview for an operation (ANSI C).**

The parts of the source code that may not be modified are protected by the editor. Alternatively, the code can be edited externally. Poseidon for UML Embedded Edition can be configured to automatically synchronize between the model and the source code.

Several source files will be generated for a UML class (see chapters 3 and 4). Thus, the Code Preview for a class provides different views. You can switch to the next view using the menu entry **View > Next Source**. Alternatively, the shortcut **ALT-N** can be used.
2.4 Code Generation

The source code is generated to the file system using the Code Generation Dialog. It can be opened using the Generation>Embedded menu entry.

![Image of the Code Generation dialog]

**Figure 2-5. Code Generation dialog.**

The model tree in the upper part of the dialog lets you select the elements for the next generation step.
3 Code Generation for Classes in ANSI C

Poseidon for UML Embedded Edition can generate ANSI C code for the class and state diagrams of your UML model. Since C is not an object-oriented language, it is necessary to map the object-oriented features in the model to the target language. Of course, the restrictions of small embedded systems must be taken into account. This chapter describes the underlying mapping concept for ANSI C and possible optimizations.

3.1 Basic Mapping Scheme

Basically, Poseidon maps an UML class to a C structure. The attributes of the class become members of the structure. The operations of a class are mapped to C functions with an additional first parameter that represents the object for which the corresponding operation is called.

As you can see, the functions are prefixed by the class name. This shows that a function represents an operation of a class. Furthermore, name conflicts are avoided.

As already mentioned, the parameter self references the object of the class for which the function (i.e. operation) has been called. Usually, e.g. in C++ or Java, the currently active object is named this. The name self was chosen in Poseidon in order to avoid conflicts when the generated code must be used in combination with C++.

The reason for this will become clear when inheritance and dynamic method calls come into play (see section 3.3).

Within this setting, objects of a class can be created by defining variables of the corresponding structure, and operations can be executed for an object by calling the corresponding C function with the object variable as the first argument.

3.2 Generating and using a simple class

Of course, the mapping principle described in the previous section is simplified and not very handy when it comes to using the generated code. This section explains how classes can be used in ANSI C and how the code that is generated for a class works. We only show a simple standalone class here. Advanced topics are covered step by step in the following sections.

Let’s have a look at a simple class representing segment displays:
Figure 3-2. A simple class modeling segment displays.

The class **SegmentDisplay** has an attribute **portAddr** that determines the hardware port address of the particular display object. The first operation **SegmentDisplay**, which has the same name as the class, is a so-called constructor and is responsible for the correct initialization when an object is created. The task of the two remaining operations is to clear a specific display and to show a digit, respectively.

The Poseidon ANSI C code generator creates three files for each UML class:

![Files generated for a UML class (ANSI C).](image)

3.2.1 Using a generated class

We first take a look at how a class can be used provided that it has been generated with Poseidon. Using a class essentially comprises

- declaring, creating, and destroying objects of a class; and
- executing operations of these objects.

Concerning the creation, lifetime, and destruction of objects two variants are available: the dynamic and the static object model.

Dynamic objects are declared as pointer variables and are allocated on the heap. They must be destroyed explicitly. The constructor is called implicitly with the object creation, the destructor with the object destruction.

Static objects are declared as regular variables and usually reside on the stack of the enclosing function. Of course, there may be global instances as well. Static objects must be initialized (constructor call) and deinitialized (destructor call) explicitly.
The following code shows the usage of the class `SegmentDisplay` based on the dynamic object model:

```c
/* include the class declaration */
#include "SegmentDisplay.h"

void dynamicUsage() {
    /* declare an object */
    SegmentDisplay *segDisp;

    /* dynamically create an object */
    NEW_SegmentDisplay(segDisp, P2);

    /* use an object */
    MSG_SegmentDisplay_clear(segDisp);
    MSG_SegmentDisplay_showDigit(segDisp, 7);

    /* destroy an object */
    DELETE_SegmentDisplay(segDisp);
}
```

**Figure 3-4. Using a class with the dynamic object model.**

In order to use a class, its declaration file must be included (line 2). A dynamic object variable is declared in line 7. Line 10 shows how an object is dynamically created. You simply use the macro command `NEW_SegmentDisplay`, which is automatically generated by Poseidon. The first argument of a `NEW` macro is always a pointer to the object that is to be created. The following arguments are passed to the constructor of the class. In our example, a value representing the port address of the segment display is expected. The `NEW` macro first allocates the necessary memory and then calls the constructor of `SegmentDisplay`. In lines 13 and 14, the operations `clear` and `showDigit` are called for the object `segDisp`. This is done via so-called *façade message macros*. These macros hide the actual calling mechanism. In this case, a simple function call will be performed. The first argument of the message macros is the object for which the corresponding operation should be executed. Further arguments are mapped to additional operation parameters. Finally, in line 17, the object is destroyed, i.e. the destructor of the class is called first and the allocated memory is freed.
The following code shows how to work with classes in the static object model:

```c
/* include the class declaration */
#include "SegmentDisplay.h"

void staticUsage() {

    /* declare an object */
    SegmentDisplay segDisp;

    /* initialize the object */
    INIT_SegmentDisplay(&segDisp, P2);

    /* use the object */
    MSG_SegmentDisplay_clear(&segDisp);
    MSG_SegmentDisplay_showDigit(&segDisp, 7);

    /* deinitialize the object */
    DEINIT_SegmentDisplay(&segDisp);
}
```

Figure 3-5. Using a class with the static object model.

In line 7 a static object is declared. In ANSI C this is also the point of definition for the object and memory is reserved on the stack accordingly. Since variables are not initialized in C, an explicit initialization is necessary. This is done in line 10 using the `INIT` macro. The macro performs a constructor call and expects the address of the object as the first argument. Again, the message macros are used to call operations for the object in lines 13 and 14. The only difference is that in this case object addresses must be passed. Since `segDisp` is an automatic object (regarding its storage class), it is discarded at the end of the function. Therefore, an explicit deinitialization is necessary that calls the destructor of the class. Line 17 shows the use of the `DEINIT` macro.

The following sections describe in detail the structure of the code generated for a simple class.
3.2.2 The Forward Declaration of a class (fdef-file)

The first file that is generated for a class (*SegmentDisplay_fdef.h* in the example) is the so-called *fdef-file*. It essentially contains a macro definition for the generated structure type of the class. Thus, the class name can directly be used as a type name for variables. The macro definition also serves as a kind of forward declaration of the class. Thus, circular references between classes can be expressed without problems.

```c
/*@ <FileComment ID=1052449171725> */
/***************************************************************************/
* Class      : SegmentDisplay
* File       : SegmentDisplay_fdef.h
* Generated with : Poseidon for UML EmbE 1.6
* Last generation: Fri May 09 04:59:31 CEST 2003
/***************************************************************************/
/*@ </FileComment ID=1052449171725> */

/*@ <IncludeGuard> */
#ifndef SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H_FDEF_H
#define SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H_FDEF_H
/*@ </IncludeGuard> */

/*@ <Definitions> */
#define SegmentDisplay struct _PE_SegmentDisplay
/*@ </Definitions> */

/*@ <IncludeGuardEnd> */
#endif  /* SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H_FDEF_H */
/*@ </IncludeGuardEnd> */
```

Figure 3-6. The fdef-file of a class.

Each file generated by Poseidon is made up of several protected sections. These sections are surrounded by special comments, which contain tags indicating the purpose of the section, e.g.:

```c
/*@ <IncludeGuard> */

#ifdef SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H_FDEF_H
define SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H_FDEF_H
/*@ </IncludeGuard> */

/*@ <Definitions> */
#define SegmentDisplay struct _PE_SegmentDisplay
/*@ </Definitions> */

/*@ <IncludeGuardEnd> */
#endif  /* SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H_FDEF_H */
/*@ </IncludeGuardEnd> */
```

The protected code contained in the sections is re-generated on every code generation step. Thus, it does not make sense to modify these sections since all modifications will be lost after the next generation. In the Code Preview Panel of Poseidon, it is not even possible to edit the protected sections. In contrast, outside the protected regions arbitrary user code is allowed and will be preserved. Operation sections (see 3.2.4.6) are handled in a special way. For further details about synchronization between model and file system see section 6.5.

The fdef-file of a class is made up of the following sections:
3.2.2.1 FileComment section

The FileComment section contains a comment describing the purpose of the file. The appearance and the contents of that section can be configured in the code generation dialog (see 6.2).

3.2.2.2 IncludeGuard section

The IncludeGuard section makes sure that the file is included only once per compilation unit. The name of the include guard is determined by the class name and a unique identifier.

3.2.2.3 Definitions section

The definitions section contains a macro definition that constitutes the forward declaration of the class.

3.2.2.4 IncludeGuardEnd section

This section terminates the include guard.

3.2.3 The declaration of a class (h-file)

The h-file generated for a class (SegmentDisplay.h in the example) contains the declaration of the class, including all its attributes and operations. Furthermore, in this file a number of macros are generated that are necessary for a convenient use of the class.

The following paragraphs explain each protected section of the h-file in detail.

We’d like to point out again that the code in the protected sections may not be modified. All code added between protected sections or at the beginning or the end of the file will be maintained by the generator.

