

Department of Electrical Engineering and Information Technology

TUTORIAL

EMBEDDED SYSTEMS

UML CODE GENERATION

Welcome to the UML code generation tutorial!

In the next few hours you will learn how to generate C++ Code out of UML diagrams. This Tutorial is based on the “UML Getting Started” tutorial of the Willert company.

So, let's start!

supported by



Software



IBM® Rational® Rhapsody® for C++

With Rhapsody, we create UMLDiagrams and generate C++-code from them. The “Blinky” example will be modeled completely with the help of this environment. (Version 8.1.4)



KEIL MDK ARM

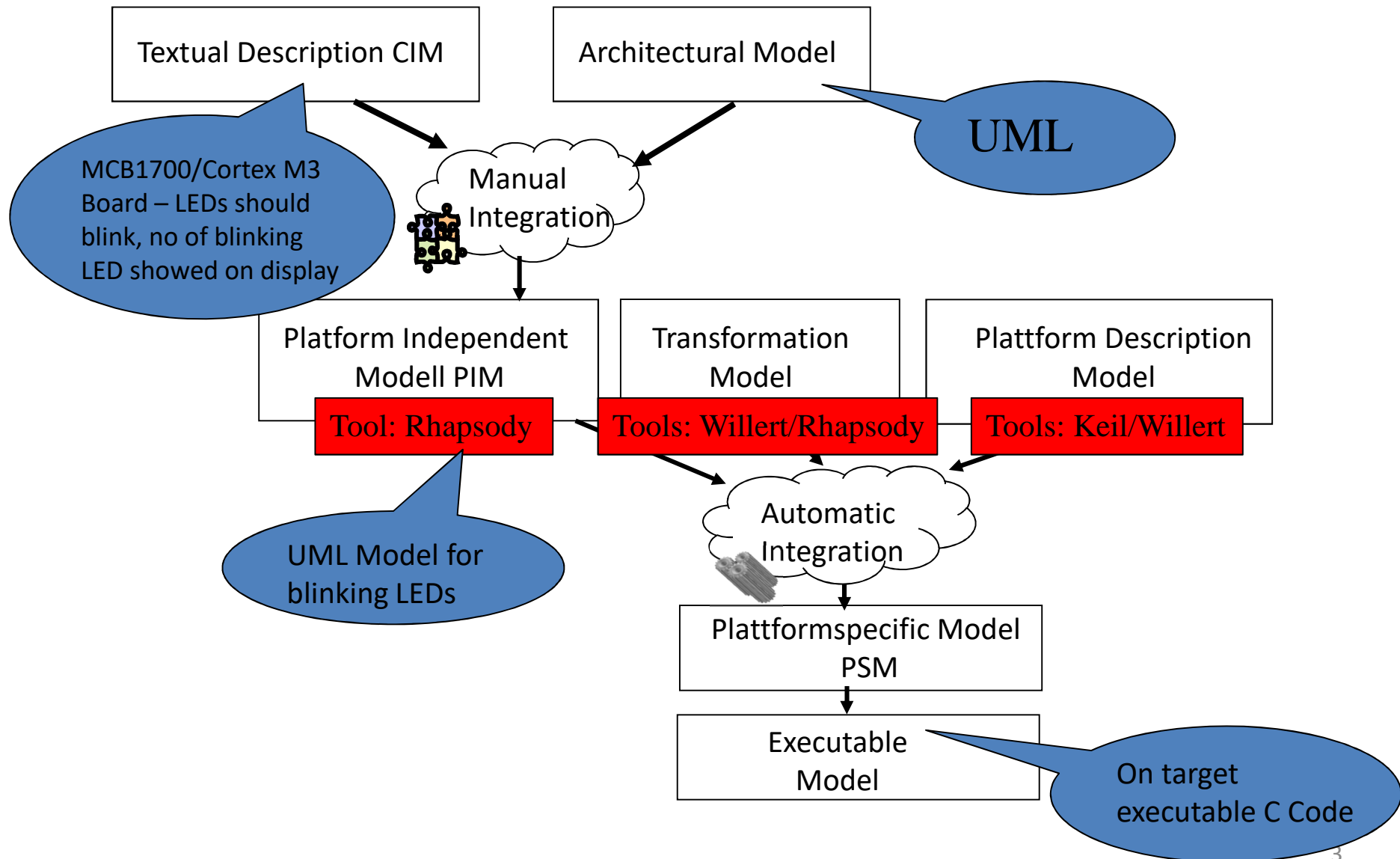
We need the Keil μ Vision IDE to flash our executable model to the target or to let it run in the built-in simulator. On top of that the IDE offers additional debugging possibilities for our model. (Version 5.22.00)



Realtime eXecution Framework

The Willert Embedded UML RXF™ combines Rhapsody with the Keil μ Vision IDE. It optimizes the code generation for a resource friendly use in an embedded real-time environment. (Version 6.01)

MDA Principle



Embedded UML Studio® (Willert Software Tools)

Rhapsody® UML Tool (I-Logix)

Rhapsody® Code Generation (I-Logix)

Embedded UML RXF (Willert Software Tools)
(Embedded UML Realtime eXecution Framework)

OSAL (Willert Software Tools)
(Operating System Abstraction Layer)

Keil™ RTX ARM
(RealView Real-Time Library)

Embedded OO RTX
(Willert Software Tools)
(Embedded Object Oriented
RealTime eXecution System)

Keil™ µVision (C/C++ Compiler)

Alternativ

11 April, 2007

26

Willert Software Tools

Available predefined types:

char	char*	double
float	int	unsigned long
long	long double	short unsigned
void	void*	unsigned short
RiCString	RiCBoolean	OMString

Automatically generated functions

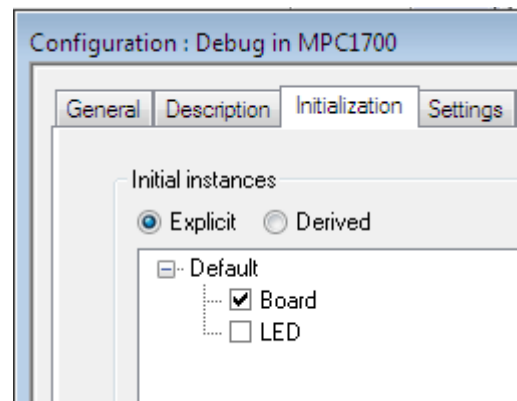
- Rhapsody generates operations and objects to **create**, to **initialize**, to **cleanup**, and to **destroy** objects.
- **Class constructors**
called with the creation of an object
- **Class destructors**
called when an object is deleted

Reactive Objects

- Reactive Objects are objects which can **receive and deal with events**. Usually they are state dependent:
 - They are **described by a state machine** or
 - They have an **event reception**
- Rhapsody automatically generates state based functions
 - for entry of a state
 - for consuming events
 - for state request
 - for leaving a state

Objects

- Functionality of classes is only usable by instances / objects
- Creation of objects is done directly in Object Model Diagram or Model
- Automatically generated Objects at program start are possible:



- **Attention:** Always decide for reasonable objects, e.g. for a LED bar with 50 LEDs you should not create 50 single objects

Events

Events:

- enable asynchronous communication between objects and via statecharts
- created inside the class which receives the event or alternatively defined as global operation
- defined by using the Makro CGEN which is available in Rhapsody:
CGEN(&Receiver, eventToBeSent());
e.g. CGEN(&myLED, evLEDChange());
- The receiver of an event may be a global object, a local object or a subobject.
- Events can transport information by using **parameters**.
The receiver of an event can get the parameter values via
params->parametername

Note:

Creation of an event with parameters in Rhapsody: Create an operation, insert parameters, change operation type to type reception.

