

# UML - Getting Started

## IBM Rational Rhapsody

To a running UML-Model in just a few steps.

**WILLERT.**

Welcome to our UML - Getting Started. In this tutorial, everything revolves around the first practical steps in the world of Embedded UML.

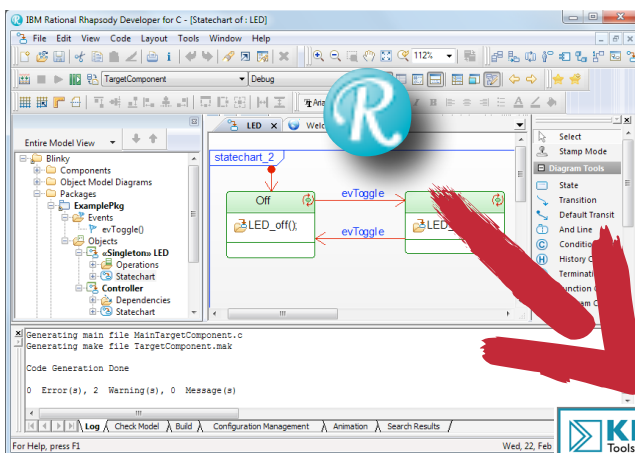
With the help of a tiny object-oriented UML-Blinky we will make a small journey through the UML and enlighten the functions of the required software.

We provide the necessary programs as free evaluation version for you either as download or as DVD.

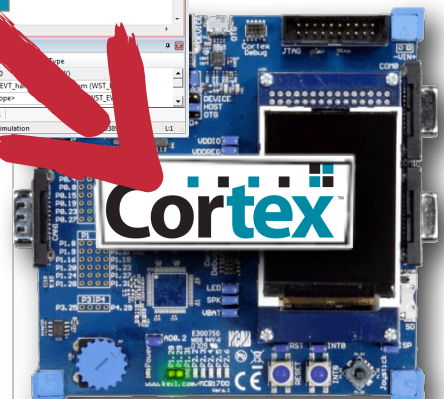
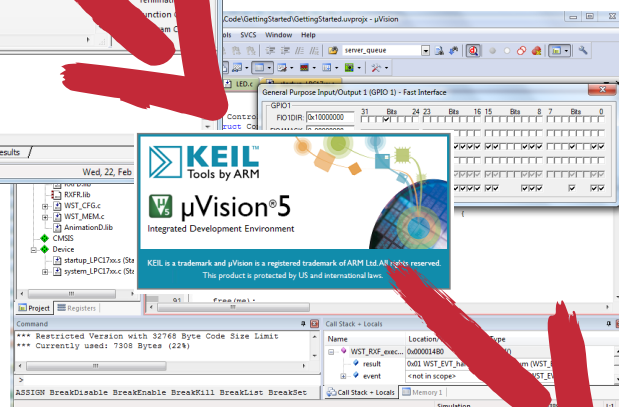
This will enable you to use the software without restrictions during a 30 day evaluation period.

As target environment we use a Keil MCB1760 with an LPC1768 (CortexM3). Unfortunately only a few people will probably be in the possession of this evaluation board, so we will test our final program at the end of this tutorial with help of the simulator built-in in Keil's µVision IDE. So, apart from a PC, you do not need additional hardware. If you want, however, you can purchase an MCB1760 board directly from Willert Software Tools GmbH.

For the toolchain installation and implementation of this tutorial you should reserve about 60 minutes and two cups of ☕. Have fun and success with it...



- IBM Rational Rhapsody for C 8.1.5
- Keil µVision MDK ARM v5.21a
- Keil MCB1760 CortexM3



**Code Generation: ANSI C**

# Software & Installation

We will start with the installation of our toolchain. All the software you need is located on our Willert DemoDVD. In addition to the necessary tools you will also find an installation guide that will quickly help you through the installation process. Therefore we will not describe the installation process in detail in this tutorial. We rather briefly discuss the possibilities of the individual tools.

You can order, or download, this free demo-DVD on our website.

- Download ->

[www.willert.de/uml-getting-started-en/](http://www.willert.de/uml-getting-started-en/)

## Content of the DemoDVD

### IBM® Rational® Rhapsody®



With Rhapsody, we create UML-Diagrams and generate ANSI 'C'-code from them. The "Blinky" example will be modeled completely with the help of this environment.

### Keil MDK ARM



We need the Keil µVision IDE to flash our executable model to the target or to let it run in the built-in simulator. On top of that the IDE offers additional debugging possibilities for our model.

### Realtime eXecution Framework



The Willert Embedded UML RXF™ combines Rhapsody with the Keil µVision IDE. It optimizes the code-generation especially for a resource friendly use in an embedded real-time environment.

- **Rhapsody v8.1.5**  
UML Development Environment
- **Willert RXF**  
Realtime eXecution Framework  
(for the ARM Cortex M3)
- **Keil MDK ARM 5.21a**  
Compile / Debug / Flash

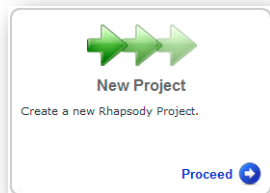
#### System Requirements (minimum)

- Windows 7 or Windows 10
- 2 GB free disk space
- 2 GB memory

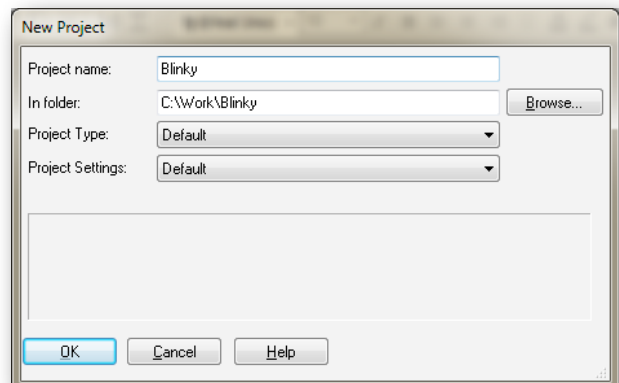
# Start Rhapsody / Create a Project

Now we start "IBM Rational Rhapsody for C 8.1.5" and model an object-oriented Blinky for a CortexM3. Of course we could save a few pages, by omitting classes and objects in this simple Blinky. However since we will later mostly work with classes, we'd rather just put a solid foundation.