3.2.3.1 FileComment section

```
/*@ <FileComment ID=1052449171725> */
/****************************************************************************
* Class          : SegmentDisplay
* File           : SegmentDisplay.h
* Generated with : Poseidon for UML EmbE 1.6
* Last generation: Fri May 09 04:59:31 CEST 2003
****************************************************************************/
/*@ </FileComment ID=1052449171725> */
```

Figure 3-7. FileComment section of the h-file.

The FileComment section contains a comment describing the purpose of the file. The appearance and the contents of that section can be configured in the code generation dialog (see section 6.2).
3.2.3.2 IncludeGuard section

```c
/*@ <IncludeGuard> */
#ifndef SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H
#define SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H
/*@ </IncludeGuard> */
```

Figure 3-8. IncludeGuard section of the h-file.

The IncludeGuard section make sure that the file is included only once per compilation unit. The name of the include guard is determined by the class name and a unique identifier.

3.2.3.3 Include section

```c
/*@ <Include> */
#include <stdlib.h>
#include <string.h>
#include "SegmentDisplay_fdef.h"
/*@ </Include> */
```

Figure 3-9. Include section of the h-file.

The Include section lists all `#include` directives that are automatically derived from the model:

- `#include` of standard headers. These may be needed for example for memory allocation. In the example code above `stdlib.h` and `string.h` are included. Of course, the use of these files is not prescribed. You can specify the appropriate library include files for your target system in the code generation settings dialog (see section 6.4).

- `#include` of the forward declaration of the class itself

More sophisticated classes may also contain the following `#include` directives:

- `#include` of the declaration of the base class if this class is a derived class.

- `#include` of declarations of classes used by this class either as an attribute type, a parameter type, an association, or an explicitly modeled dependency.

- Additional `#includes` that are specified in the C/C++ properties panel for this class.
3.2.3.4 Definitions section

The Definitions section is by far the largest and also the most important section of the h-file.

The first part is a comment block describing the interface of the class. The documentation of
the attributes and operations is taken from the model.

```c
/*@ <Definitions> */
/* Documentation of class SegmentDisplay */

/* Constructor: */
* ------------
*     Create(u16 * port) : void
*
/* Destructor: */
* -----------
*     Destroy() : void
*
/* Methods: */
* --------
*     // Clears this segment display.
*     + clear() : void
*
*     // Shows a digit (number between 0 and 9) on this segment display.
*     + showDigit(u8 digit) : void
*
/* Attributes: */
* -------------
*     // Represents the hardware port address of this segment display.
*     - portAddr : u16 *
*/
```

Figure 3-10. Definitions section of the h-file: class documentation.
The following part contains a declaration for the constructor, the destructor and every operation of the class:

```c
void _PE_SegmentDisplay_Create(void * _self, u16 * port);

void _PE_SegmentDisplay_Destroy(void * _self, unsigned char isSuper);

/**
* Clears this segment display.
*/
void SegmentDisplay_clear(void * _self);

/**
* Shows a digit (number between 0 and 9) on this segment display.
*/
void SegmentDisplay_showDigit(void * _self, u8 digit);
```

Figure 3-11. Definitions section of the h-file: declaration of operations.

The function representing the constructor of the class is called `Create`. Correspondingly, the destructor is called `Destroy`. The constructor and the destructor of a class are always generated. Even if they are not modelled explicitly, implicit default versions are added to the implementation. The names of the functions generated for the constructor and the destructor are internal names. All internal names are prefixed by the string `_PE_`, which stands for Poseidon Embedded.

As already mentioned, the object for which an operation is called is passed to the corresponding function via an untyped pointer (i.e. `void *`) as the first argument.

An internal macro for the list of all attributes of the class is defined in the next section (`_PE_SegmentDisplay_ATTR` in our example). The macro is then used for the definition of the class’ structure. The reason why an internal attribute list macro is defined becomes clear when a class is derived from another class by inheritance: the base class attribute macro can then be used in order to correctly define the attribute list of the derived class (see section 3.3.2).

```c
/****************************
* Attributes of class SegmentDisplay *
****************************/
#define _PE_SegmentDisplay_ATTR
ul6 * portAddr;

struct _PE_SegmentDisplay {
 _PE_SegmentDisplay_ATTR
};
```

Figure 3-12. Definitions section of the h-file: declaration of the class structure.
The following part provides the user with macros for creation and destruction of objects of the class. The creation of an object comprises the call of the constructor function. Correspondingly, the destruction of an object comprises the call of the destructor function. There are two versions of the macros, one for dynamic and one for static usage of the object.

```c
/**********************************************
* Macros for constructor and destructor calls *
**********************************************/
#define NEW_SegmentDisplay(obj, port)  
{ if ((obj=(SegmentDisplay*)malloc(sizeof(SegmentDisplay))) != NULL)  
  _PE_SegmentDisplay_Create(obj, port);  
}
#define INIT_SegmentDisplay(obj, port)  
  _PEE_SegmentDisplay_Create(obj, port);
#define DELETE_SegmentDisplay(obj)  
{ if (obj != NULL) {  
  _PEE_SegmentDisplay_Destroy(obj, 0);  
  obj = NULL;  
}  
}
#define DEINIT_SegmentDisplay(obj)  
{ if (obj != NULL)  
  _PEE_SegmentDisplay_Destroy(obj, 1);  
}
```

Figure 3-13. Definitions section of the h-file: macros for object creation and destruction.

The address of an object of the class is always expected as the first parameter.

The **NEW** macro dynamically creates an object of the class. It first tries to allocate memory and then calls the constructor on success. In the example, the ANSI C standard function `malloc` is used. Target system-specific memory allocation and deallocation functions may be used instead (see section 6.4).

The **INIT** macro initializes a static object by calling the constructor of the class.

The **DELETE** macro is used for the destruction of a dynamic object and calls the destructor of the class. In contrast to the **NEW** macro, the memory is not directly deallocated by the **DELETE** macro. This happens in the destructor function in order to guarantee that in inheritance relations all base class destructors are called before the memory is released.

The **DEINIT** macro deinitializes static objects and calls the destructor without deallocating any memory.
The last part of the Definitions section provides the definition of the façade message macro for each operation of the class. In our example, the expansion of the macros will simply result in calling the corresponding C functions. Later, when dynamically bound operations or inline operations are addressed, you will see that other implementations of the message macros are possible. It is a crucial point of the concept that operations are called solely using the façade message macros. This guarantees that the calling mechanism of a specific operation is hidden. It can be changed later without breaking existing client code.

```
/**********************
 * Macros for method calls *
 ***********************/
#define MSG_SegmentDisplay_showDigit(obj, digit)  \
    SegmentDisplay_showDigit(obj, digit)

#define MSG_SegmentDisplay_clear(obj)  \
    SegmentDisplay_clear(obj)
```

Figure 3-14. Definitions section of the h-file: façade message macros.

3.2.3.5 IncludeGuardEnd section

```
/*@ <IncludeGuardEnd> @*/
#endif  /* SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H */
/*@ </IncludeGuardEnd> @*/
```

Figure 3-15. IncludeGuardEnd section of the h-file.

This section terminates the include guard.
3.2.4 The implementation of a class (c-file)

The c-file of a class (\texttt{SegmentDisplay.c} in the example) contains the implementation of the class. A c-file may contain two additional sections that are not discussed at this point: SingletonMethods (see 3.10) and StateMachine (see 5).

3.2.4.1 FileComment section

```c
/*@ <FileComment ID=1052913452468> @*/
/*******************************************************************************/
/* Class : SegmentDisplay */
/* File : SegmentDisplay.c */
/* Generated with : Poseidon for UML EmbE 1.6 */
*******************************************************************************/
/*@ </FileComment ID=1052913452468> @*/
```

Figure 3-16. FileComment section of the c-file.

The FileComment section contains a comment describing the purpose of the file. The appearance and the contents of that section can be configured in the code generation dialog (see 6.2).

3.2.4.2 Include section

```c
/*@ <Include> @*/
#include "SegmentDisplay.h"
/*@ </Include> @*/
```

Figure 3-17. Include section of the c-file.

The Include section lists all \#include directives that are automatically derived from the model. First of all, the h-file of the class is included. More \#include directives are generated for dependent classes.
3.2.4.3 Definitions section

```c
/*@ <Definitions> @*/

#ifndef _IMPLICIT_SELF
#define DEFINESELF \\
    SegmentDisplay *self = (SegmentDisplay *)_self;
#else
#define DEFINESELF
#define self \\
    ((SegmentDisplay*)_self)
#endif

/**********************************
* Macro definition for attributes *
**********************************/
#define self_portAddr self->portAddr

/*@ </Definitions> @*/
```

Figure 3-18. Definitions section of the c-file.

This section defines some macros that are needed to implement the bodies of the operation. The `DEFINESELF` macro is used as a means to make the `self`-object accessible in the body. It is generated at the beginning of every operation body. The effect of the macro is that the identifier `self` with type “pointer to class structure” can be used in the operation body. It was previously mentioned that the `self`-object is passed to an operation as a parameter with type `void *` in order to realize polymorphism. The `DEFINETHIS` macro applies a type cast to this parameter. Thus the `self` object can be used with the correct type.