Other useful functions

Using the **IS_IN** function you can decide whether an object is in a special state.

Notation:

```
IS_IN( &object, &object_specificState)
```

e.g.

```
if ( IS_IN( me, me_LEDOn ) ) ...  
return value is 0 or 1 (boolean value)
```

Further Informations

- **Roundtrip** means to generate a UML model out of existing code. You don't want to use this feature in the lab project.

Always choose „No to all“

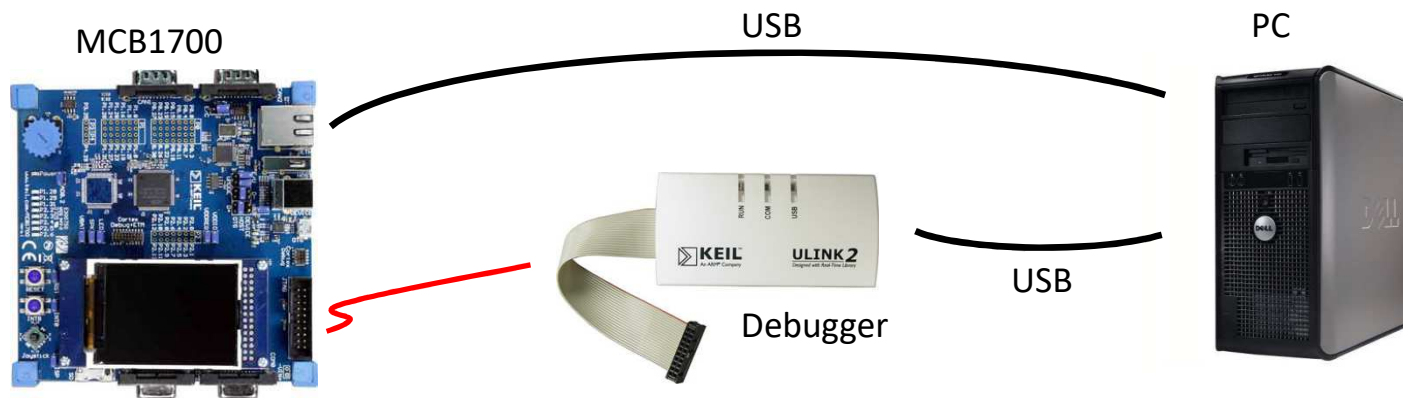
If you are asked for roundtrip multiple times you probably have „old“ or „wrong“ elements in your model.

- The path to the keil-Project is a configuration in Rhapsody. It can be changed but it is more convenient to keep it as defined.

MCB1700 Board

We use a Keil MCB1700 evaluation board with a LP1768 Cortex M3 processor
To debug our Software we use the Keil Ulink2 Debugger, which allows us to debug our embedded programs on our target.

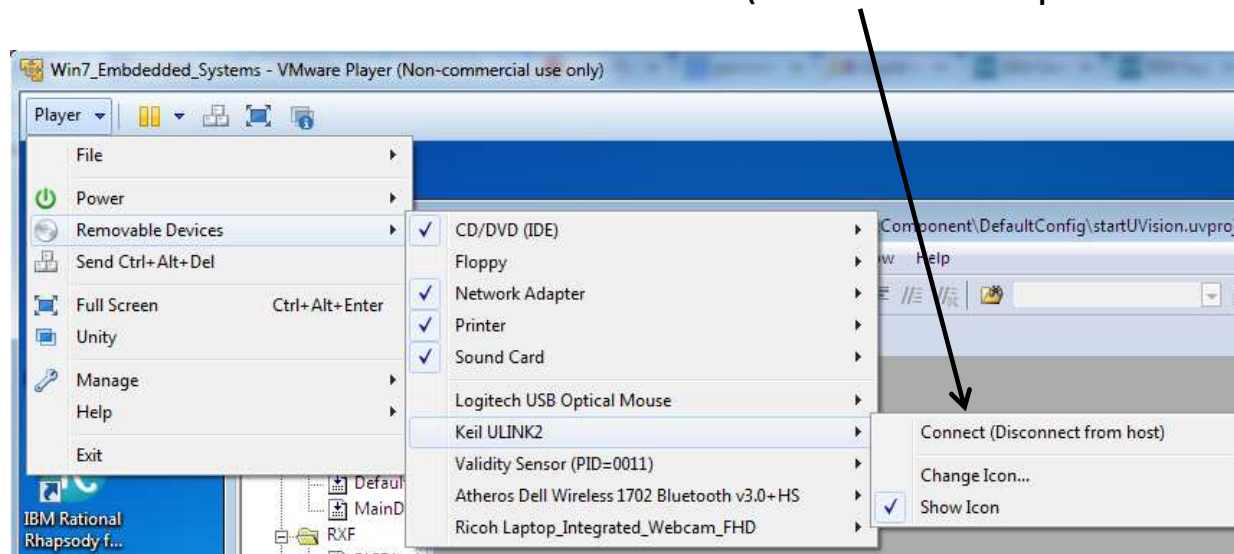
- Connect the JTAG plug of the ULINK2 to the connection jack of the MCB1700
- Now connect the two USB cables to ULINK2 and the 1700
- Make sure that the POWER LED (red) is on



MCB1700 Board

Use the Virtual Machine **VM_SS2017_EM SYS**

In case that Keil has problems to find the debugger after you connected it to the PC: Connect Keil with the VM (as seen in the picture below)



Usually the USB Devices are detected and connected automatically.

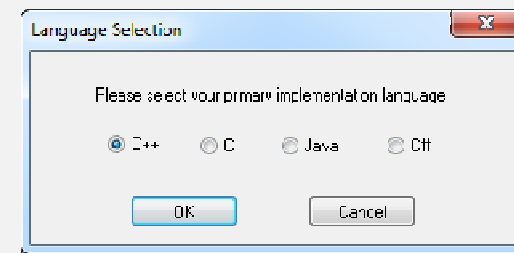
Blinky project *GettingStarted*

Let's start *IBM Rational Rhapsody Developer for C++* in the startup menu. Off-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.

On the Rhapsody welcome screen of we click on the **Next** symbol, or choose **File/New** from the menu.