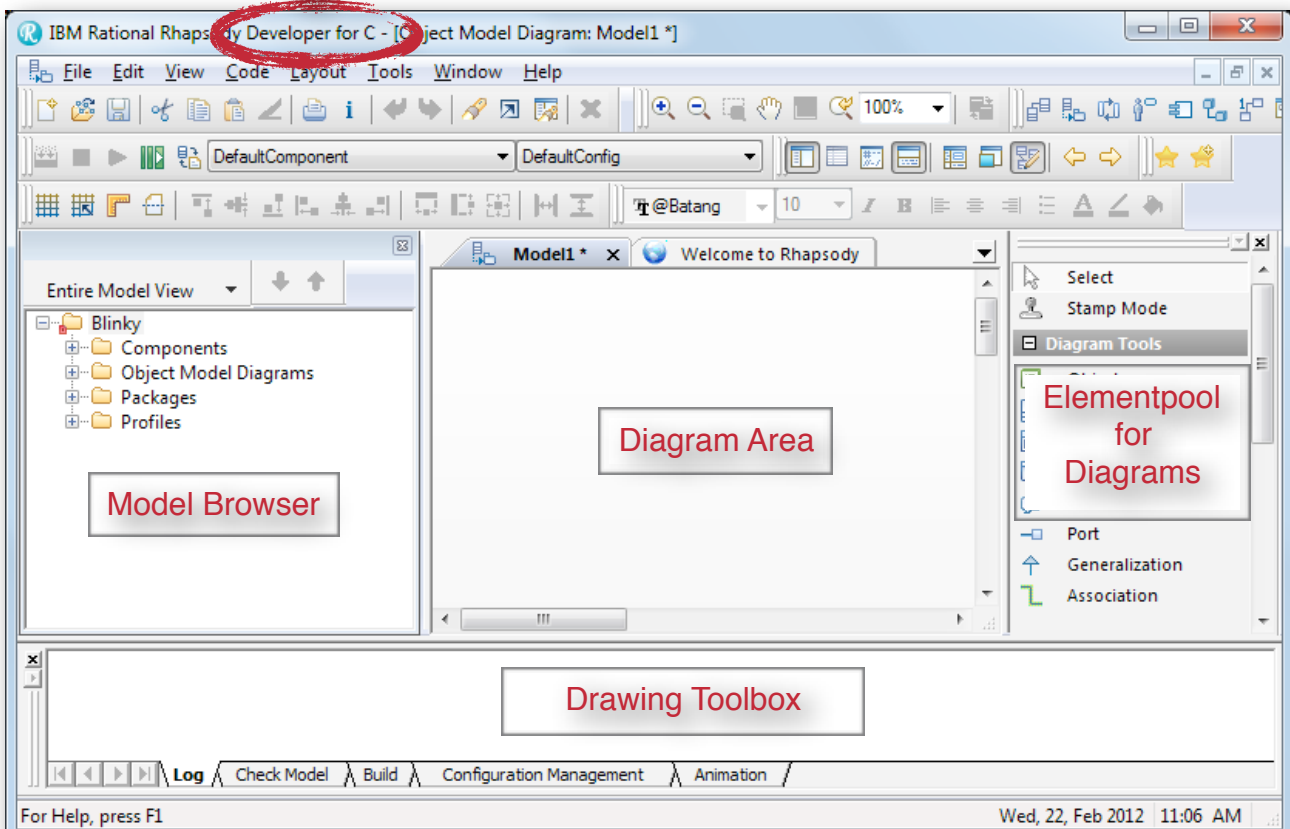
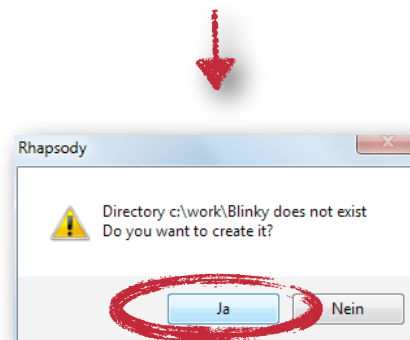
In the Welcome - Screen of the Rhapsody window we click on the "Next" symbol, or choose "File/New"



We determine a appropriate name for our project and location than continue with OK.



If you have installed Rhapsody with more than one language, please be sure to start Rhapsody in C. So go to the windows start menu and navigate to Programs / IBM Rational / IBM Rational Tools / IBM Rational Rhapsody 8.1.5 / Rational Rhapsody Developer Edition / Rational Rhapsody Developer for C.



# Adding Profiles

With profiles we have the possibility to easily add settings to a Rhapsody project. We provide profiles to adapt the code generation of Rhapsody to work with our Framework.

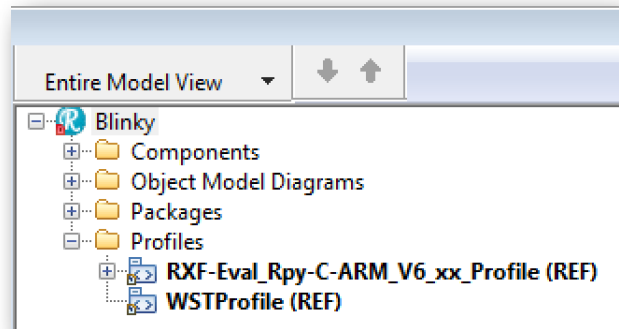
We add such a profile to our project. For this situation, Willert Software Tools already created a profile that we can find at the following location:

```
<Rhapsody User Path>\  
Share\Profiles\WST_RXF_V6\  
WSTprofile.sbs
```

```
<Rhapsody User Path>\  
Share\Profiles\WST_RXF_V6\  
RXF-Eval_Rpy-C-ARM_V6_xx_Profile.sbs
```

- click on: File -> Add Profile to Model
- navigate to the profile containing folder
- select "WSTProfile.sbs"
- click: Open

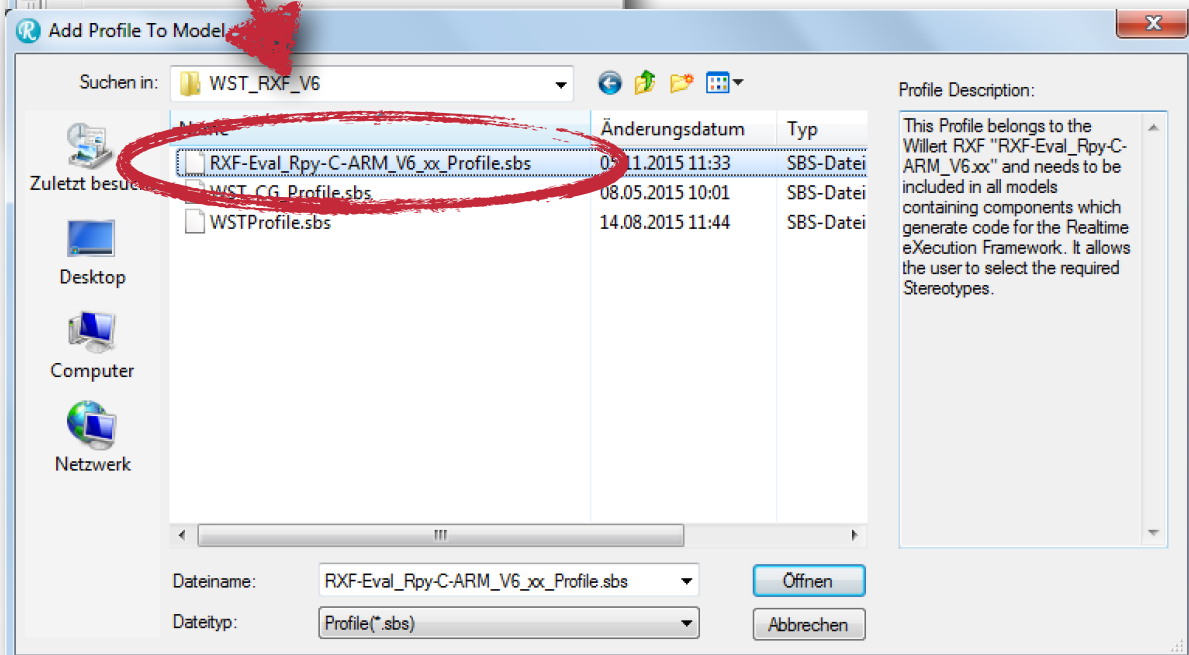
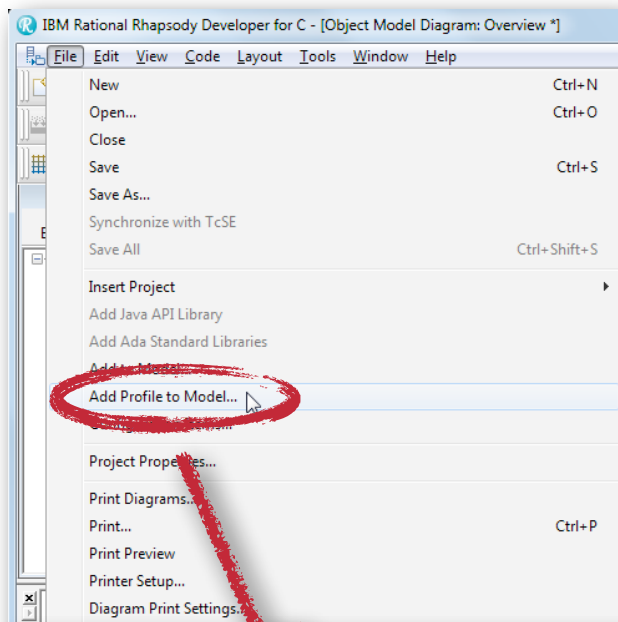
Do this again with the profile named "RXF-Eval\_Rpy-C-ARM\_V6\_xx\_Profile.sbs".



Under Profiles, our WSTprofile (REF) now appears. (REF) means, that this file is not copied to our project directory but that it is just referenced.

The profile WSTProfile.sbs is an optional profile and is used to beautify our diagrams.