The implementation of the `DEFINESELF` macro depends on the definition of the macro `IMPLICIT_SELF`. If it is not defined, `self` becomes an explicit local variable of the function. This is especially useful for debugging purposes. If `IMPLICIT_SELF` is defined, then every use of `self` is expanded directly to a cast to the appropriate type. This setting results in lower stack consumption and is reasonable for the final production code. The `IMPLICIT_SELF` macro must be defined by the user, e.g. in a common include file.

The last part of the Definitions section defines a façade access macro for every attribute of the class. In analogy to the façade message macros, the purpose of those macros is that the attributes of a class can be easily and uniformly used in the implementation of operations. It is crucial to use the attribute access macros instead of directly addressing the members of the class structure, since other object implementations may be deployed (e.g. singleton classes). In our example the class `SegmentDisplay` has a single attribute `portAddr`. The façade access macro `self_portAddr` is generated for this attribute.
3.2.4.4 Constructor section

```c
/*@ <Constructor ID=10-10-1--41-1d41116:f507f54250:-7fd9> @*/
/**
 * Constructor.
 */
void _PE_SegmentDisplay_Create(void * _self, u16 * port) {
    DEFINESELF
    /* -> add your code here */
   /*@ <Init> @*/
   /*@ </Init> @*/
    /* -> add your code here */
    self_portAddr = port;
}
/*@ </Constructor ID=10-10-1--41-1d41116:f507f54250:-7fd9> @*/
```

Figure 3-19. Constructor section of the c-file.

The constructor section contains the function that is generated for the constructor of the class. It has a special Init subsection, which happens to be empty in our example. In the Init section you will find code that initializes the attributes of a class according to the default values specified in the model. Apart from the Init section, user code may be added anywhere in the body of the function. In the example the parameter `port` is assigned to the attribute `portAddr` using the attribute access macro.

3.2.4.5 Destructor section

```c
/*@ <Destructor> @*/
void _PE_SegmentDisplay_Destroy(void * _self, unsigned char isSuper) {
    DEFINESELF
    /* Auto-generated default destructor. Do not edit! */

    if( isSuper == (unsigned char)0 /*false*/ ) {
        free(self);
    }
}
/*@ </Destructor> @*/
```

Figure 3-20. Destructor section of the c-file.

The function generated for the destructor of the class is defined in the Destructor section. Since the model did not specify a destructor for the class, the destructor function is generated automatically and no user code may be added. If you need to add class-specific deinitialization code, you will have to model the destructor explicitly. A destructor is an special operation with the name of the class, prefixed by a tilde character, e.g. `~SegmentDisplay`. If inheritance is used, the parameter `isSuper` and the memory
deallocation code make sure that all base class destructors are called before the memory is actually freed.

### 3.2.4.6 Operation section

```c
/*@ <Operation ID=10-10-1--41-1d41116:f507f54250:-7fe0> @*/
/**
 * Clears this segment display.
 */
void SegmentDisplay_clear(void * _self) {
    DEFINESELF
    /* -> add your code here */
    *self_portAddr |= 0x00FF;
}
/*@ </Operation ID=10-10-1--41-1d41116:f507f54250:-7fe0> @*/

/*@ <Operation ID=10-10-1--41-1d41116:f507f54250:-7fdd> @*/
/**
 * Shows a digit (number between 0 and 9) on this segment display.
 */
void SegmentDisplay_showDigit(void * _self, u8 digit) {
    DEFINESELF
    /* -> add your code here */
    *self_portAddr |= (0xFF00 & digit);
}
/*@ </Operation ID=10-10-1--41-1d41116:f507f54250:-7fdd> @*/
```

Figure 3-21. Operation sections of the c-file.

The Operation sections are the most important parts of the implementation of a class. Initially, a section with an empty function code frame is generated for each operation of the class. The bodies of the operations must be implemented by the user either directly in Poseidon or in any other editor. Arbitrary user code may be added inside the function. The user code will be preserved in future generation steps.
3.3 Inheritance

In the object-oriented paradigm, inheritance (often called generalization) is a very important relationship between classes. A class Sub that inherits from another class Super inherits all features (i.e. attributes and operations) from Super and may extend Super by additional features. An operation in the base class Super may also be redefined in the derived class Sub, which means that it gets a new implementation.

![Class Diagram](image)

Figure 3-22. Class Sub inherits from class Super.

This mechanism forms the basis for polymorphism and dynamic binding: an object of the class Sub may be used whenever an object of class Super is expected. If a dynamically bound operation is called via a base class object reference, the decision, which operation is to be executed, is made at runtime. Such operations are also called virtual operations.

On one hand, inheritance and dynamic binding are the key to extensibility and reusability. On the other hand, these powerful mechanisms require additional resources, both memory and runtime. The ANSI C code generator of Poseidon supports single inheritance in a way that minimizes the overhead.

We consider the following example to show how inheritance is used and implemented:

![Class Diagram](image)

Figure 3-23. Inheritance with redefinition.

The class Switch has an attribute state and two operations on and off. The class DimSwitch introduces an additional attribute position. It redefines the behavior of the
operation `off` and extends `Switch` by an operation `setPosition`. The operation `off` will be bound dynamically, whereas `on` and `setPosition` can be regular C functions.

### 3.3.1 Using Derived Classes

The class `DimSwitch` can be used like a regular class:

```c
/* define and create a dynamic DimSwitch object*/
DimSwitch *dsw;
NEW_DimSwitch(dsw);

/* call operations */
MSG_DimSwitch_on(dsw);
MSG_DimSwitch_setPosition(dsw, 42);
MSG_DimSwitch_off(dsw);

/* destroy DimSwitch object */
DELETE_DimSwitch(dsw);
```

Figure 3-24. Using a derived class.

Please note that the façade message macros are generated also for inherited operations like `on`. The actual calling mechanism is hidden by using the message macros.

The following code shows how polymorphism is used:

```c
/* declare a base class object variable */
Switch *sw;

/* create a DimSwitch object and let sw reference it */
VIRTUALNEW_DimSwitch(Switch, sw);

/* call operation */
MSG_Switch_on(sw); /* -> Switch::on() is called */
MSG_Switch_off(sw); /* -> DimSwitch::off() is called */

/* destroy DimSwitch object */
DELETE_Switch(sw);
```

Figure 3-25. Polymorphism and dynamic binding.

The `VIRTUALNEW` macro allows the polymorphic creation of objects. A `DimSwitch` object is created and its address is assigned to the `Switch` object pointer `sw`. The operations `on` and `off` are called via the base class pointer `sw`. Since the operation `off` is a virtual operation, the macro call `MSG_Switch_off` effectively executes the operation `off` from class `DimSwitch`. 
3.3.2 Virtual Method Tables

This section shows how inheritance and dynamic binding are implemented by the code generator. The example from Figure 3-23 is used for this purpose.

With every generation step, the optimizer analyzes the model. Only operations that are redefined in an inheritance hierarchy are marked as virtual operations. All other operations may safely be implemented as regular functions without any additional overhead. If necessary, the default generation mode can be overridden for each individual operation by specifying another binding method (see 3.7).

If the set of all virtual operations is known, a so-called virtual method table (VTable) is built for each class that has at least one virtual method. A virtual method table is a structure with a function pointer entry for each virtual method of the class. The VTable structure of a class is declared in the Definitions section of its h-file:

```
/****************************
* Virtual method table structure of class Switch *
****************************/

#define _PE_Switch_vTableDef
 void (*_PE_destroy)(void * _self, unsigned char isSuper); \
 void (*off)(void * _self);

typedef struct {
     _PE_Switch_vTableDef
 } _PE_Switch_vTableStr;
```

Figure 3-26. Virtual method table structure for class Switch

Clearly, there is an entry for the virtual operation off. There is also an entry for the destructor since both classes have implicit destructors. Of course, the binding of destructors can be overridden.

The VTable for the derived class DimSwitch has the same structure, since no further virtual operations are added:

```
/****************************
* Virtual method table structure of class DimSwitch *
****************************/

#define _PE_DimSwitch_vTableDef
 _PE_Switch_vTableDef

typedef struct {
     _PE_DimSwitch_vTableDef
 } _PE_DimSwitch_vTableStr;
```

Figure 3-27. Virtual method table structure for class DimSwitch.

Exactly one VTable variable per class is defined in the Definitions section of the c-file. The function pointers are set correctly:
const _PE_Switch_vTableStr _PE_Switch_MT = {
    _PE_Switch_Destroy,
    Switch_off
};

Figure 3-28. VTable definition of class Switch.

const _PE_DimSwitch_vTableStr _PE_DimSwitch_MT = {
    _PE_DimSwitch_Destroy,
    DimSwitch_off
};

Figure 3-29. VTable definition of class DimSwitch.

The figures reflect the code that is generated when the user decides to place the VTables in the ROM. Alternatively, VTables may be placed in RAM (see 3.4).

In order to use the dynamic binding mechanism, each object must know the VTable of its class. This means that for each object of a class with virtual operations an additional implicit attribute is generated. The value of this attribute is initialized to the address of the VTable in the constructor.

The following code snippets show the definitions of the structures for the classes Switch and DimSwitch, respectively.

/*****************************
* Attributes of class Switch *
*******************************/
/* definition of attributes */
#define _PE_Switch_ATTR
    unsigned char state;

struct _PE_Switch {
    const _PE_Switch_vTableStr * _PE_vMethods;
    _PE_Switch_ATTR
};
The façade message macros for virtual methods are defined such that the function is called indirectly using the corresponding VTable entry:

```c
#define MSG_Switch_off(obj)  
    ((obj)->_PE_vMethods->off(obj))
```
3.4 Classes

A UML class is the central unit of code generation. The target language of a class (C or C++) can be selected in the C/C++ Properties panel. The default target language can be set on the generation settings dialog (see section 6.1).