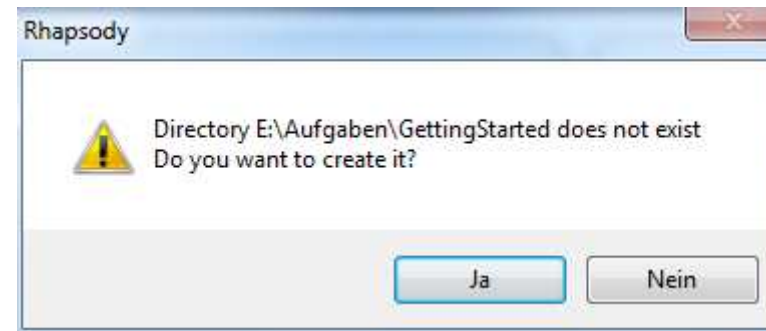
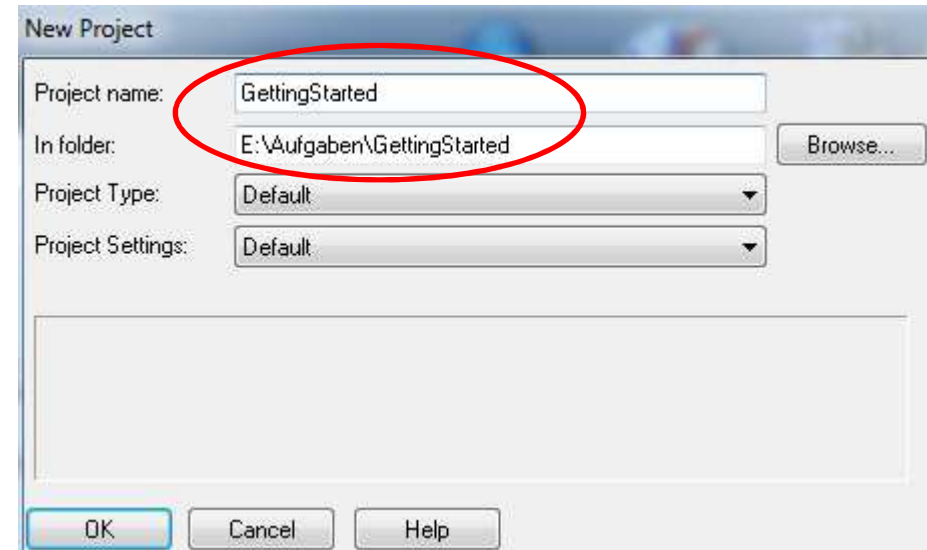
If you started *IBM Rational Rhapsody Architect for Software* choose **C++** as your primary implementation language.



Getting Started project

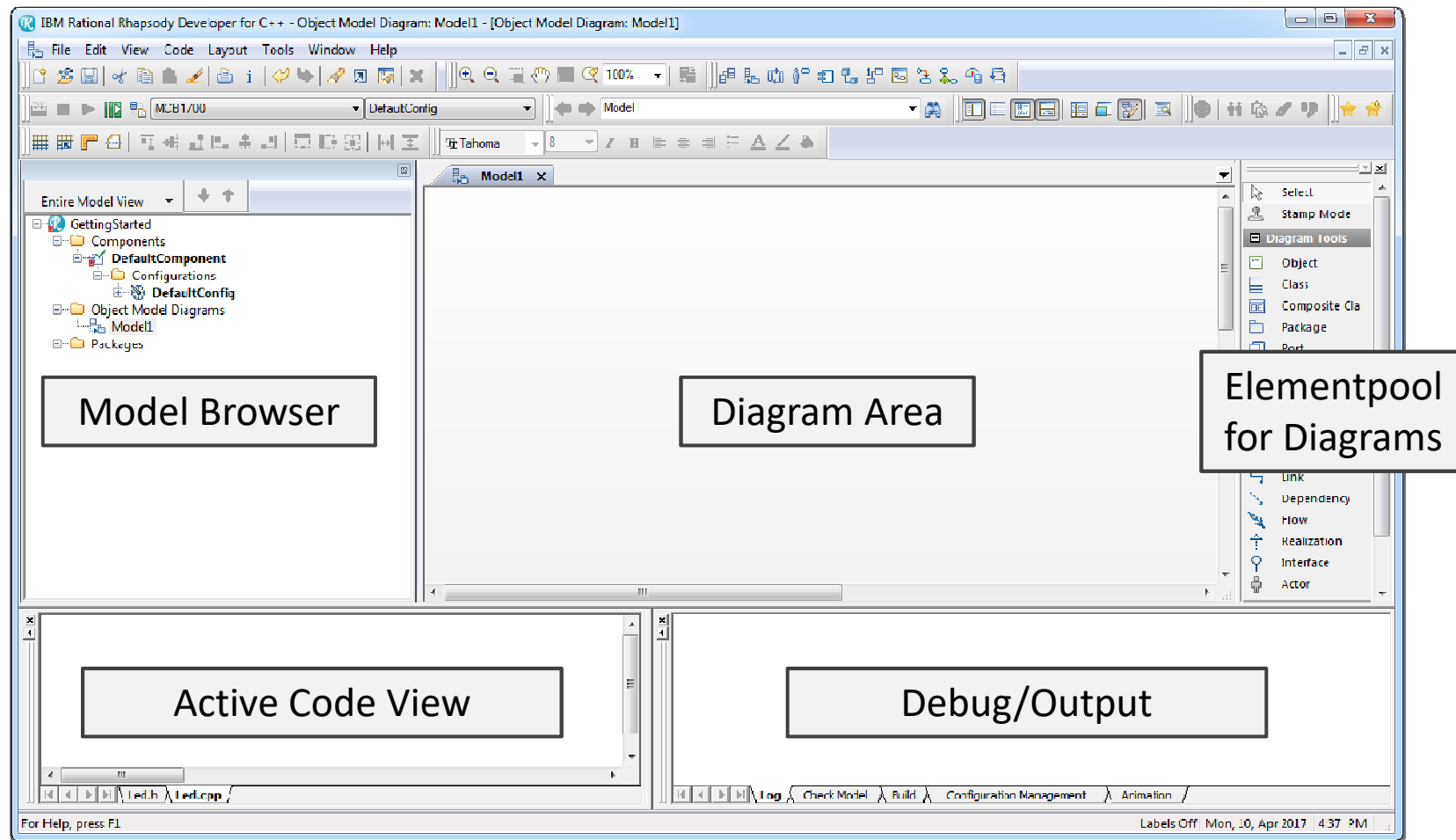
We determine an appropriate name for our project, e.g. GettingStarted in a folder in our working directory.

Yes you want to...



Getting Started project

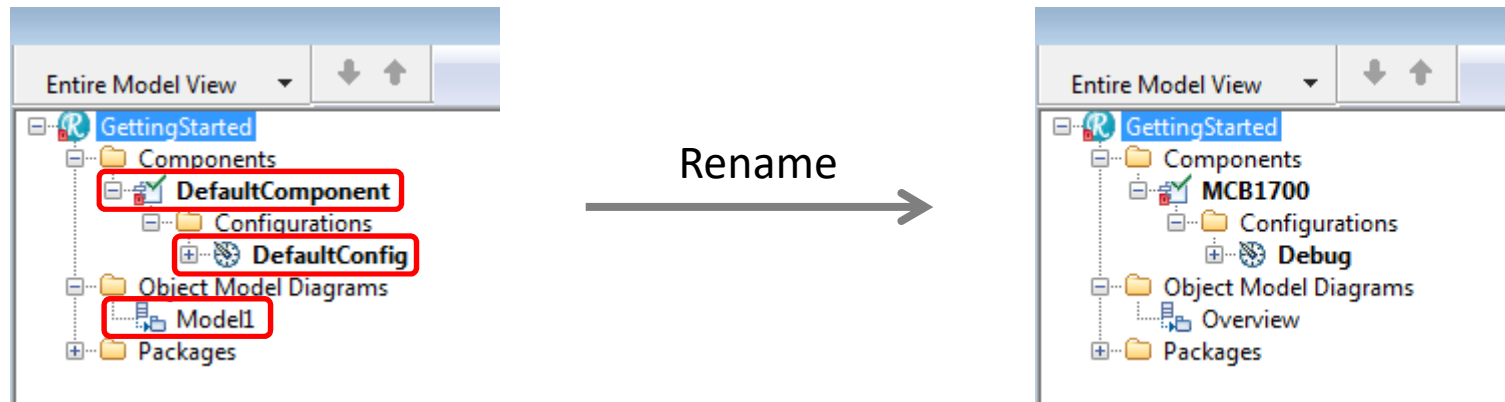
This is the user interface of Rhapsody. Please make yourself familiar with it, because this is the program we work with most of the time.