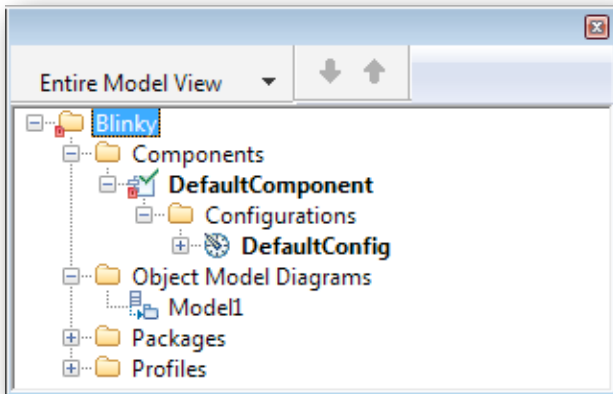
It will, amongst other things, take care that arrows in diagrams are straight instead of curved.



# Model browser / Project Settings

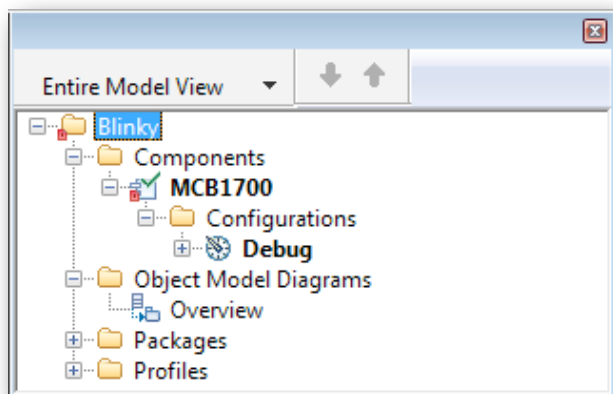
We take a look at the model browser. Similar to the Windows Explorer, you can navigate using the (+/-) symbols through the model. All elements of our UML model are accessible through the browser.

We first start with renaming a couple of items in the model browser. The easiest way to do that is clicking two times on the element to be renamed.



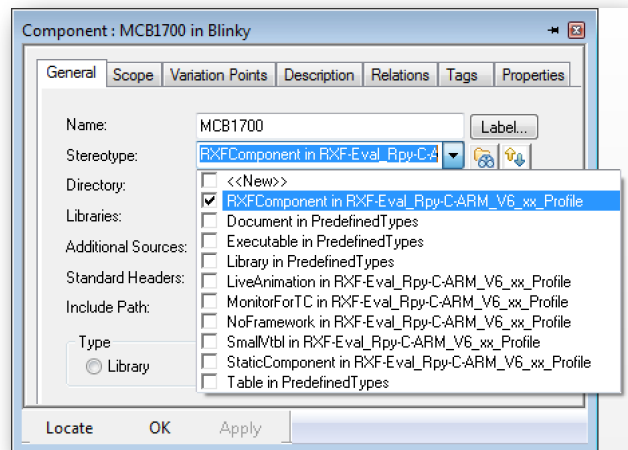
DefaultComponent -> MCB1700  
DefaultConfig -> Debug  
Model1 -> Overview

After this our model looks as follows:



Now we double-click on the MCB1700 component we have just renamed and select the following stereotype in the pop-up window:

RXFCOMPONENT in  
RXF-Eval\_Rpy-C-ARM\_V6\_xx\_Profile

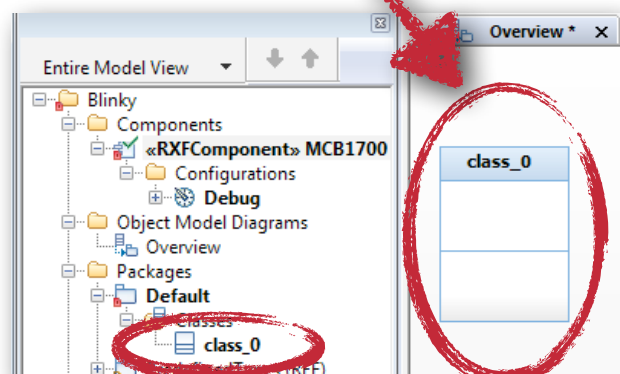
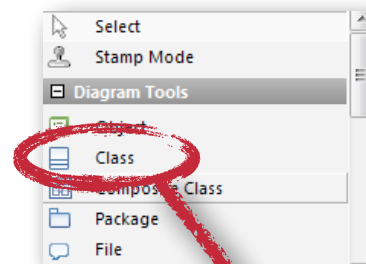


We open the freshly renamed OMD (Object Model Diagram) "Overview" by a double-click.

This will open the Overview OMD in our diagram area. We want to draw a class in our diagram and therefore we select the class symbol from our drawing toolbox. After that we click somewhere on the grey area in the diagram and we see our newly created class "class\_0".

Exactly this class can also be found in the model browser under:

Packages / Default / Classes / class\_0





# Active Code View

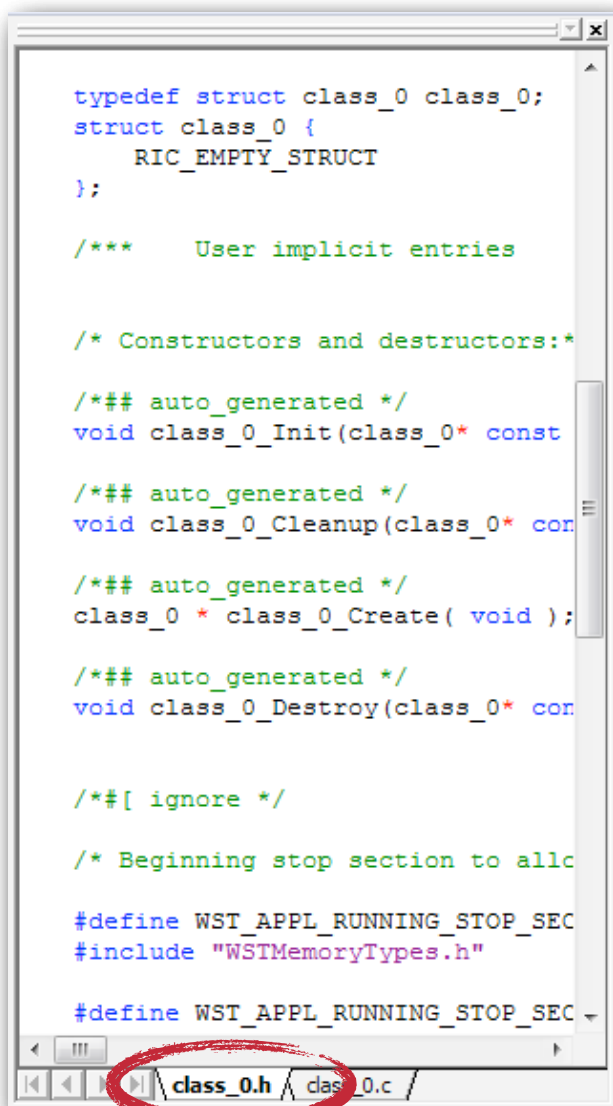
Now let's take a look at the 'C'-code that Rhapsody generates from our model elements. Wouldn't it be nice when we could see what would come out of Rhapsody before we generate our code.

With Active Code View we have the possibility to do just that.

Therefor we click on the Active Code View Button.



Select the "class\_0" UML element from your model browser.



Rhapsody will generate a \*.c and \*.h file for every class.

We can see our just created class in 'C'-code, which at the moment is represented by an empty struct. A click on the "class\_0.c" tab gives us further insights.

Right away we notice four functions that are essential for Object Orientation.

## class\_0\_Init

Our Initializer is needed to initialize an Object after it is created. This function can be compared with a constructor in C++.

## class\_0\_Cleanup

Is there to allow the object to clean up before it is destroyed. Is the equivalent of the destructor in C++.

## class\_0\_Create

Creates the object. reserving memory and call the initializer. Is equal to what "new" is in C++.

## class\_0\_Destroy

Destroys the object, freeing the memory like "delete" in C++.