The C/C++ Properties panel for the target language ANSI C looks like this:

![Figure 3-30. Properties panel for classes (ANSI C).](image)

The following settings can be made:

- **Target language.** ANSI C or ANSI C++ can be selected as the target language for this class.

- **Additional includes.** `#include` directives that are not derived from the model can be specified here. The text that is entered in this area is copied verbatim into the Include section of the declaration file (h-file) of the class.

- **Target file names.** By default, the names of the three target files are derived from the class name. If necessary, the base names of the files can be overwritten here. The file extensions can be specified globally in the generation settings dialog (see 6.1). The target file path can be specified globally in the code generation dialog and for any enclosing package in the C/C++ Properties panel for packages (see 3.5).

- **VTable placement.** There are two variants for the implementation of the virtual method table (VTable) mechanism. The VTable can either be placed in the RAM – in this case the VTable is built during the first constructor call of the class – alternatively, the VTable can be placed in the ROM, which means that it is constructed statically.
3.5 Packages

Packages are useful for structuring the classes in logical units.

The code generator can use packages to determine the target path of all enclosed classes:

- **Namespace.** If checked, this package constitutes a namespace. This setting does not have any effect on the ANSI C code generation.

- **Target path.** A target path for all enclosed classes can be specified. You can enter an absolute path here. Alternatively, the path variables $\text{$\text{ROOT}$}$ and $\text{$\text{PARENT}$}$ can be used at the beginning of the path. $\text{$\text{ROOT}$}$ refers to the global output folder specified in the code generation dialog. $\text{$\text{PARENT}$}$ refers to the output path specified for the parent package of this package. The following default settings are applied when a new package is created:
  - $\text{$\text{ROOT}$/<package name>}$ for top-level packages
  - $\text{$\text{PARENT}$/<package name>}$ for nested packages
3.6 Attributes

![Properties panel for attributes (ANSI C).](image)

3.6.1 Attribute Name

The name of an attribute is specified in the general Properties panel of Poseidon. The attribute name should satisfy the ANSI C rules for identifiers, because it is used as an identifier in the source code. For convenient use, spaces in the attribute name are replaced by underscore characters.

3.6.2 Multiplicity

The multiplicity of an attribute can be specified in the general Properties panel of Poseidon. Multiplicities can be used for generating arrays. Only limited numbers and ranges are allowed. The upper bound of a multiplicity is taken as the dimension of the resulting array. Unlimited ranges such as \([0..\star]\) are not considered by the code generator. Associations should be used for this purpose.

3.6.3 Visibility

The visibility of an attribute can be specified in the general Properties panel of Poseidon. Only the visibility levels `public`, `protected`, and `private` can be used. Unfortunately, the C programming language does not offer an effective mechanism for prohibiting the access to a member of a structure. On the other hand, no visibility violations will occur if you obey the rule that the attributes of a class should be accessed only using the façade access macros. These macros are defined only in the implementation file (c-file) of a class and thus can not be used externally.

3.6.4 Type and Containment

The type of an attribute can be specified in the general Properties panel of Poseidon. By default, the type `int` is chosen. Other types may be selected from the drop-down list. If a new non-class data type is needed, it can be created on the Properties panel for the model (topmost node in the navigation pane).

The C/C++ Properties panel provides an additional option for attributes, which is called containment. This property determines the way how an attribute is contained in an object of its class. For the target language C, possible values for the containment of an attribute are `by value` or `by pointer`. `By value` is selected by default. In this case the type of the resulting structure member is taken verbatim from the model. If the containment mode `by pointer` is chosen, an additional pointer declaration is added to the specified type.
The following example illustrates the difference:

```
A
B
```

**Figure 3-34.** Objects of class A contain objects of class B.

If the containment option for attribute \( b \) of class \( A \) is set to *by value*, every object of class \( A \) has a nested static object of class \( B \):

```c
#define _PE_A_ATTR \
  B b;

struct _PE_A {
  _PE_A_ATTR
};
```

**Figure 3-35.** Structure for class A in case of *by value* containment.

Please note, that the nested object \( b \) needs to be initialized. The best place to do this is in the constructor of class \( A \). Correspondingly, \( b \) should be deinitialized in the destructor of \( A \) if an object of \( A \) is destroyed.

We show a sample implementation here:

```c
/*@ <Constructor ID=127-0-0-1-1756456:f541dd6d3a:-7ffc> @*/
void _PE_A_Create(void * _self) {
  DEFINESELF
 /*@ <Init> @*/
  INIT_B(&self_b);
 /*@ </Init> @*/
}

/*@ </Constructor ID=127-0-0-1-1756456:f541dd6d3a:-7ffc> @*/

/*@ <Destructor ID=127-0-0-1-1756456:f541dd6d3a:-7ff9> @*/
void _PE_A_Destroy(void * _self, unsigned char isSuper) {
  DEFINESELF
  DEINIT_B(&self_b);
  if( isSuper == (unsigned char)0 /*false*/ ) {
    free(self);
  }
}

/*@ </Destructor ID=127-0-0-1-1756456:f541dd6d3a:-7ff9> @*/
```

**Figure 3-36.** Constructor and destructor in case of *by value* containment.
On the other hand, if *by pointer* containment was chosen, an object of class **A** object just contains a reference (i.e. pointer) to some **B** object:

```
#define _PE_A_ATTR
    B * b;

struct _PE_A {
    _PE_A_ATTR
};
```

**Figure 3-37. Structure for class A in case of by pointer containment.**

It is the responsibility of the programmer to ensure that \(b\) really points to an object of class **B**. For instance, a new **B** object could be created in the constructor and destroyed again in the destructor:

```
/*@ <Constructor ID=127-0-0-1-1756456:f541dd6d3a:-7ffc> @*/
void _PE_A_Create(void * _self) {
    DEFINESELF
    /*@ <Init> @*/
    /*@ </Init> @*/

    NEW_B(self_b);
}
/*@ </Constructor ID=127-0-0-1-1756456:f541dd6d3a:-7ffc> @*/

/*@ <Destructor ID=127-0-0-1-1756456:f541dd6d3a:-7ff9> @*/
void _PE_A_Destroy(void * _self, unsigned char isSuper) {
    DEFINESELF
    DELETE_B(self_b);
    if( isSuper == (unsigned char)0 /*false*/ ) {
        free(self);
    }
}
/*@ </Destructor ID=127-0-0-1-1756456:f541dd6d3a:-7ff9> @*/
```

**Figure 3-38. Constructor and destructor in case of by pointer containment.**

It should be mentioned that using associations is an alternative and more explicit way to express such relations between objects. The resulting C code is practically the same. For more information on associations refer to section 3.8.
3.6.5 Initial value

The initial value of an attribute can be specified in the general Properties panel of Poseidon. It is copied verbatim into the source code by the code generator.

For regular attributes with an initial value, a corresponding assignment statement is generated in the Init section inside the constructor of the class. Class attributes (i.e. static attributes, see 3.6.6) are initialized accordingly at the point of their definition in the c-file.

3.6.6 Class (Static) Attributes

A regular attribute of a class is object-specific, which means that for each object of the class there is an individual corresponding data slot. These data slots may have different values. On the other hand, a class attribute is class-specific, which means that it has the same value for all objects of the class. Class attributes are also called static attributes.

An attribute may be specified as static by checking the corresponding box either on the general or the C/C++ Properties panel.

For a static attribute, no corresponding member in the structure of the class is generated. Instead, a global variable is generated in the Definitions section of the c-file. The name of that variable is the name of the static attribute prefixed by the class name.

As an example we look at the following class TempControl, which has a regular attribute currentTemp and a static attribute maxTemp initialized to 99:

```
TempControl
- currentTemp : unsigned int
- maxTemp : unsigned int = 99
```

Figure 3-39. A temperature controller with a static attribute.

The class’ structure is defined in the h-file as follows:

```c
/**********************************************************
 * Attributes of class TempControl *
 **********************************************************/
/* definition of attributes */
#define _PE_TempControl_ATTR
        unsigned int currentTemp;

struct _PE_TempControl {
    _PE_TempControl_ATTR
};
```

The Definitions section in the c-file contains the following code:

```c
/*@ <Definitions> @*/
#ifndef IMPLICIT_SELF
#define DEFINESELF
#endif
```
The static attribute `maxTemp` can be accessed in the implementation of the class using the name `TempControl_maxTemp`, e.g.:

```c
if (self_currentTemp > TempControl_maxTemp) {
    MSG_ErrorHandler_Alarm(theErrorHandler, ERR_TEMP);
}
```

### 3.6.7 Constant class attributes

Attributes can be specified as constant by checking the corresponding box on the C/C++ properties panel. Concerning code generation, only the special case of constant static attributes is currently supported. For each constant static attribute, a preprocessor macro with the same name as the attribute is defined in the h-file. The initial value of the attribute is used as the value of the macro.

### 3.6.8 Bit Attributes

Classes often have attributes representing a value that is either true or false (so-called flags). Some boolean data type is normally used for this purpose. However, such attributes typically consume at least one byte of memory. For this reason bit coding techniques are often used in order to save memory.