Getting Started project

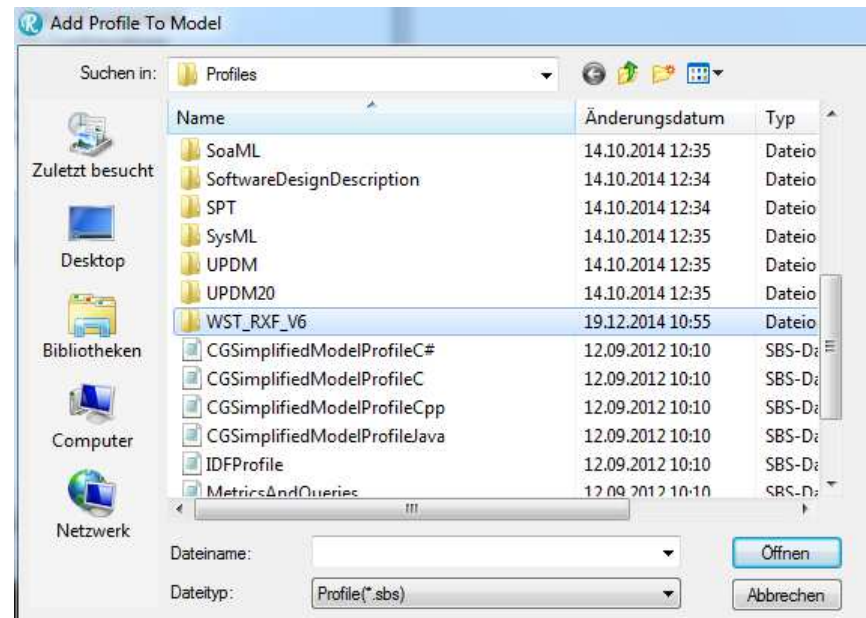
We take a look at the model browser. Similar to the Windows Explorer, you can navigate using the (+/-) symbols through the model.

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 a slow double-click on the element to be renamed.

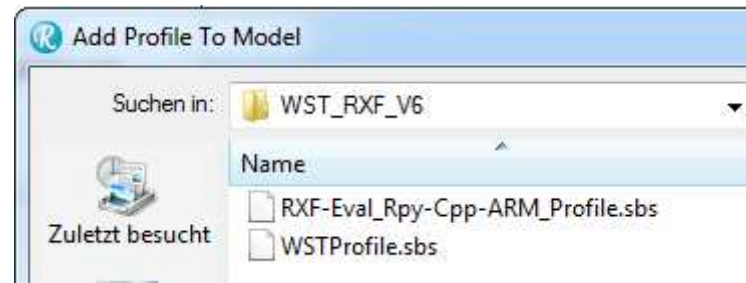


Select Profiles

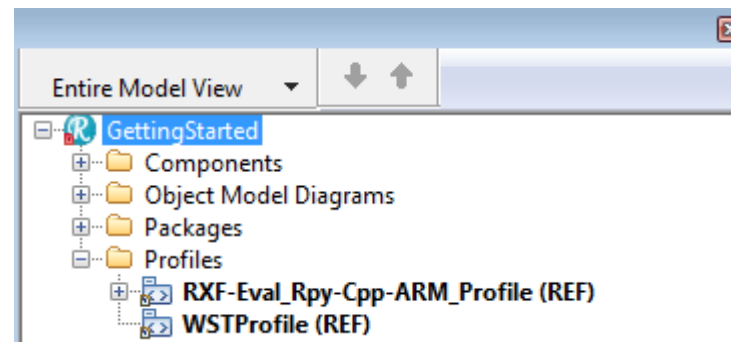
Chose the WST-Profile for the Willert RT-OS select *File* → *Add Profile to Model* and select the folder *WST_RXF_V6*.



Add both profiles.

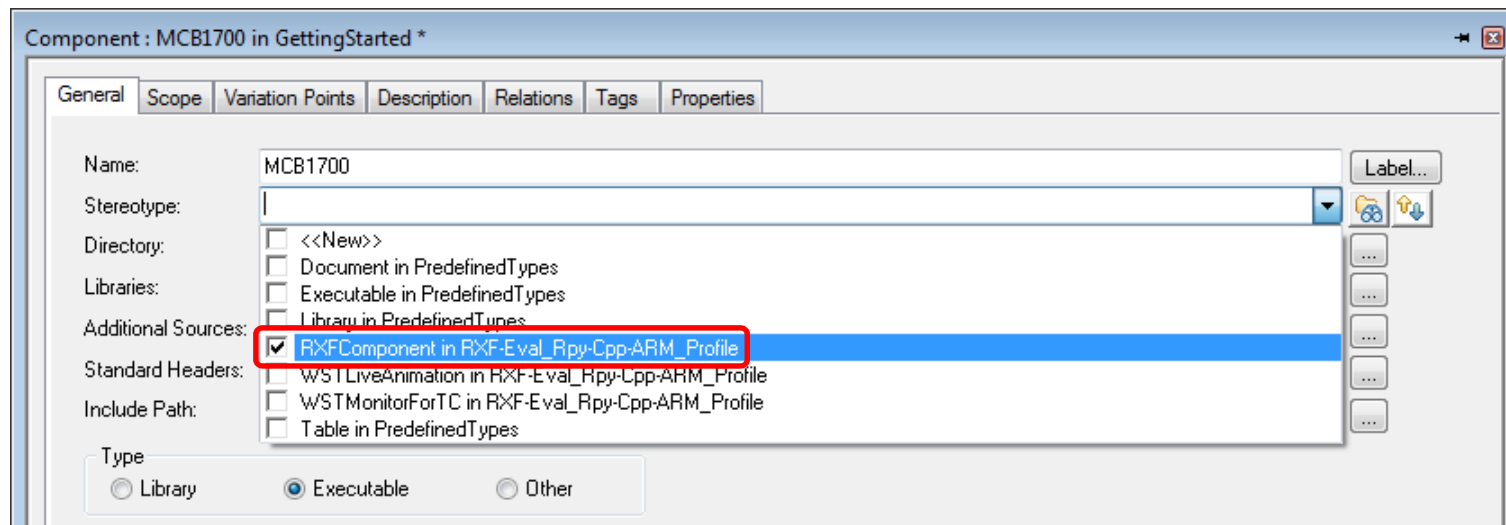


The Profiles folder of your Project should look like this.