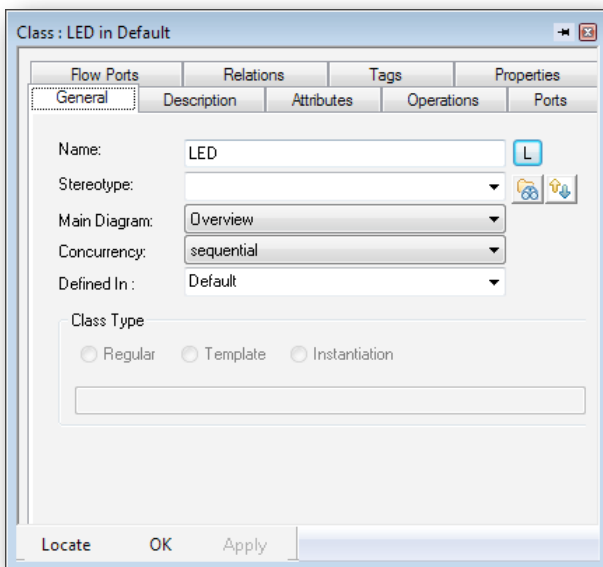
When there is no knowledge present in the area of object oriented programming. Using the UML, in conjunction with Active Code View, can be a wonderful guide to understand the relations in OOP. Many questions can be solved by just looking at the generated code. When you have a large graphical resolution you should lock the Active Code View window in your Rhapsody view.

With a single click on a model element, Active Code View automatically brings us to the right position in the 'C'-code.

## Methods

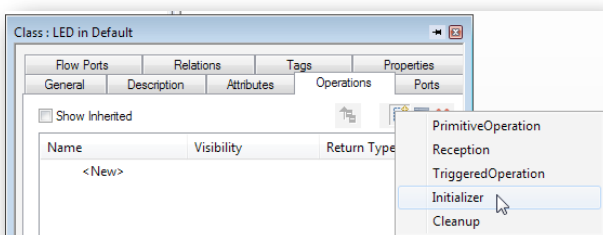
Each element in our model has its own "features window". In this window, we can edit all properties of the element. We open the features window by double-clicking on the element. It does not matter at what point we are doing this in Rhapsody. Double-clicking on the chart class\_0 has the same effect as double-clicking on class\_0 in the browser.

Now we open the features of the class "class\_0" and rename our class to "LED" in the General Tab.

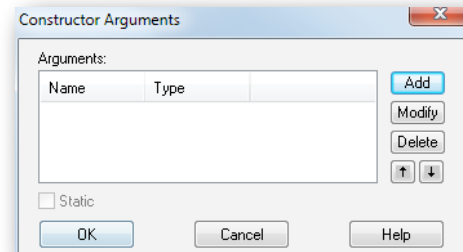


Since we have already opened the features window of our LED class, we will quickly insert two operations and an initializer.

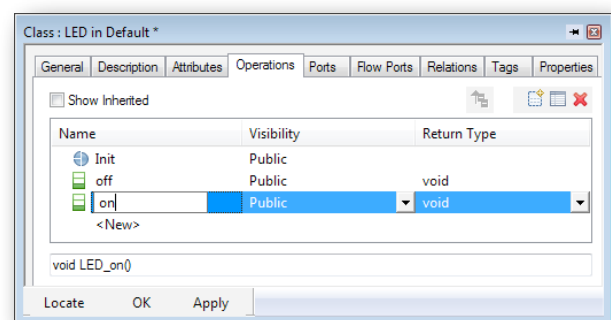
In the Operations-Tab we click on the "New" symbol  and create an initializer.



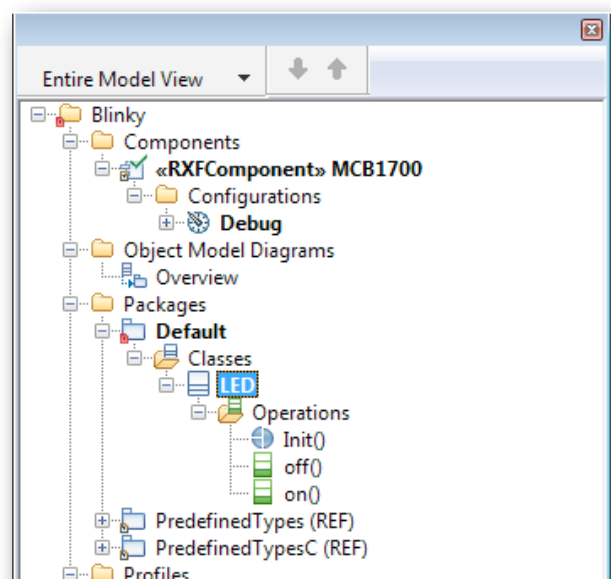
In the next window Rhapsody asks us what arguments the initializer should get. We will postpone that and simply click OK for the time being.



In the same way we add two Primitive-Operations to the LED class and call them "off" and "on".



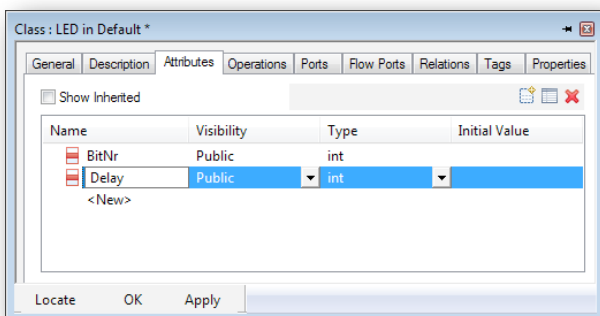
Our class now has an initializer and two operations. These are also displayed in our model browser.



# Attributes

Next, we put two attributes in our class. The first one we will name “BitNr”, this is where we store the number of the Port-Pin to which the LED is connected. The second one will be called “Delay” and contains the delay time we will use for the on and off blinking of the LED.

To create the attributes, we open the features of the LED class in the Attributes tab and create two new attributes called “BitNr” and “Delay”.



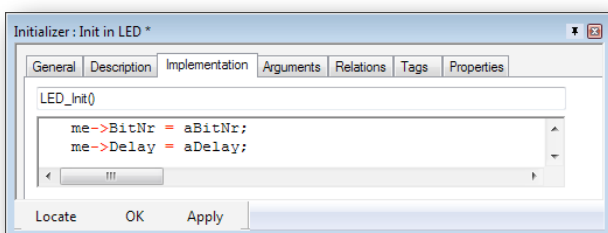
Now our LED class has three empty operations ( Init / off / on ) and two attributes ( BitNr / Delay ).

In order to get both attributes initialized at create time we will add the following ‘C’-code to the Implementation-Tab of the initializer. Now, open the features window from the initializer.

```
me->BitNr = aBitNr;  
me->Delay = aDelay;
```

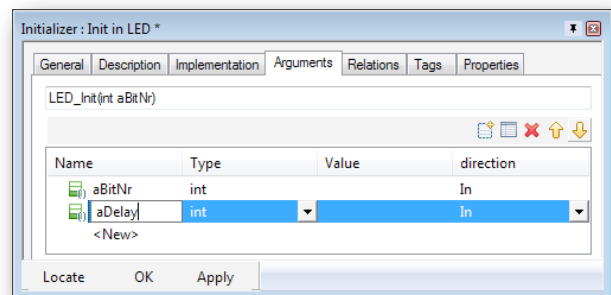
What happened here?

If we later instantiate an object of the “LED” class, each object should have his own “BitNr” and his own “Delay”. This assignment is shown in the first two lines in the picture below.



With that we almost completed our initializer. Just the two arguments that we already used in the implementation are still missing.

So we open the Features of the initializer again and switch to the Argument-Tab. There we create, using the “New” Symbol, the arguments “aBitNr” and “aDelay”.



A look behind the scenes, shows us that Rhapsody has embedded our variables in the LED structure.

```
typedef struct LED LED;  
struct LED {  
    int BitNr;        /*## attribute BitNr */  
    int Delay;        /*## attribute Delay */  
};
```

Using the arrow operator (->) in ANSI C, we can now directly access the elements of this structure.

Like our initializer for example ....