The special data type `bit` can be used to model status flags in Poseidon. The C code generator automatically packs the bit attributes of a class into a byte array of the matching size and provides corresponding façade message macros.
The following example shows how bit attributes are used. The class Timer has a regular attribute and two bit attributes.

<table>
<thead>
<tr>
<th>Timer</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ticks : unsigned long</td>
</tr>
<tr>
<td>-active : bit = 0</td>
</tr>
<tr>
<td>-cyclic : bit</td>
</tr>
</tbody>
</table>

Figure 3-40. Timer class with bit attributes.

As you can see, bit attributes may also have initial values. The structure generated for the class in the h-file contains a member representing both bit attributes:

```c
/* Attributes of class Timer */
#endif
/* definition of attributes */
#define _PE_Timer_ATTR 
    unsigned long ticks; 
    unsigned char _PE_Timer_BitAttr0;

struct _PE_Timer {
    _PE_Timer_ATTR
};
```

Up to eight bit attributes can be used without larger memory consumption. The following message macros are generated automatically:

```c
#define MSG_Timer_isActive(obj)  
    (((obj)->_PE_Timer_BitAttr0) & (1<<0)) ? 1 : 0)

#define MSG_Timer_setActive(obj, arg)  
    (arg ? 
        (((obj)->_PE_Timer_BitAttr0) |= (1<<0)) : 
        (((obj)->_PE_Timer_BitAttr0) &= ~(1<<0)))

#define MSG_Timer_isCyclic(obj)  
    (((obj)->_PE_Timer_BitAttr0) & (1<<1)) ? 1 : 0)

#define MSG_Timer_setCyclic(obj, arg)  
    (arg ? 
        (((obj)->_PE_Timer_BitAttr0) |= (1<<1)) : 
        (((obj)->_PE_Timer_BitAttr0) &= ~(1<<1)))
```
3.7 Operations

3.7.1 Operation Name

The name of an operation is specified in the general Properties panel of Poseidon. The name should satisfy the ANSI C rules for identifiers because it is used as an identifier in the source code. For convenient use, spaces in the attribute name are replaced by underscore characters.

3.7.2 Visibility

The visibility of an operation can be specified in the general Properties panel of Poseidon. Only the visibility levels public, protected, and private are supported.

By default, the code generator adds the prefixes priv_ and prot_ to the names of private and protected operations, respectively. This naming scheme is also regarded when the message macros are generated. The prefixes may be changed or omitted on the generation settings dialog (see 6.1).

Since protected operations must be visible for derived classes, they must be declared in the h-file of a class. Consequently, they could also be accessed by external classes. The prefix prot_ helps to avoid such errors.

Private methods are generated as static C functions in the c-file of a class. The façade message macros are also generated only in the c-file of a class. Since static functions have file scope, they are not visible externally.

3.7.3 Parameters

Parameters can be added to an operation on the Properties panel of Poseidon. The return type of an operation is defined by the special parameter with the name return.

The passing policy for a parameter can be specified in the C/C++ Properties panel:

![Figure 3-41. Properties panel for parameters (ANSI C).](image)
3.7.4 Binding

The C/C++ Properties panel lets the user determine the binding for an operation.

By default, the option unspecified is selected. In this mode the generator decides how the operation must be bound. If the operation is not redefined in an inheritance relationship, then it is statically bound. This means that it will be called just like a regular C function. If the operation is redefined, the generator makes it a virtual operation and it will be called via the VTable.

You may also decide to specify the binding of an operation explicitly. If you select the option virtual, the operation will be bound dynamically, regardless of whether it is redefined or not. Correspondingly, non-virtual methods are always statically bound.

Abstract methods are a special case of virtual methods. They have no implementation, i.e. no corresponding C function is generated and no user code can be supplied as function body. The VTable of the enclosing class has a corresponding entry that is initialized to NULL. Abstract methods may not be called directly. They are used to define the interface of a base class and must be redefined in derived classes.

Please note that the binding specification for an operation is only relevant for the generation if the operation is neither static nor inline (see below). Static or inline operations always have static binding.

3.7.5 Class (Static) Operations

An operation may be specified as static by checking the corresponding box either on the general or the C/C++ Properties panel. Static operations are also called class operations. Class operations have class scope and are called independently from any object of the class. Consequently, they can access only static attributes.

The following example shows the class TempControl with an additional static operation getMaxTemp:

```
TempControl
- currentTemp : unsigned int
- maxTemp : unsigned int = 99
+ getMaxTemp() : unsigned int
+ getTemp() : unsigned int
```

A regular C function without a self parameter is generated for a static operation:
Just as regular operations, class operations are called via message macros, e.g.:

```c
maxT = MSG_TempControl_getMaxTemp();
```

### 3.7.6 Inline Operations

Operations can be generated inline by choosing the corresponding option on the C/C++ Properties panel. Inline operations are a very important means for efficient support of encapsulation.

The body of an inline operation can be edited only in the Code Preview editor of Poseidon.

For illustration, we add an inline operation `getTemp` to our example class `TempControl`. Assuming that `getTemp` should return the maximum value of the attribute `currentTemp` and the static attribute `maxTemp`, the body of the operation could look like this:

```c
inline unsigned int TempControl_getTemp(obj) {
    /** lock-begin */
    return |self_currentTemp < TempControl_maxTemp ? self_currentTemp : TempControl_maxTemp|;
    /** lock-end */
}
```

![Figure 3-44. Implementation of an inline operation.](image)

As you can see, the body of inline operations can be edited as usual. Only the pseudo keyword `inline` indicates that this operation will be expanded inline when the code is generated.

No C functions are generated for inline operations. Instead, their body is transformed and pasted verbatim into the definition of the corresponding façade message macro, e.g.:

```c
#define MSG_TempControl_getTemp(obj) \  
(obj)->currentTemp<TempControl_maxTemp ? (obj)->currentTemp : TempControl_maxTemp
```

![Figure 3-45. Façade message macro generated for an inline operation.](image)
Due to the generation schema, only very simple operations should be defined inline. Typical applications for inline operations are attribute access operations. Static operations can also be defined inline.

### 3.7.7 Constructors and Destructors

Constructors and destructors are special operations of a class which must be executed when an object of that class is created and destroyed, respectively. The naming was chosen in the style of C++: the constructor of a class must have the same name as the class and the name of the destructor must be the name of the class prefixed by a tilde character.

There is exactly one constructor and one destructor per class. If these operations are not specified in the model, they are generated implicitly.

Constructors may have any number of parameters, destructors must not have parameters.

The following example shows a class \texttt{Buffer} and a derived class \texttt{RingBuffer}.

```plaintext
Buffer
- size : unsigned int
- buf : unsigned char *
+ Buffer(size:unsigned int)
+~ Buffer()
+ add(val:unsigned char)
+ get() : unsigned char

RingBuffer
+ RingBuffer(size:unsigned int)
+ add(val:unsigned char)
+ get() : unsigned char
```

Figure 3-46. Classes with constructors and destructors.

\texttt{Buffer} objects should represent byte buffers of a specific size. The buffer should be allocated dynamically when an object is created and should be deallocated automatically when the object dies.

```plaintext
/*@ <Constructor ID=10-10-1--41-ff0d4b:f54e8b27be:-7ff5> @*/
void _PE_Buffer_Create(void * _self, unsigned int size) {
  DEFINESELF
  /* -> add your code here */
  @*/<Init> @*/
  self->_PE_vMethods = &_PE_Buffer_MT;
  @*/</Init> @*/
  /* -> add your code here */
```
The constructor contains a protected subsection called Init. User code may be added before and after this section.

The derived class `RingBuffer` also defines a constructor with a parameter `size`. In the constructor of a derived class it is important to call the constructor of the base class. This base class constructor call is generated automatically in the user code before the Init section. Of course, when the base class constructor has parameters, the generator cannot guess which values have to be passed as arguments. These values have to be filled in by the user. The base class destructor call is also generated automatically.
When an object is created and destroyed dynamically using the NEW and DELETE macros, the corresponding constructor and the destructor are called automatically. For static objects the INIT and DEINIT macros have to be used in order to explicitly call the constructor and the destructor, respectively.

The binding can be specified for destructors exactly as for regular operations (see 3.7.4).

### 3.8 Associations

Associations are relationships between classes that model links between the objects of these classes. Objects can cooperate using these links, e.g. by calling each other’s methods.

Associations are only included in code generation if they are navigable and if they have role names. The Poseidon ANSI C component generates attributes for associations, i.e. the object links are realized by attributes.

![Association between classes A and B.](image)

In the example, class **A** has a navigable association to class **B**. The opposite association end has the role name **b**. Thus, the code generator will add an attribute named **b** to class **A**. The structure of class **A** looks like this:

```c
#define _PE_A_ATTR  
  B * b;

struct _PE_A {  
  _PE_A_ATTR
};
```

**b** can be used just like a regular attribute, which means that a corresponding façade message macro **self_b** is available in the c-file of the class.
Of course, associations can be bi-directional as well. In this case Poseidon generates attributes for both involved classes, provided that both roles are named and that the association is navigable in both directions. A class may also have self-associations.

The type of the attribute generated for an association depends on containment settings and the multiplicity of the opposite association end.