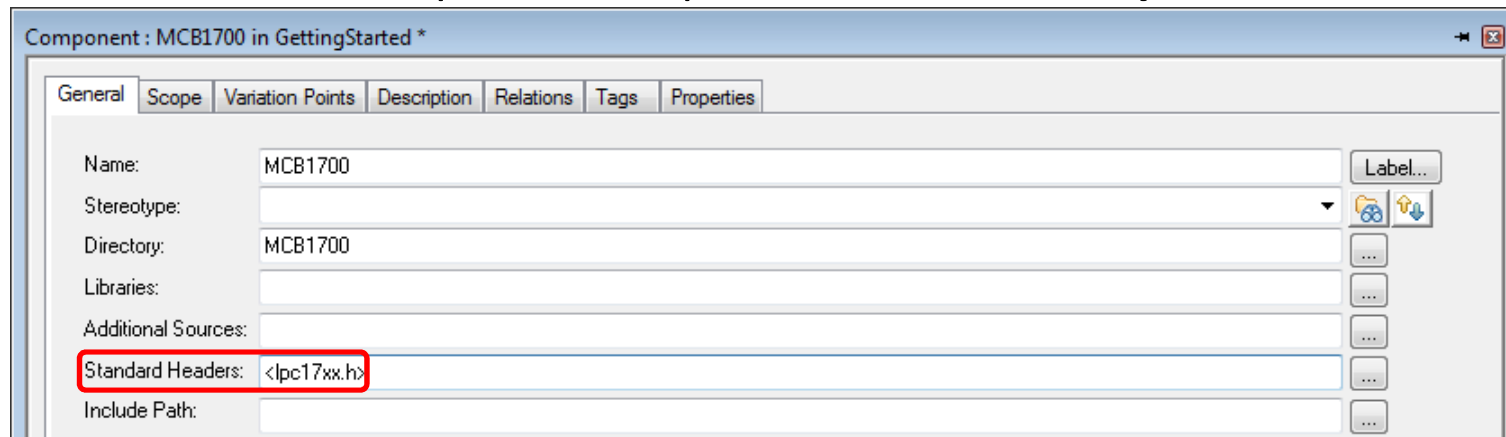


GettingStarted project

Double-click on the MCB1700 component and select the following stereotype in the pop-up window: ***RXFComponent...***



For the hardware specific components add the file ***lpc17xx.h***:

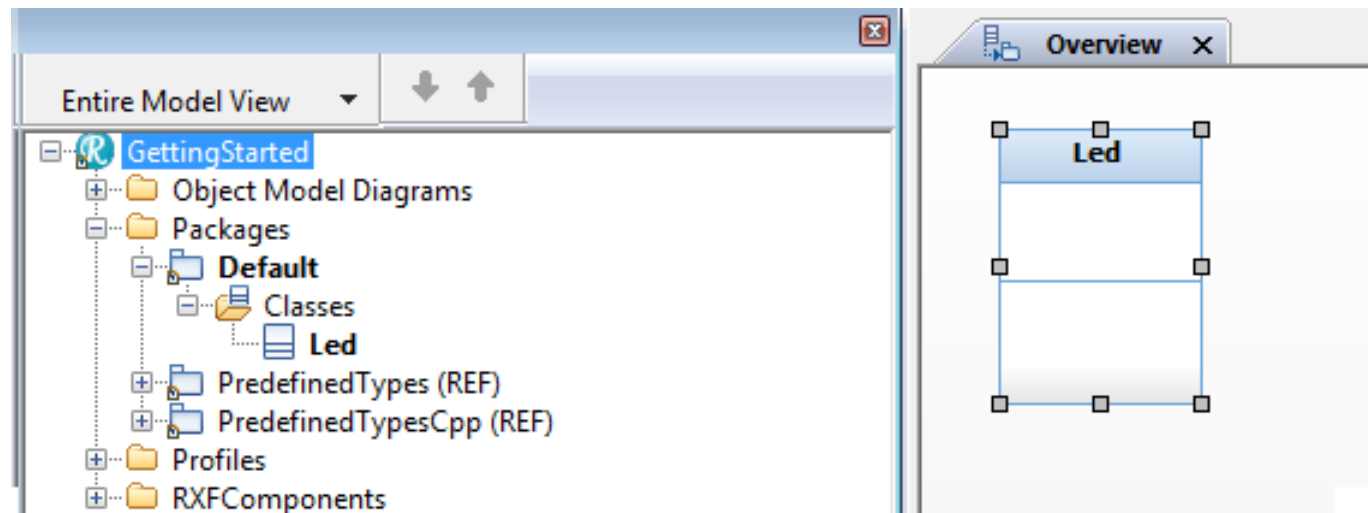


GettingStarted project


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 draw a class in our diagram with selecting the class symbol from our element pool and name the class `Led`.

This class can also be found in the model browser under:

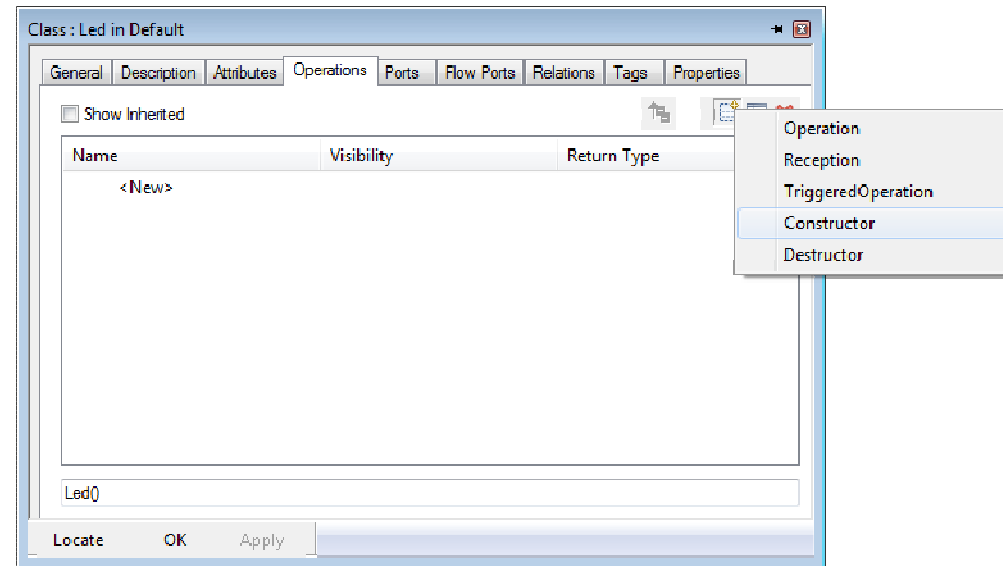
Packages / Default / Classes / `Led`



Blinky project – Methods

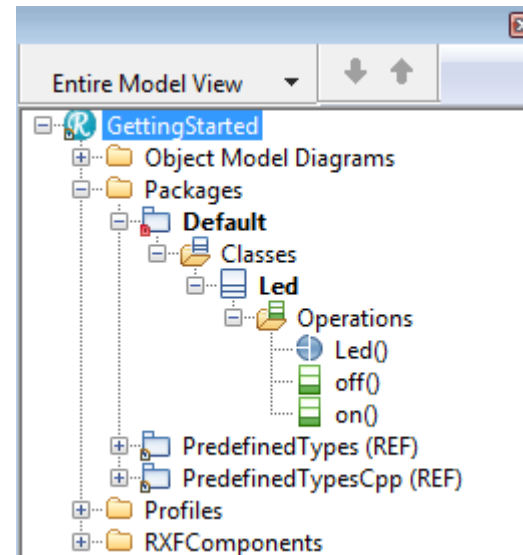
Switch to the *Operations*-Tab and click on the *New* symbol  and a *Constructor*.

Rhapsody will ask you which arguments the constructor should get. We will do that later. Just click *OK*.



Add the *Operations* `on()` and `off()` in the same way.