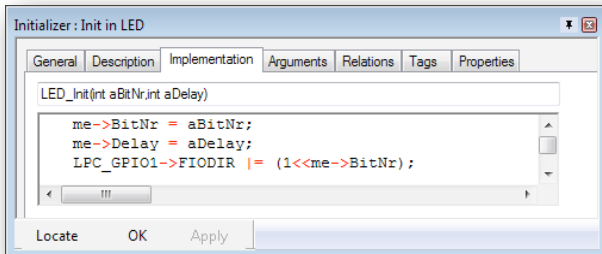
```
/*## operation Init(int,int) */  
void LED_Init(LED* const me, int aBitNr,  
int aDelay) {  
    /*[ operation Init(int,int) */  
    me->BitNr = aBitNr;  
    me->Delay = aDelay;  
    /*] */  
}
```

The “me” Pointer always points to the object itself (It is represented by a structure here).



## Hardwarespecific C-Code

In order to define one of the eight LED's on the MCB1760 as output, we need to set the appropriate bit in the FIODIR register of the CortexM3. We do this with another code line in the Implementation tab of our initializer.

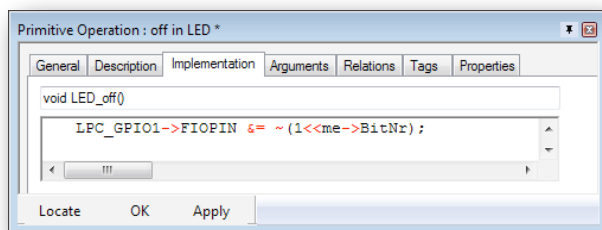


```
LPC_GPIO1->FIODIR |= (1<<me->BitNr);
```

Now our initializer is complete and we can deal with both on- and off-operations.

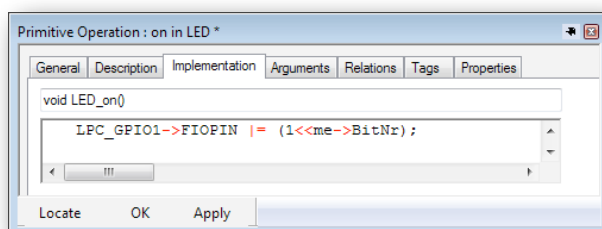
These two operations should be responsible to set the Port-Pin, which we defined as output in our initializer, to low or high respectively.

Let's start with the content of the implementation tab of the "off"-operation.



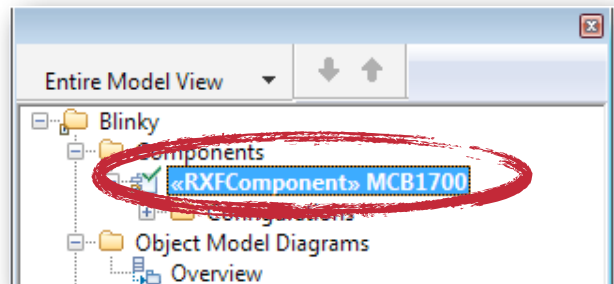
```
LPC_GPIO1->FIOPIN &= ~(1<<me->BitNr);
```

...and after that the "on"- operation

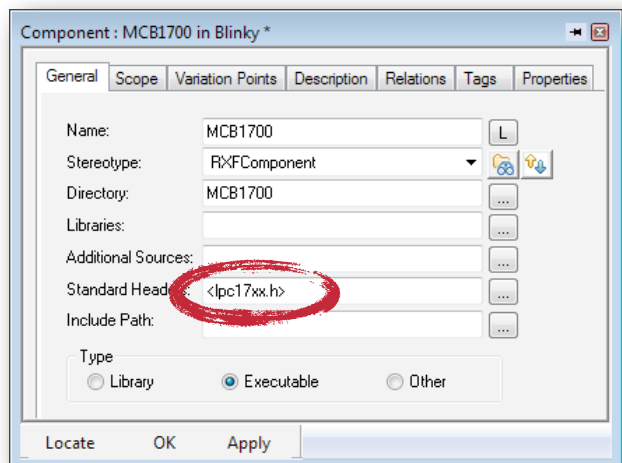


```
LPC_GPIO1->FIOPIN |= (1<<me->BitNr);
```

We now should give our model a link to the header file that describes our specific hardware so that our compiler knows what to do with the hardware specific 'C'-code of the "Init"-, "off"- and "on"-methods. As often, Rhapsody knows a lot of different ways to achieve this goal. The fast and easy way here is to use the features of our "MCB1700"- Component.



In the General-Tab we enter under "Standard-Headers" the name of the header file "<lpc17xx.h>" that we want to include. Please do not forget the <>!



After the next code generation, our LED.h contains the <lpc17xx.h> include.

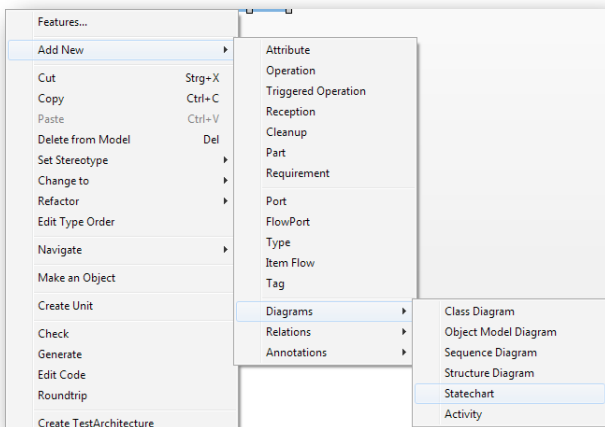
```
/*## auto_generated */  
#include <lpc17xx.h>
```

# Statecharts

Our initializer will be automatically called at the initialization of the LED. But who will now call both our “off”- and “on”- operations?

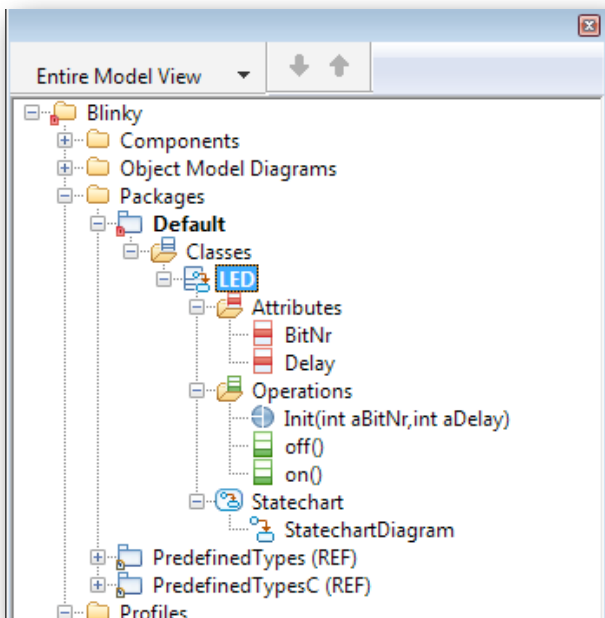
For that we use one of the most commonly used UML diagrams. The statechart (state diagram).

In the model browser, we right click on the LED class, and wind our way through the menus.

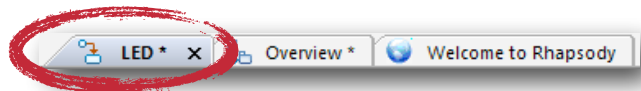


Add New / Diagrams / Statechart

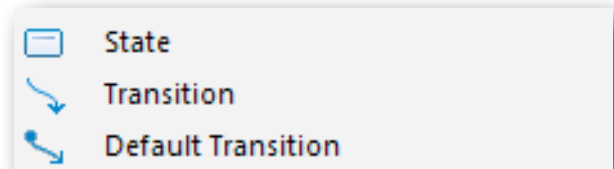
After that our project looks like this:



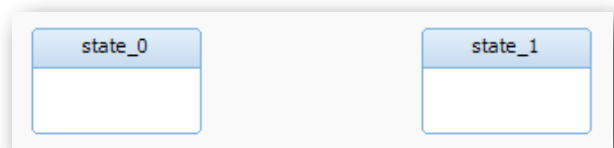
Above our diagram area we see a tab bar in which we now see, next to our Object Model Diagram and the Welcome Screen our newly created Statechart.