There are two possible options for the containment: by value and by pointer. The containment for an association can be specified on the C/C++ Properties dialog for the opposite association end. In the example (Figure 3-47) the opposite association end is the end with the role name b.

![Figure 3-48. Properties panel for association ends (ANSI C).](image)

By default, the containment for associations and aggregations is set to by pointer. For compositions by value is initially chosen. Of course, these default settings may be changed.

If the containment is set to by pointer, an object pointer type is generated for the attribute. An object value type is generated if by value is chosen.

Concerning multiplicities, a difference is made between the limited and unlimited case. For limited multiplicities, such as 5 or [7..10], arrays are generated. Unlimited multiplicities, such as [0..*], are mapped to the special container class GrArry. GrArry implements a dynamic (growable) array that can contain objects of any type. It is delivered with Poseidon for UML Embedded Edition as model and source code and can therefore be adapted if necessary.

The following table summarizes the default generation scheme for associations:

<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Containment by pointer</th>
<th>Containment by value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>B *b</td>
<td>B b</td>
</tr>
<tr>
<td>0..1</td>
<td>B *b</td>
<td>B *b</td>
</tr>
<tr>
<td>42 (any limited number)</td>
<td>B *b[42]</td>
<td>B b[42]</td>
</tr>
<tr>
<td>0..10 or 5..10 (any limited range)</td>
<td>B *b[10]</td>
<td>B *b[10]</td>
</tr>
<tr>
<td>0..* or 1..* (any unlimited range)</td>
<td>GrArry *b</td>
<td>GrArry *b</td>
</tr>
</tbody>
</table>

The default attribute type generated for an association can be overridden by an arbitrary type on the C/C++ Properties panel.

Additionally, the static modifier can be selected. In this case, the generated attribute becomes a class attribute.
3.9 Dependencies

Usually, objects collaborate in order to fulfill their tasks. Objects reference other objects and call their operations. The classes of collaboration objects depend on each other. From the modeling point of view, there are implicit and explicit dependencies.

Implicit dependencies result from class relationships (inheritance and association) and from using other classes as types (attribute and parameter types).

 Explicit dependencies must be used if a class uses another class in its implementation, e.g. in the body of an operation.

![Diagram showing dependencies between two classes: Client and Supplier](image)

**Figure 3-49. Dependency between two classes.**

The class **Client** might need to use a **Supplier** object in the operation **op**:

```c
void Client_op(void * _self) {
    DEFINESELF
    /* -> add your code here */
    int val;
    Supplier *sup;

    NEW_Supplier(sup);
    val = MSG_Supplier_compute(sup, 42);
    DELETE_Supplier(sup);

    /* ... */
}
```

**Figure 3-50. Using an object of another class as a local variable.**

For code generation, dependencies are evaluated in order to generate `#include` directives.

3.10 Singleton Classes

The singleton design pattern often occurs when designing software for embedded systems. 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 Poseidon ANSI C generator provides a special object layout for singleton classes. It results in even less overhead and allows very efficient access.

Singleton classes are specified by the stereotype `<<Singleton>>`. The stereotype for a class can be set on the general Properties panel of Poseidon.

The following example shows a singleton class **MessageQueue**, which associates the class **Message**.
Figure 3-51. The singleton class MessageQueue.

The instance object of a singleton class is accessed using the automatically generated operations `getInstance` and `destroyInstance`. These operations are responsible for calling the constructor and the destructor, respectively. The `NEW`, `DESTROY`, `INIT`, and `DEINIT` macros do not exist for singleton classes. The constructor and the destructor of a singleton class are generated as private operations. As with usual objects, operations and attributes are accessed using the façade macros.

```c
MessageQueue *theMsgQ = MessageQueue_getInstance();
Message *msg;
/* ... */
MSG_MessageQueue_enqueue(theMsgQ, msg);
/* ... */
msg = MSG_MessageQueue_dequeue(theMsgQ);
```

Figure 3-52. Using the singleton class MessageQueue.

No structure is generated for a singleton class. Instead, all attributes are defined as regular variables in the c-file of the class. Since there is no actual object, the operations of a singleton class do not need a `self` parameter. This means a lower need for stack memory. The use of the façade macros for attribute and operation access hides the singleton generation scheme.

There are several restrictions for singleton classes:

- Singleton classes may not have inheritance relationships to other classes.
- Singleton classes may not have virtual or abstract methods. All methods 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.
4 Code Generation for Classes in ANSI C++

The code generation scheme for C++ is much more straightforward than for ANSI C. C++ is an object-oriented programming language. Thus, concepts like classes, inheritance, objects, polymorphism, and dynamic binding are directly available in the target language and need not be imitated by the code generator.

This chapter describes how classes and their properties and relationships are mapped to ANSI C++ source code by Poseidon for UML Embedded Edition. The generation for statechart diagrams is explained in chapter 5.

4.1 Classes

The Poseidon ANSI C++ code generator creates two files for each UML class that is selected for code generation:

- **SegmentDisplay.h**
- **SegmentDisplay.cpp**

The technique used for the synchronization between the model and the code is basically the same as in the ANSI C++ component. Each generated file contains several protected sections. These sections are surrounded by meta comments containing tags indicating the purpose of the section, e.g.:

```cpp
/*@ <Definitions> */
...
/*@ </Definitions> */
```

The code in a protected section may not be modified, since it is rewritten on every generation step. Any modification in a section will be lost after the next generation. The only exceptions are operation sections, which contain the implementation of the operations of a class. The body of an operation can be implemented either in Poseidon's Code Preview panel or using any external editor. Operation bodies will be preserved by the generator. The same holds for user code that is added outside the protected sections. For further details about synchronization options see section 6.5.

The particular sections of the two generated files are explained using the class **SegmentDisplay** as an example.
4.1.1 Sections of the Declaration File

The first file that is generated for a class is the declaration or header file. It mainly contains the declaration of the C++ class.

```cpp
/*@ <FileComment ID=1053416803100> */
/********************************************************************************/
* Class          : SegmentDisplay
* File           : SegmentDisplay.h
* Generated with : Poseidon for UML EmbE 1.6
* Last generation: Tue May 20 09:46:43 CEST 2003
*******************************************************************************/
/*@ </FileComment ID=1053416803100> */

/*@ <IncludeGuard> */
#ifndef SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H
#define SegmentDisplay_10_10_1__41_1d41116_f507f54250__7fec_H
/*@ </IncludeGuard> */

/*@ <Include> */
#include "common.h"
/*@ </Include> */

/*@ <Definitions> */
class SegmentDisplay
{
public:
    // -----------------------------------------------------------------------------
    // public constructors
    // -----------------------------------------------------------------------------

    /**
     * Constructor.
     */
    SegmentDisplay(u16 * port);

    // -----------------------------------------------------------------------------
    // public operations
    // -----------------------------------------------------------------------------

    /**
     * Clears this segment display.
     */
```
void clear();

/**
 * Shows a digit (number between 0 and 9) on this segment display.
 */
void showDigit(u8 digit);

private:
    // private attributes
    u16 * portAddr;
};

Figure 4-2. The declaration file of a class in ANSI C++.

**FileComment section**
The FileComment section contains a comment describing the purpose of the file. The appearance and the contents of that section can be configured in the code generation dialog (see 6.2).

**IncludeGuard section**
The IncludeGuard section makes sure that the file is included only once per compilation unit. The name of the include guard is determined by the class name and a unique identifier.

**Include section**
The Include section contains the dependencies that the class has to other classes. These dependencies are expressed either as #include directives or as class forward declarations.

**Definitions section**
The definitions section contains the declaration of the class.
4.1.2 Sections of the Implementation File

The implementation file of a class contains the implementations of the operations of the class:

```cpp
/*@ <FileComment ID=1053416803100> @*/
/******************************************************************************
 * Class          : SegmentDisplay
 * File           : SegmentDisplay.cpp
 * Generated with : Poseidon for UML EmbE 1.6
 * Last generation: Tue May 20 09:46:43 CEST 2003
*******************************************************************************/
/*@ </FileComment ID=1053416803100> @*/

/*@ <Include> @*/
#include "SegmentDisplay.h"
/*@ </Include> @*/

/*@ <Definitions> @*/
/*@ </Definitions> @*/

/*@ <Operation ID=10-10-1--41-1d41116:f507f54250:-7fd9> @*/
/** *
 * Constructor.
 */
SegmentDisplay::SegmentDisplay(u16 * port)
{
    portAddr = port;
}
/*@ </Operation ID=10-10-1--41-1d41116:f507f54250:-7fd9> @*/

/*@ <Operation ID=10-10-1--41-1d41116:f507f54250:-7fe0> @*/
/** *
 * Clears this segment display.
 */
void SegmentDisplay::clear()
{
```

IncludeGuardEnd section

This section terminates the include guard.
```cpp
portAddr |= 0x00FF;
}
/*@ </Operation ID=10-10-1--41-1d41116:f507f54250:-7fe0> @*/

/*@ <Operation ID=10-10-1--41-1d41116:f507f54250:-7fdd> @*/
/**
 * Shows a digit (number between 0 and 9) on this segment display.
 */
void SegmentDisplay::showDigit(u8 digit)
{
    portAddr |= (0xFF00 & digit);
}
/*@ <Operation ID=10-10-1--41-1d41116:f507f54250:-7fdd> @*/
```

Figure 4-3. The implementation file of a class in ANSI C++.