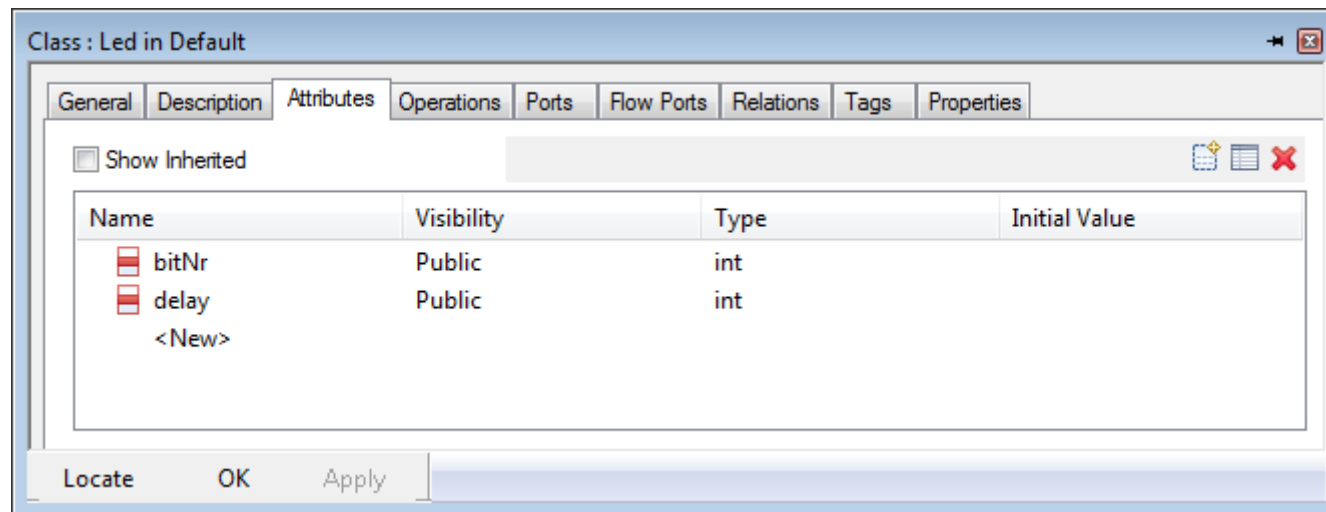
Your class `Led` should have three operations now and look like this:




Blinky project – Methods and Attributes

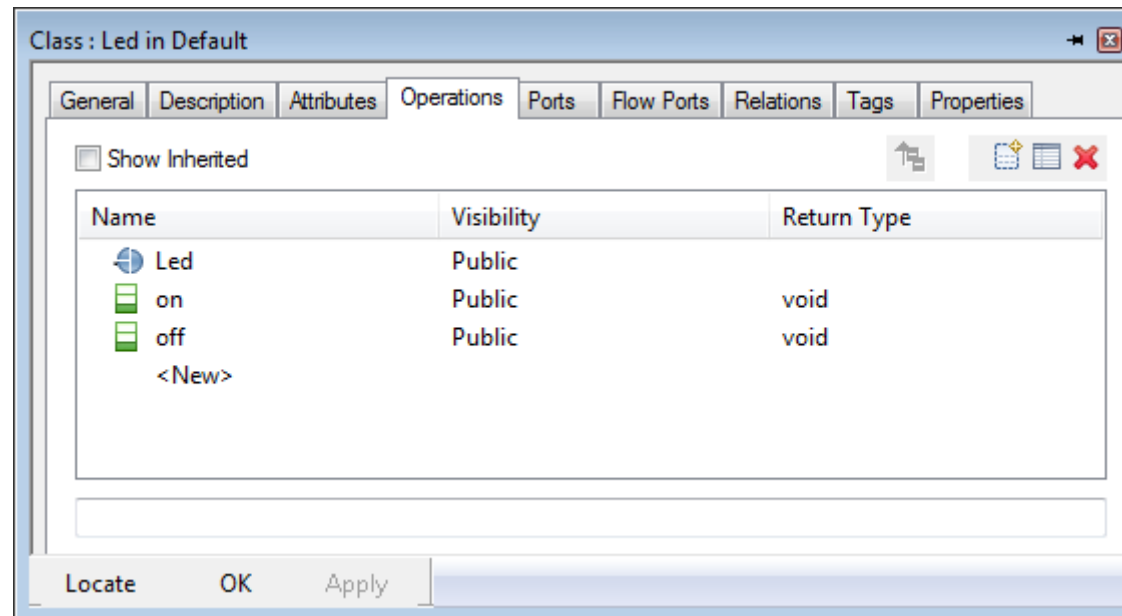
Next, we add two attributes to our class. The first one is `bitNr`, where the information which port is connected to which Led is located. The second one is `delay`. It's the delay time between on and off.

Open the features of our class `Led` (double-click or right-click → Features) and create the two attributes at the *Attributes*-Tab.



Blinky project – Methods and Attributes

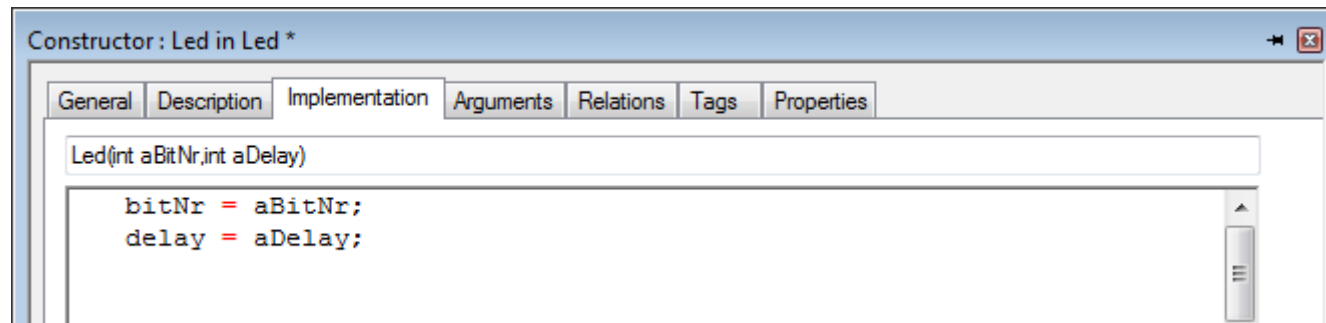
So open the *Features* of the constructor `Led()` again and embark in the *Argument* tab. We create the arguments `aBitNr` and `aDelay` there, using the *New* symbol  .



Blinky project – Attributes

We have three empty methods in our class (`Led()`, `off()`, `on()`) and two attributes (`bitNr`, `delay`)

In order to get both variables initialized at create time we will add the following C++-code to the Implementation-Tab of the constructor `Led()`.



What happened here?

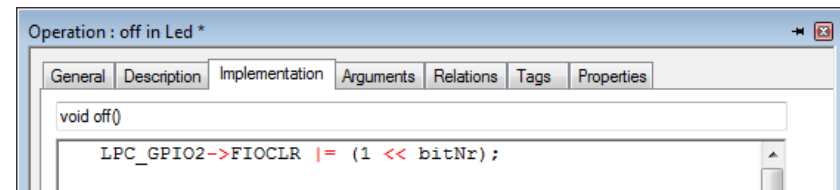
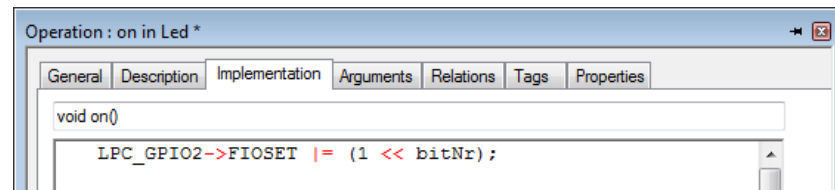
If we later create an object of the class `Led`, the constructor `Led()` assigns a number and a delay time passed to the constructor `Led()` as parameters to this object.