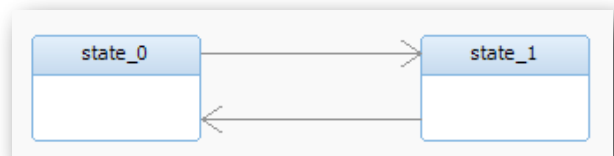
Just for a test, click on the tab for our “Overview” diagram (OMD) and watch the drawing toolbox at the same time. As we can see this changes with the selected diagram type. Now we go back and continue with the statechart.



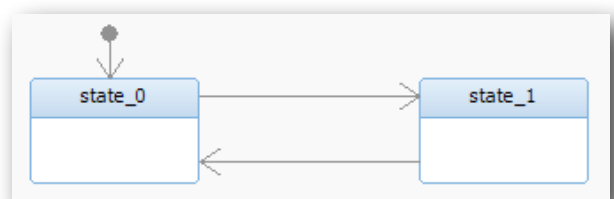
For the statechart of the LED-class we need only three different elements. In the drawing toolbox we click on “State” and then on the grey diagram area. We will repeat that for another state. The result should look as follows.



Next we draw an arrow (Transition) from “state\_0” to “state\_1” and an other one in the opposite direction.



Now the only thing missing in our statechart is the entry point. We draw a default transition from any point to “state\_0”.



# Statecharts

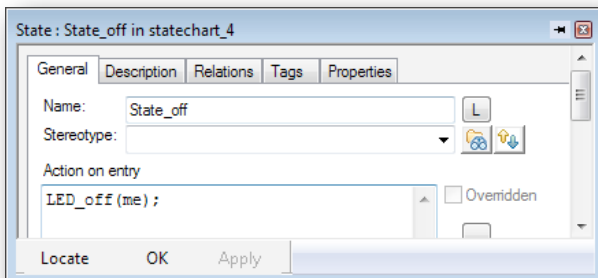
If we want to draw the diagrams very precise then this toolbar will support us in that:



If you select multiple objects in a diagram while keeping the Shift-Key pressed, the symbols in this toolbar will become active. Now you can align the selected objects to the edge of the diagram or to each other. The last selected object is used as pivot point!

Now we open, with a double-click on “state\_0” the features of that state. In the “General”-Tab we rename “state\_0” in “State\_off” and enter the call of the “off” operation under “Action on entry”.

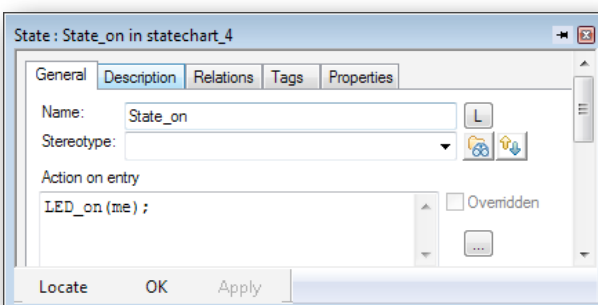
```
LED_off(me);
```



...and now for “state\_1”

Rename in “State\_on” and enter the “on” operation call to the “Action on entry” field.

```
LED_on(me);
```



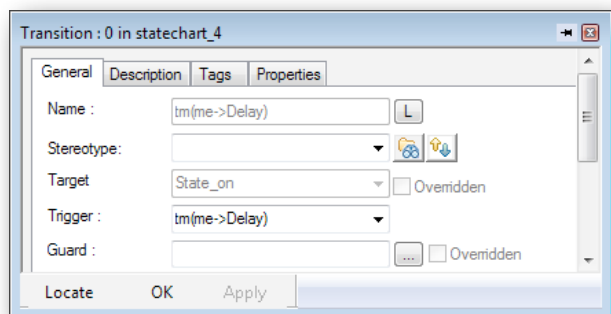
In the current situation our statechart would, together with all the CortexM3 offers, toggle between both states.

The Transitions between the states would be executed immediately. To change that we open the features of one of the transitions with a double-click.

Under Trigger we enter:

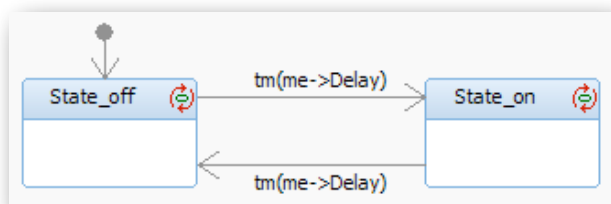
```
tm(me->Delay)
```

in both transitions.



The `tm(me->Delay)` uses the system tick of the CortexM3, to delay the transition for the time we specified. (Later we will initialize the attribute with a value).

Our finished statechart now looks as follows:



Every state that has an action has this symbol:

When we click on this symbol, the statements entered in “Action on entry/exit” are shown in the state icon.

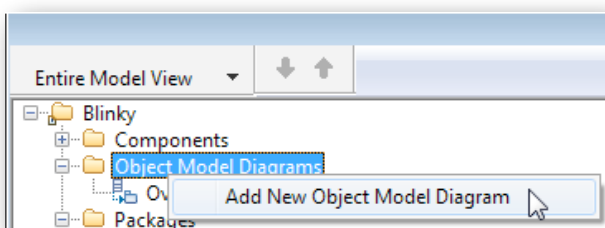
# Instances of a Class

The “LED” class is now complete.

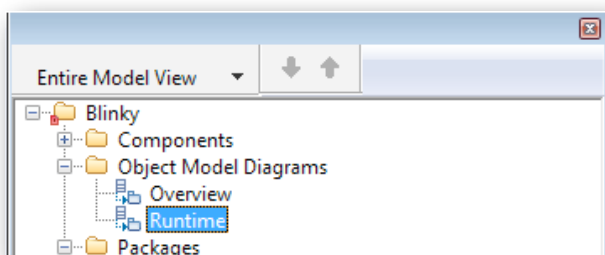
To create an object of our “LED” class at run-time we have to instantiate the class. In Rhapsody there are multiple alternatives to achieve this. The simplest possibility:

First we need a new Object-Model-Diagram. We could use the same diagram but diagrams come for free so why not use another one?

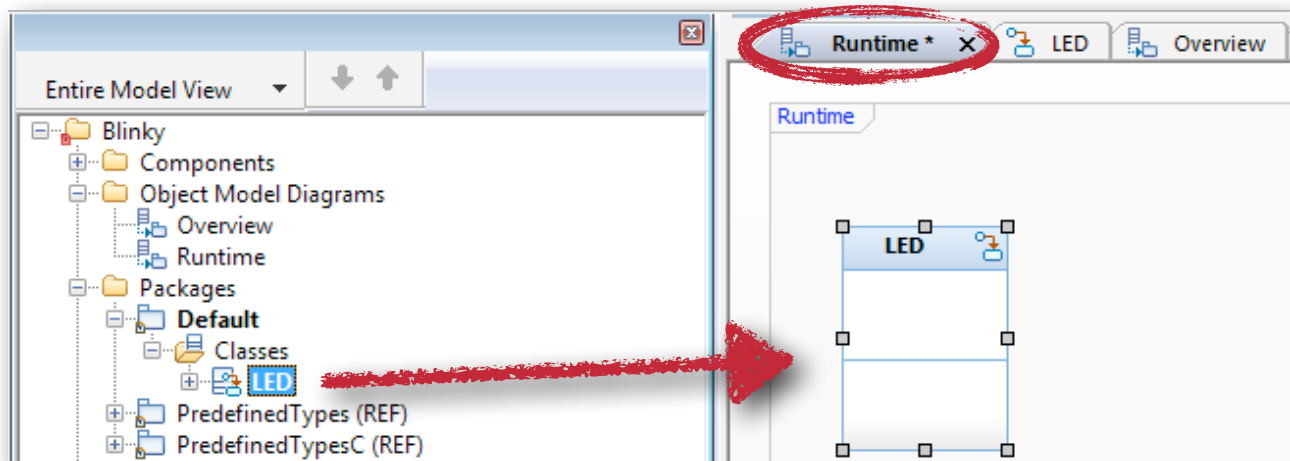
Right click on “Object Model Diagrams“



...and call it Runtime.