**FileComment section**

The FileComment section contains a comment describing the purpose of the file. The appearance and the contents of that section can be configured in the code generation dialog (see 6.2).

**Include section**

The Include section contains the `#include` directives that are needed for the implementation of a class. At least, the declaration file of the class is included here. `#include` directives for the declaration files of dependant classes may follow.

**Definitions section**

The definitions section contains the definitions and initializations of the static members of the class. In our example, the section is empty because the class has no static attributes.

**Operation sections**

The Operation sections are the most important parts of the implementation of a class. Initially, a section with an empty function code frame is generated for each operation of the class. The bodies of the operations must be implemented by the user either directly in Poseidon or in any other editor. Arbitrary user code may be added inside the operation. The user code will be preserved in future generation steps.

### 4.1.3 Class Properties

The target language for a class can be specified on the C/C++ Properties panel for a class. The default language can be set in the Generation Settings dialog (see section 6.1).
The following settings can be made:

- **Target language.** ANSI C or ANSI C++ can be selected as the target language for the class.

- **Additional includes.** `#include` directives that are not derived from the model can be specified here. The text entered in this area is copied verbatim into the include section of the declaration file of the class.

- **Target file names.** By default, the names of the target files are derived from the class name. If necessary, the base names of the files can be overwritten here. The file extensions can be specified globally in the Generation Settings dialog (see 6.1). The target file path can be specified globally in the Code Generation dialog and for any enclosing package in the C/C++ Properties panel for packages (see 4.2).

For classes with an associated state machine, there is an additional option that determines whether Poseidon should generate code for that state machine.

### 4.2 Packages

UML Packages are useful to structure the classes in logical units. For the target language C++ they can also be used to define namespaces.

- **Namespace.** If checked, this package constitutes a namespace. All classes that are contained in the package are defined in a namespace with the name of that package. Packages and namespace definitions may be nested (see example below).

- **Target path.** A target path for all enclosed classes can be specified. You can enter an absolute path here. Alternatively, the path variables `$ROOT` and `$PARENT` can be used at the beginning of the path. `$ROOT` refers to the global output folder specified in the code generation dialog. `$PARENT` refers to the output path specified for the parent
package of this package. The following default settings are applied when a new package is created:

- $\$ROOT$/<package name> for top-level packages
- $\$PARENT$/<package name> for nested packages

The following example shows how packages are mapped to namespaces.

![Diagram of nested packages](image)

**Figure 4-6. A nested package diagram.**

Provided that the “Is namespace” property is checked for both packages Foo and Bar, the following class definition is generated:

```cpp
/*@ <Definitions> */
namespace Foo
{
    namespace Bar
    {
        class A
        {
            
        } // namespace Foo
    } // namespace Bar
/*@ </Definitions> */
```

**Figure 4-7. C++ class defined in nested namespaces.**
4.3 Features of a Class

Attributes and operations form the features of a class. This section outlines the common properties.

Name
The name of a feature is specified in the general Properties panel of Poseidon. It should satisfy the C++ rules for identifiers, because it is used as an identifier in the source code. Spaces in the feature name are replaced automatically by underscore characters.

Type
A type must be specified for attributes and operation parameters. By default, the type `int` is selected. Other types may be selected from the type drop-down list on the Properties panel. If a new non-class data type is needed, it can be created on the Properties panel for the model (topmost node in the navigation pane).

Visibility
The visibility of a feature is specified on the Properties panel of Poseidon. Only the visibility levels `public`, `protected` and `private` are supported by the code generator.

4.4 Attributes

The following settings are available in the C/C++ Properties dialog for attributes.

4.4.1 Multiplicity
The multiplicity of an attribute can be specified in the general Properties panel of Poseidon. Multiplicities can be used in order to generate arrays. Only limited numbers and ranges are allowed. The upper bound of a multiplicity is taken as the dimension of the resulting array. Unlimited ranges such as `[0..*]` are not considered by the code generator. Associations should be used for this purpose.

4.4.2 Initial value
The initial value of an attribute can be specified in the general Properties panel of Poseidon. It is copied verbatim into the source code by the code generator.

If a regular attribute has an initial value, a corresponding initialization is added to the member initialization list of every constructor of the class.

Class attributes (see section 4.4.4) are initialized in the Definitions section of the implementation file of the class.

4.4.3 Containment
The C/C++ Properties panel provides an additional option for attributes, called containment.
This containment property determines the way how an attribute is contained in an object of its class. For the target language C++, possible values for the containment of an attribute are by value, by pointer, or by reference. By value is selected by default. In this case the type of the resulting structure member is taken verbatim from the model. If the containment mode by pointer is chosen, an additional pointer declaration is added to the specified type. Accordingly, if by reference is selected, the resulting attribute type will be a reference type.

### 4.4.4 Attribute Modifiers

The following modifiers affect the code generation for an attribute.

**const**

The resulting attribute is declared as a constant attribute. It is recommended to specify an initial value for constant attributes.

Example:

```cpp
const int a1;
```

**static**

The attribute is a class (static) attribute. Class attributes have the same value for all objects of a class.

Example:

```cpp
static int a2;
```

**mutable**

The resulting attribute is declared with the `mutable` keyword.

Example:

```cpp
mutable int a3;
```

**volatile**

The resulting attribute is declared with the `volatile` keyword.

Example:

```cpp
volatile int a4;
```
4.5 Operations

The following code generation modifiers can be selected on the C/C++ Properties dialog for operations:

**const**

The operation is generated as a constant operation.

Example:

```cpp
void op1() const;
```

**static**

The operation is generated as a class (static) operation.

Example:

```cpp
static void op2();
```

**inline**

The operation is generated inline, i.e. the implementation of the operation is contained in the declaration of the class. The body of an inline operation must be edited in the Code Preview panel in Poseidon.

4.5.2 Operation Binding

There are several options for the binding of an operation:

**abstract**

The operation is generated as a pure virtual operation.

Example:

```cpp
virtual void op1() = 0;
```

**virtual**
The operation is generated as a virtual operation.
Example:

```cpp
virtual void op2();
```

**non-virtual**

The operation is generated as a regular operation.
Example:

```cpp
void op3();
```

**friend**

The operation is generated as a friend function.
Example:

```cpp
friend void op4();
```

**unspecified**

This is the default setting for the binding of an operation. The operation is generated as a regular non-virtual operation.

### 4.5.3 Parameters

Parameters can be added to an operation in the general Properties panel of Poseidon. The return type of an operation is defined by the special parameter with the name `return`.

The passing policy for a parameter can be specified in the C/C++ Properties panel:

![Properties panel for parameters (ANSI C++)](image)

Figure 4-10. Properties panel for parameters (ANSI C++).

If the `const` modifier is checked, the corresponding parameter will become a constant parameter.

### 4.5.4 Constructors and Destructors

Constructors are operations that have the same name as the class. An operation with the name of the class prefixed by a tilde character (~) is the destructor of the class.

A constructor may have any number of parameters. If the class has attributes with initial values, a member initialization list is generated for every constructor. Additional initializers may be specified in the C/C++ Properties panel.
Constructors and destructors can be defined inline. The `explicit` modifier for constructors adds the keyword `explicit` to the constructor definition.

### 4.6 Associations

The C++ generation scheme for associations is pretty much the same as for C.

Associations are only included in code generation if they are navigable and if they have role names. The Poseidon ANSI C++ component generates attributes for associations, i.e. the object links are realized by attributes.

![Association between classes A and B.](image)

The following class definition is generated for class A:

```cpp
/*@ <Definitions> */
class A
{
public:
    // public associations
    B * b;
};
/*@ </Definitions> */
```

As already described in section 3.8, the type of the generated attribute depends on the containment and the multiplicity of the opposite association end.

In C++ there are three possible containment types: by value, by pointer, and by reference. By default, the containment property is set to by pointer for associations and aggregations. For compositions, by value is chosen initially. The containment type can be changed in the C/C++ Properties panel of the opposite association end:
Figure 4-13. Properties panel for association ends (ANSI C++).

The available modifiers allow the generation of static, constant, and mutable attributes for associations.
5 Code Generation for State Machines

In UML, class diagrams describe the static structure of software systems. Statechart diagrams are a very powerful means for modeling the behavior of event-driven systems.

Every class may have an associated state machine, which may be spread over several statechart diagrams. Poseidn for UML Embedded Edition can generate highly efficient executable code for a state machine. It automatically adds attributes and operations to the class that are used to control the state machine: the state machine can be initialized and events can be sent to it in order to trigger state transitions.

The underlying generation algorithm is basically the same for both target languages C and C++. Only the access operations are slightly different. Therefore, this chapter explains the code generation for state machines for both target languages.

Although general statecharts may be modeled with Poseidon for UML, the C/C++ code generator supports only a subset of all constructs possible in the UML (5.7). The generated code is very efficient and compact.

5.1 Using State Machines

We consider a simple fictitious class Player, which has operations to switch the device on and off and to start and stop playing:

```
+switchOn()
+switchOff()
+play()
+stop()
```

![Figure 5-1. A simple class Player.](image)

We create the following statechart diagram for the class Player:

![Figure 5-2. Statechart diagram for class Player.](image)
The state machine has three simple states **Off**, **Stop**, and **Playing** as well as a composite state **On**. Initially, the state machine is in the state **Off**. When the **POWER_ON** event is received, the composite state **On** is entered, which finally causes the state machine to change to state **Stop**. The event **POWER_OFF** lets the state machine change to the state **Off**, regardless of the currently active sub-state of **On**.