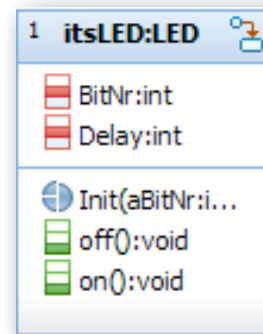
Now drag the “LED” class from our model browser and drop it on the “Runtime” OMD.



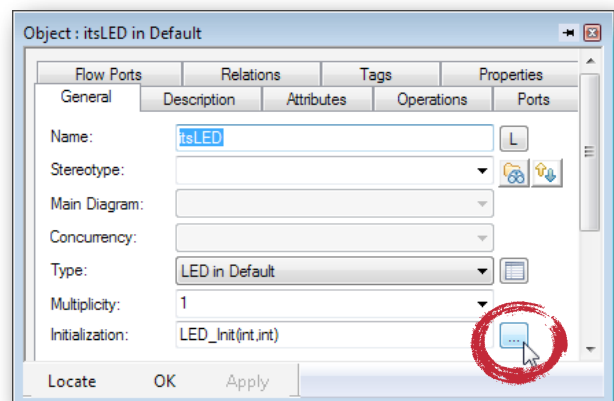
In the “Runtime” diagram we right click on the “LED” class and select (from the context menu) the following item:

Make an Object

From our class “LED” an object will be created at run-time. The object has the name “itsLED“.

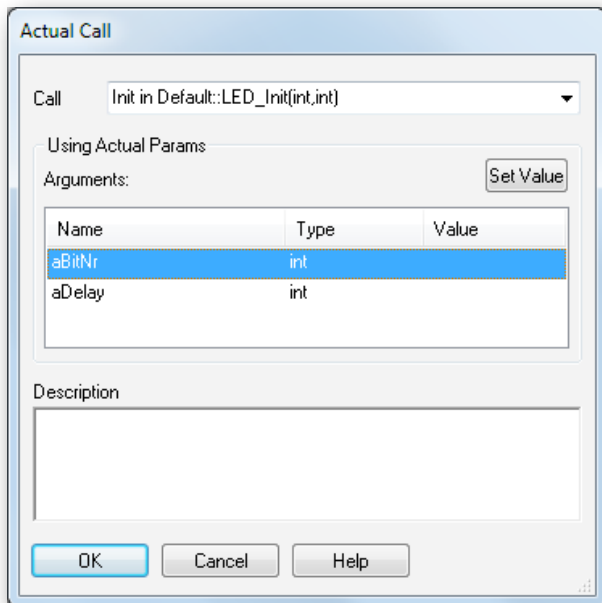


The only thing this object is missing is a suitable “BitNr” and a value for the “Delay” in our statechart. For this we open the features of the object “itsLED” and click the “General” tab under Initialization on the Extend button.

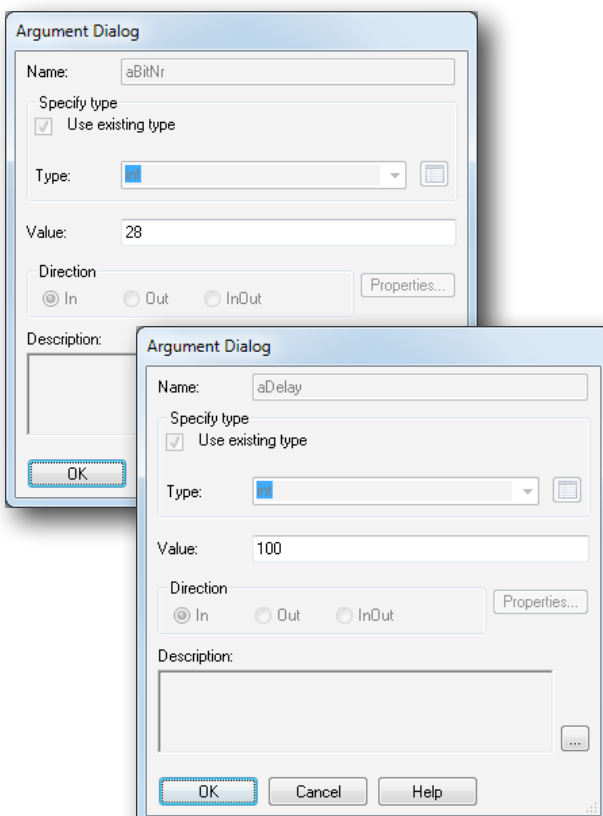


## Instances of a Class

In the next window, we select the variable "aBitNr" and we click on "SetValue".



To address the LED P1.28 on the Keil MCB1760, we have to set or clear the 28th bit of the FIOPIN / FIODIR Register. So we enter the value "28" in the Argument Dialog under "Value".



Repeat the same now with the "aDelay" argument. As value we take "100" (milliseconds).

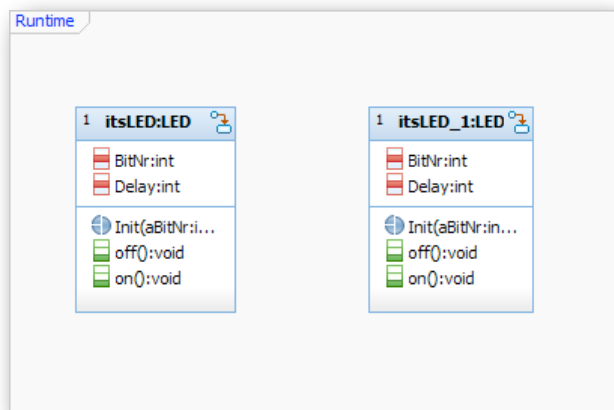
To fully exploit the advantages of object orientation, we will immediately add a second instance of our class.

We drag the "LED" class again from the browser to the "Runtime" diagram, next to the other instance. Right click the class and select...

Make an Object

from the context menu.

Thereafter select the General tab. Under Initialization click the Extend button again and enter a "29" for the "aBitNr" argument and a "50" for the "aDelay" argument.

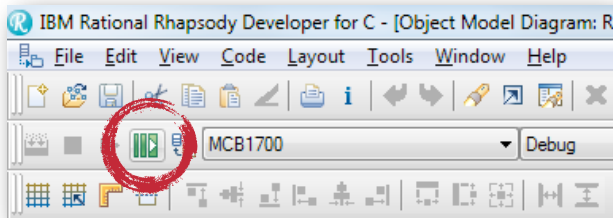


That's it!

We have just instantiated two objects from the same class in a very simple way. Both objects can be distinguished using name or the attributes values.

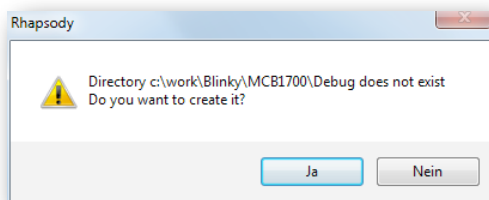
# Generate / Make / RUN

If we have made no mistakes, the model should now be able to run. To test this, we click in Rhapsody on the GMR-Button (Generate / Make / Run). It will automatically go through all three steps in succession.



In the first step Rhapsody generates ANSI 'C'-code from our diagrams and the other model components. The result of the "Generate", after you confirm the next question, can be found in:

C:\work\Blinky\MCB1700\Debug



"Generate" should complete with the following message in the Debug-/Output-window:

All Checks Terminated Successfully

```
Checker Done
0 Error(s), 0 Warning(s)

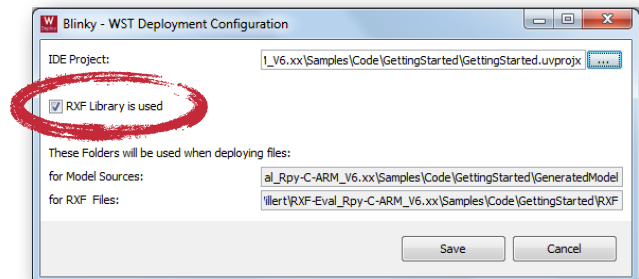
Code generated to directory: C:\work\Blinky\MCB1700\Debug
Generating file Default.h
Generating file LED.h
Generating file Default.c
Generating file LED.c
Generating main file MainMCB1700.h
Generating main file MainMCB1700.c
Generating make file MCB1700.mak

Code Generation Done

0 Error(s), 0 Warning(s), 0 Message(s)
```

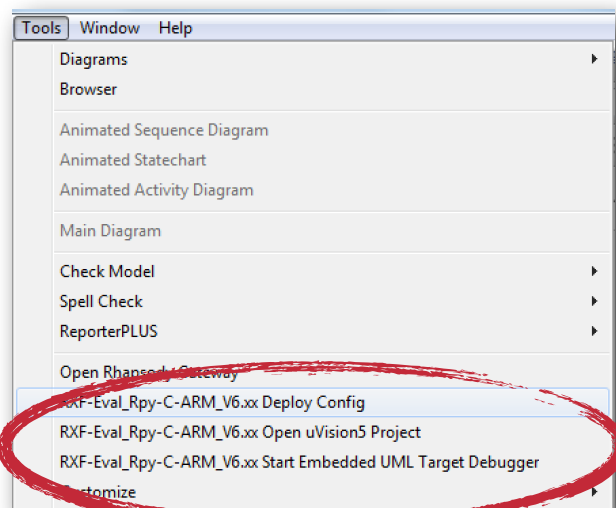
In the next step all generated files will be automatically deployed into a µVision project. First time the GMR-Button is

pressed the deployer config will pop up and it will ask for a destination project. Chose the file „GettingStarted.uvprojx“ in the Folder „<product installation path>/samples/GettingStarted/“



Make sure to select the Option RXF Library is used.

The deployer configuration dialog can be opened any time clicking the dependent



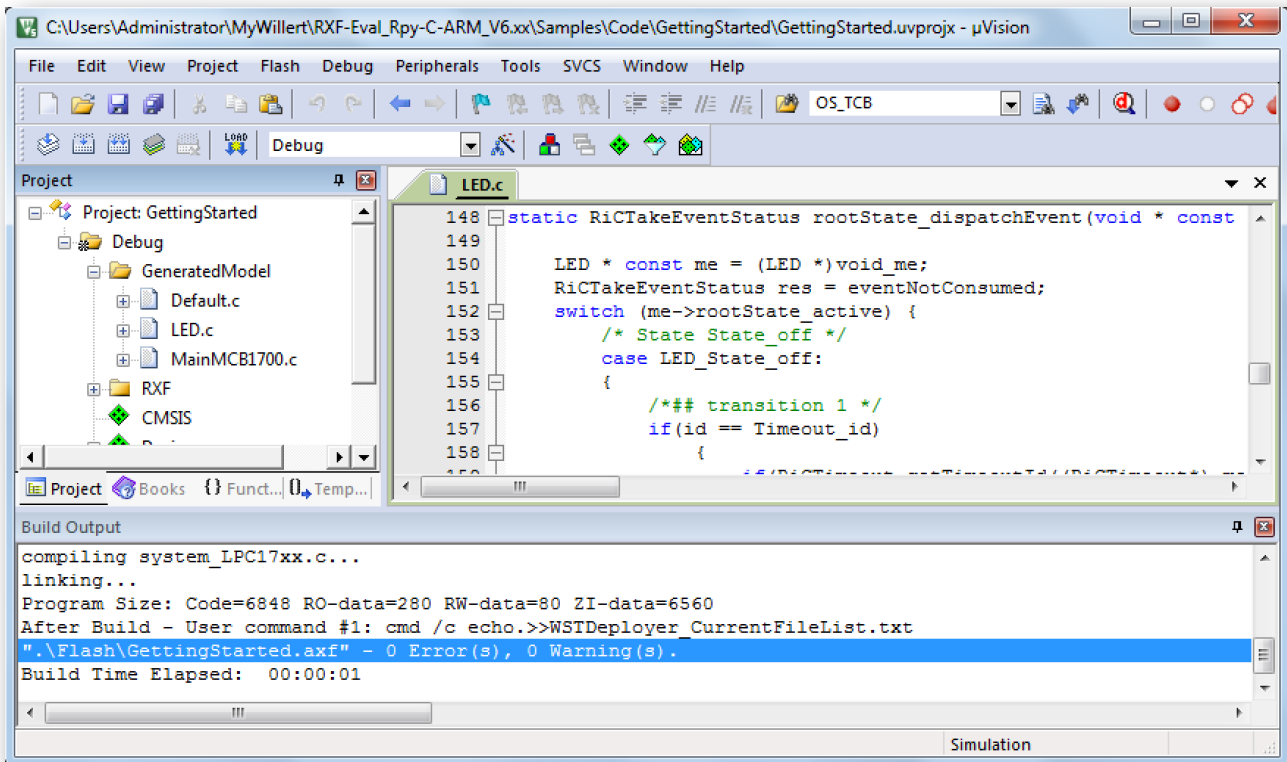
Tip: When errors occur during code generation, a double click on the error message takes you to the place of the error in your Rhapsody model.


entrance in the menubar „Tools“. Now its time to start µVision to test the behavior of our model. µVision is used to compile and run the software.

It can be started manually by double clicking the „\*.uvprojx“ file or by selecting the dependent entry in the menubar „tools“.

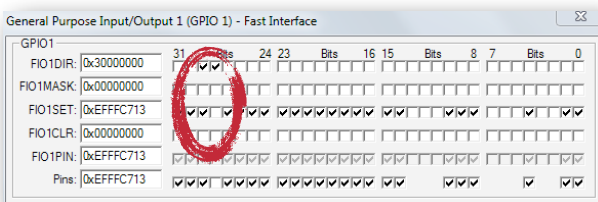



# Keil µVision Development Environment



Of course, now we want to see if our LED's will flash in rhythm. We must, however, change with this  button in the debug mode of the Keil IDE.

When we are in debug mode, an IO-window automatically opens through which we can monitor the state of the

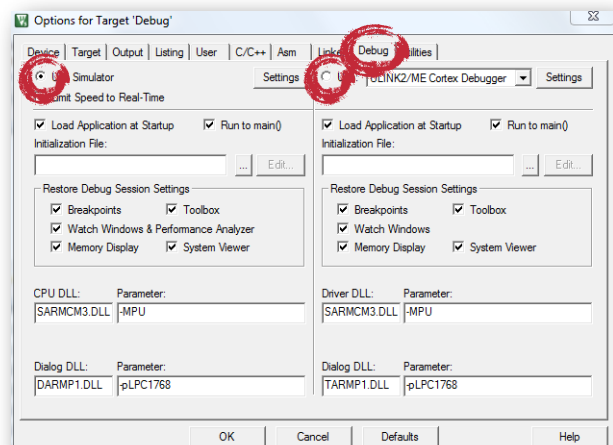


ports. If we now click on the  RUN button, we can see how bit "28" of FIO1SET toggles in the 100ms rhythm and bit "29" in 50ms intervals.

**Tip: Switch Target from Simulator to Hardware.**

As a target we have, by default, the simulator. If you have hardware beside you with a Cortex-M3 processor you can use the hardware mode as shown in the picture blow. This can be done, as long as you are not in debug mode.

(check-> Use ULINK Cortex Debugger)



## Last but not least

Please note that we can only scratch the surface of modeling with Rhapsody with this tutorial. UML and Rhapsody are book filling, powerful subjects that can, unfortunately, not be packed in a short script. Please do not be discouraged by this!

With just a few solid basic skills, most projects are not a big hurdle for you anymore. Even if the UML2 has over thirteen different types of diagrams, for your first projects, you need only three of them (which lowers the mountain a bit).

Now, if you are keen for more, we can highly recommend you our "Embedded UML Start-Up Training". Learn in an compact training, all about the possibilities of the UML and how to deal with Rhapsody.

More information about this course is available on our homepage:

<http://www.willert.de/events/>

For questions about Rhapsody and the UML, please use this forum:

<http://www.willert.de/uml-forum/>

We hope we was able to arouse your curiosity in this fascinating subject and would be delighted to see you in our next UML - Startup Training.



Editor:

WILLERT SOFTWARE TOOLS GMBH  
Hannoversche Straße 21  
31675 Bückeburg  
Tel.: +49 5722 - 9678 60

[www.willert.de](http://www.willert.de)      [info@willert.de](mailto:info@willert.de)

Tel.: +49 5722 9678 - 60

Author: Marco Matuschek