After the statechart is created, an additional option appears in the C/C++ Properties panel, which lets you decide whether target code should be generated for the state machine. If this option is checked and code has been generated the first time – either to the file system or the Code Preview panel – the Player class is expanded by an attribute and two operations:

<table>
<thead>
<tr>
<th>Player</th>
</tr>
</thead>
<tbody>
<tr>
<td>- PE_state_of_Player : unsigned char</td>
</tr>
<tr>
<td>+ switchOn()</td>
</tr>
<tr>
<td>+ switchOff()</td>
</tr>
<tr>
<td>+ play()</td>
</tr>
<tr>
<td>+ stop()</td>
</tr>
<tr>
<td>+ init()</td>
</tr>
<tr>
<td>+ onSignal(in sigId:unsigned char) : unsigned char</td>
</tr>
</tbody>
</table>

Figure 5-3. The player class with additional state machine features.

The generated attribute will contain the current state of the object when the state machine is running.

The task of the **init** operation is to initialize the state machine and to perform the initial transition. It is important that this operation is executed prior to using the state machine. Often, it is a good idea to call **init** in the constructor of the class.

If the state machine is activated, events can be sent to it using the operation **onSignal**. By default, this operation has one parameter representing the event to be sent. Additional parameters can be added if necessary. **onSignal** returns a boolean value that indicates whether the event has been consumed.

The names of the automatically generated features can be changed if necessary.

The following code shows how to use the state machine of a **Player** object in ANSI C:

```c
/* define a Player object */
Player *p;
NEW_Player(p);

/* initialize the state machine of the object */
MSG_Player_init(p);

/* send some events */
MSG_Player_onSignal(p, POWER_ON);
MSG_Player_onSignal(p, PLAY);
MSG_Player_onSignal(p, POWER_OFF);
```

Figure 5-4. Using a state machine in ANSI C.
As you can see, the event names can be used as specified in the model. If state machine tracing is activated, the following output can be observed:

```
state_machine_top: enter state [initial]
state_machine_top: enter state Off
Receiving signal POWER_ON
state_machine_top: leave state Off
state_machine_top: enter state On
On: enter state [initial]
On: enter state On_Stop
Receiving signal PLAY
On: leave state On_Stop
On: enter state On_Playing
Receiving signal POWER_OFF
On: leave state On_Playing
state_machine_top: leave state On
state_machine_top: enter state Off
```

Figure 5-5. Tracing output from the state machine.

The code for using a state machine in C++ is very similar:

```
// define a Player object
Player p;

// initialize the state machine of the object
p.init();

// send some events
p.onSignal(POWER_ON);
p.onSignal(PLAY);
p.onSignal(POWER_OFF);
```

Figure 5-6. Using a state machine in ANSI C++.

### 5.2 Actions

Of course, state machines are useful for the implementation of reactive systems only when actions are executed with state changes.

Actions can be associated with states and transitions. Simple and composite states may have entry and exit actions. These actions are executed when the corresponding state is entered and left, respectively. An action that is associated with a transition is called the effect of the transition. It is executed if the transition is performed.

We add some actions to our Player example:
5.3 Guards and decisions

A guard is a condition that may be specified for a transition. If a guarded transition is potentially triggered by an event, it is only performed, if the guard expression is true.

A guard condition may be an arbitrary C/C++ expression that can be evaluated as a boolean value. Again, any access to an attribute value or any operation call via `self` is transformed to the corresponding target language construct. The special guard expression else may be

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]

\[ e \text{ [else]} \]

\[ e \text{ [self.x == 42]} \]

\[ e \text{ [self.x == 17]} \]
used for one transition. This transition is taken if the guard conditions of all other transitions
with the same trigger event evaluate to false.
Choice points are another means for modeling branches and decisions in statechart
diagrams:

Figure 5-9. Statechart with choice point.

5.4 History States
A composite state can have a history state. It is a short hand notation representing the most
recent active sub-state of the composite state.

Figure 5-10. Player statechart with history state.

If a Player object receives an ERROR event in any sub-state of the composite state On, the
action handleError() is executed. After that, the state machine switches back to the last
active state. If the composite state was not entered before for any reason, the default
transition of the history state is taken. (This is not possible in the example.)
The C/C++ code generator also supports deep history states (H*), which remember the last
active state even for nested sub-states.
5.5 Final States and Completion Transitions

A final state indicates that a state machine (or sub-state machine) has finished its work. Final states cannot be left directly. Completion transitions can be used to leave composite states that have reached a final sub-state.

![Statechart with final sub-state and completion transition.](image)

Graphically, a completion transition is indicated by an anonymous transition originating from the outside edge of a composite state. The completion transition is taken automatically when a final sub-state is reached.

5.6 Tracing

For debugging purposes, the generator can be configured to instrument the state machine code with tracing commands (see 6.3). In order to enable the tracing, a global function with the following signature must be implemented and linked with the target code:

```c
void _PE__trace(char *s)
```

This function is called with an appropriate message everytime an event is received and a state transition is performed.

5.7 Rules and Restrictions

The following rules must be regarded for correct code generation:

- The top-level diagram must have an initial state.
- Every composite state with a default entry must have an initial sub-state. A default entry is an incoming transition that ends on the outside edge of a composite state.
- An initial state must have exactly one outgoing transition without a trigger event.
- All transitions that leave a state and have the same trigger event must be guarded by disjunctive guard conditions. It is recommended that one of the guard conditions is an `else` condition.
- A choice point must have at least one outgoing transition.
• Transitions leaving a choice point may not have trigger events.
• All transitions leaving a choice point must be guarded by disjunctive guard conditions. It is recommended that one guard condition is an **else** condition.
• History sub-states should have a default transition without a trigger event.
• Completion transitions are allowed only for composite states.
• A composite state may have at most one completion transition.
• A composite state can have at most one completion transition. Moreover, guard conditions are not supported for completion transitions.

The following UML statechart constructs are *not* supported by the Poseidon C/C++ code generation:

• concurrent states
• fork and join states
• synchronizations states
• do-activities.
6 Generation Options

General code generation settings can be made on the Generation Settings dialog. This dialog can be opened by pressing the “Settings...” button on the Generation dialog. Please note that “Embedded” must be selected as the kind of generation.

![Generation dialog with the button for opening the Generation Settings dialog.](image)

The individual options are described in the following sections.
6.1 General Settings

![General Settings dialog](image)

**Indentation**

These settings affect the indentation style of the generated code. The code can either be indented with spaces or tabs. If indentation with spaces is selected, the indent size (depth of indentation) can be specified. If tabs are used, the number of spaces that equal one tab can be entered.

**Prefix for operations**

These settings are relevant only for classes with ANSI C as target language. An individual prefix string can be specified for public, protected, or private operations. The code generator adds these prefixes to the corresponding operation names.

**File extensions**

Here, the extensions that should be used for the generated files can be defined for each target language.
Default language
This option determines which target language should be selected by default when a new class is created.

6.2 Fileheader Settings

In this dialog the contents of the FileComment protected sections can be specified. A FileComment section is generated at the beginning of each file. An individual text can be used for declaration files and for implementation files, respectively. The following variables may be used in the text:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Expanded to</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ClassName$</td>
<td>the name of the class</td>
</tr>
<tr>
<td>$ClassDoc$</td>
<td>the class documentation in the model</td>
</tr>
<tr>
<td>$File$</td>
<td>the name of the generated file</td>
</tr>
</tbody>
</table>

Figure 6-3. File header settings dialog.
6.3 Tracing Settings

Figure 6-4. Tracing settings dialog.

If the option in this dialog is checked, the code that is generated for state machines is instrumented with special tracing commands (see 5).
6.4 Memory Settings

Figure 6-5. Memory settings dialog.

Size of integral types
The size of the integral data types of the target system can be specified here. At the moment, this information is only used for determining the suitable data type for the state machine related attributes.

Memory management functions
These settings can be used to customize the functions names that the generator should use for the dynamic allocation and deallocation of objects. Additionally, the corresponding include file can be specified.
6.5 Synchronization Settings

![Synchronization settings dialog]

**Figure 6-6. Synchronizations settings dialog.**

**Synchronization**

Since the user code can alternatively be edited in the Code Preview panel of Poseidon or in an external Editor, a synchronization mechanism is necessary in order to keep the model and the corresponding files consistent. The code that is the target of the synchronization is the code between the protected sections and the method bodies. Apart from operations, the code in the protected sections is not synchronized.

The goal of the synchronization is to keep the complete user code in the model. Thus, the target code for a model can be regenerated completely at any time.

The generator remembers the time stamp of the last generation of a class. Before the next generation this time stamp is compared with the modification time stamps of the existing files. The user code of all files that are newer is parsed back to the model prior to generation. Additionally, the synchronization can be triggered every time the Code Preview for a class is activated.
**Behavior on conflict**

A synchronization conflict occurs if the user code of a class is changed in the Code Preview and in a file simultaneously. In this case it is impossible for the generator to decide which version is valid. Instead, the behavior is determined by the configuration in this dialog.