Blinky project – Attributes

In order to define one of the eight LEDs on the MCB1700 as output, we need to set the appropriate bit in the FIODIR register of the CortexM3. We do this with two code lines in the *Implementation*-Tab of the constructor `Led(int aBitNr,int aDelay)`.

```
bitNr = aBitNr;
delay = aDelay;
LPC_GPIO1 -> FIODIR |= 0xB0000000;
// B=1011 e.g. Pin 28 and 29 with LEDs are set to output pin
// - we will not use these
LPC_GPIO2 -> FIODIR |= 0x0000007C;
// LED 1,2,...5 are addressed by GPIO2,
// here all LED Pins are set to output
```

Add content to the `on()` and `off()` operations. Just fill in the lines below:

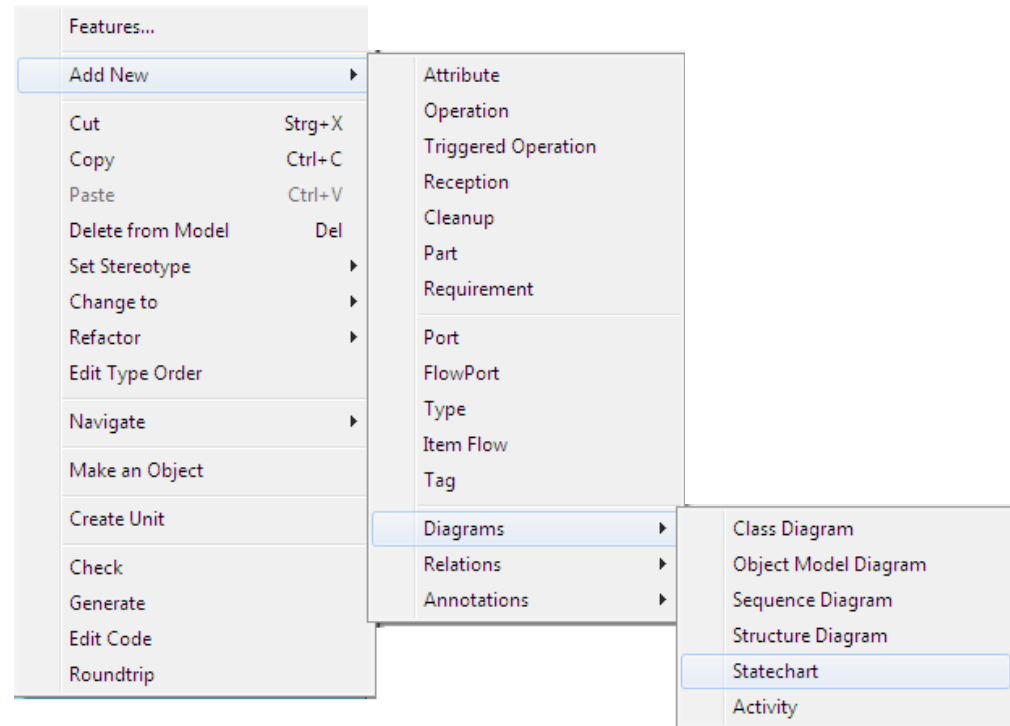


Blinky – Statecharts

Our constructor will be automatically called at the initialization of the LED. But who will now call both our `on()` - and `off()` - operations?

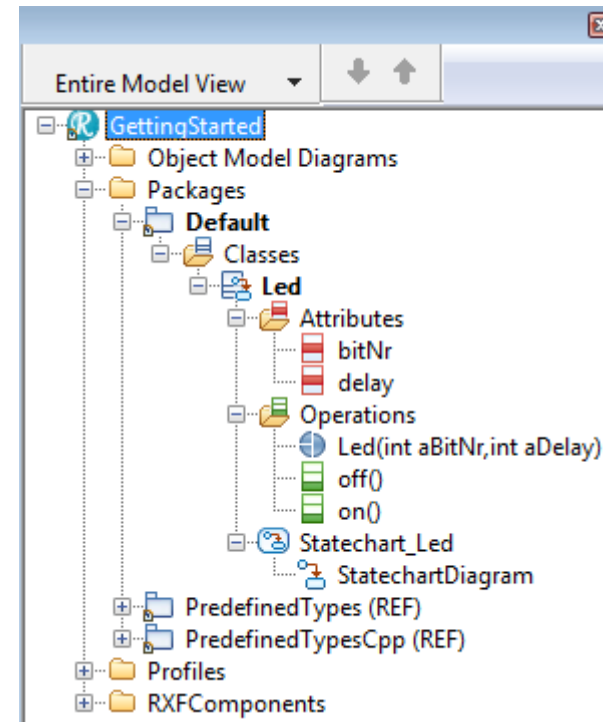
For that we use one of the most commonly used UML diagrams - the Statechart (State Diagram).

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

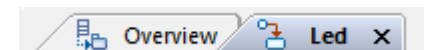


Blinky – Statecharts

Your project should look like this now:



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 state chart.

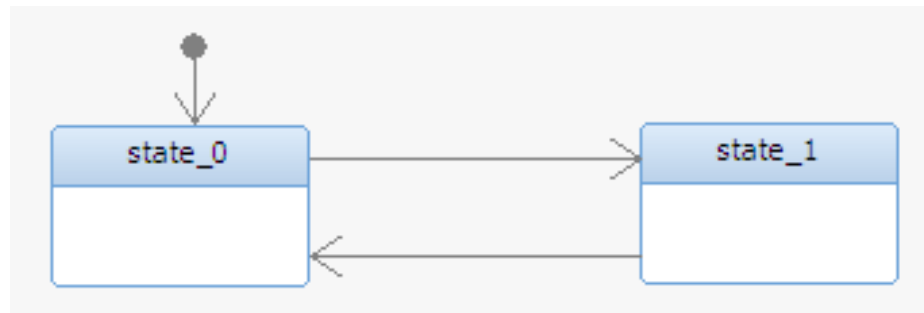


Blinky – State chart

We need three different elements to build our state chart from the element pool. Add two *States* and connect the two of them with one arrow (*Transition*) in each direction. Our chart should look like this now:



The only missing thing is an entry point now. Add it by using the *Default Transition* element.



Blinky – State chart

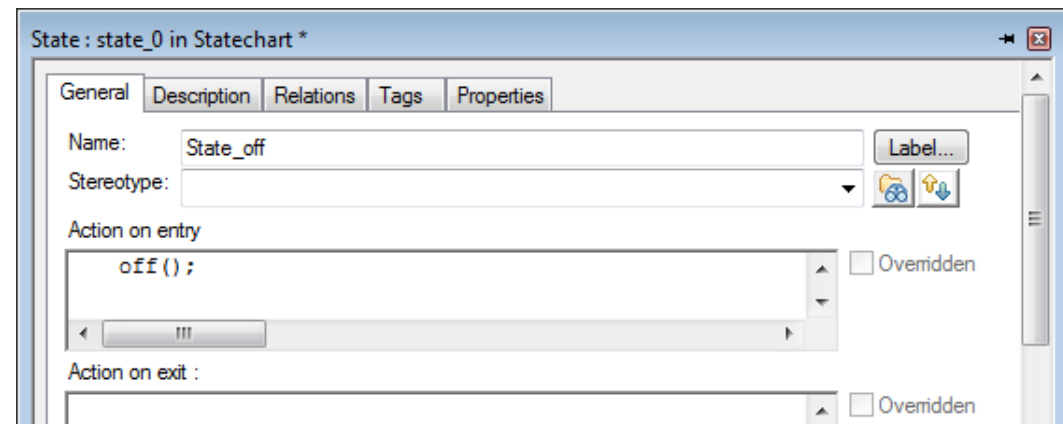


If we want to draw the diagrams more precisely then this toolbar will support us in that:



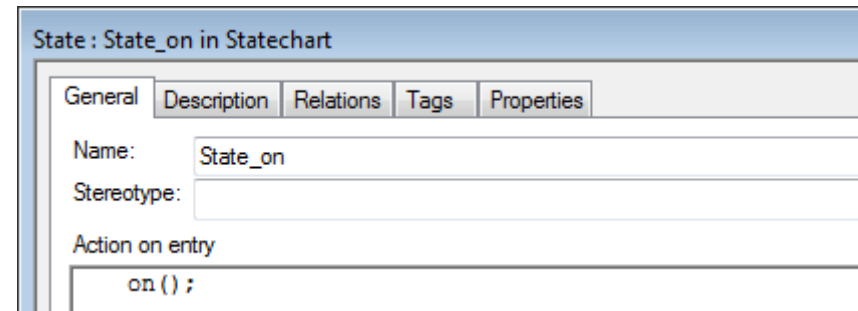
If you select multiple objects in a diagram while keeping the Shift-Key pressed, the symbols in this toolbar will become active. Now you can align the selected objects to the edge of the diagram or to each other. The last selected object is used as pivot point!

Open the `state_0` features and rename it to `State_off` and enter the call of the `off()` operation at *Action on entry*.



Blinky – State chart

Enter the similar name and operation into the general tab of `State_on`.



State : State_on in Statechart

General Description Relations Tags Properties

Name: State_on

Stereotype:

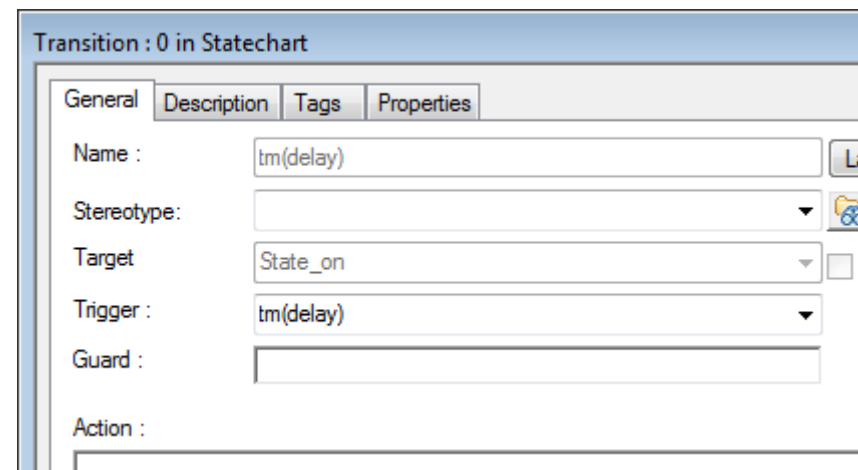
Action on entry

on () ;

In the current situation our state chart would 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. Enter `tm(delay)` in both transitions as *Trigger*.



Transition : 0 in Statechart

General Description Tags Properties

Name : tm(delay)

Stereotype:

Target State_on

Trigger : tm(delay)

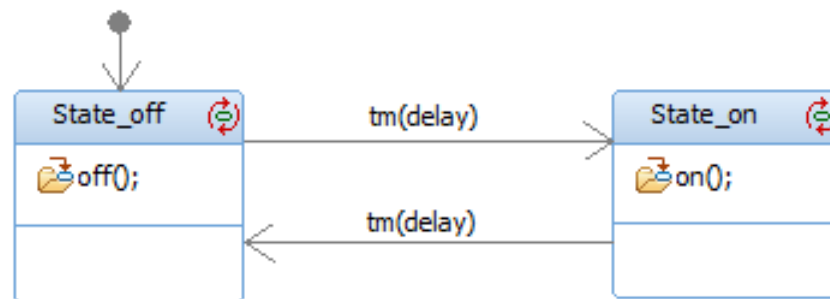
Guard :


Action :

Blinky – State chart

The `tm(me->delay)` uses the system clock of the CortexM3, to delay the transition for the time we specified. (We will initialize the attribute with a value later in this tutorial).

Our finished state chart should look like this:



A click on the symbol  makes the *Action on entry/exit* visible in the state icon.

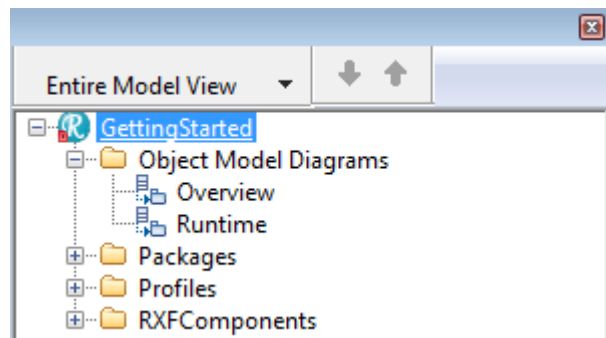
Blinky – Instances of a class

The class `Led` is now complete.

To create an object of our class `Led` at run-time we have to initialize the class. In Rhapsody there are multiple alternatives to achieve this. The simplest possibility looks as follows:

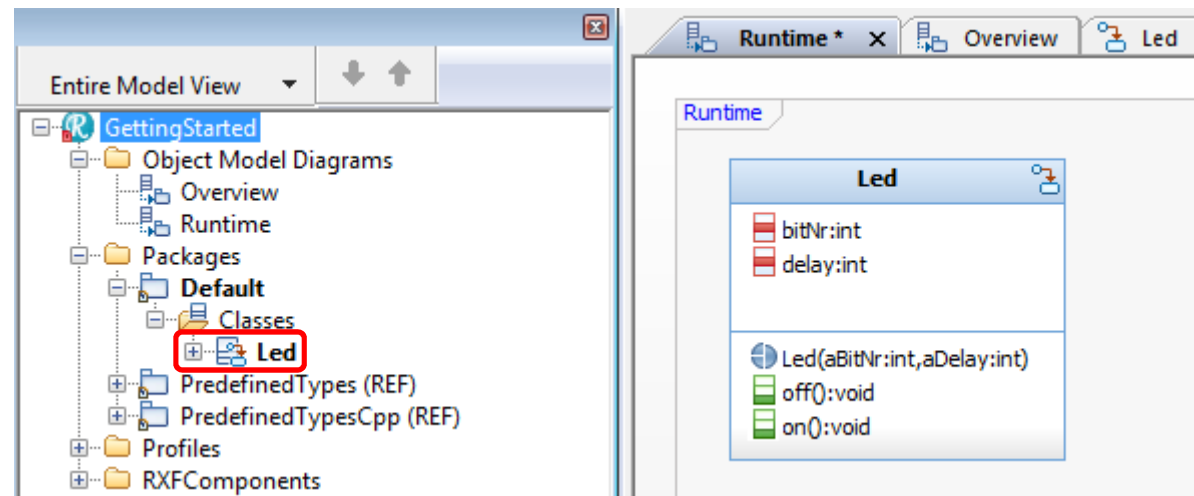
We create a new *Object Model Diagram* to differentiate between classes and instances – to use the existing diagram would also be possible

Right click on *Object Model Diagrams ...* and call it *Runtime*.



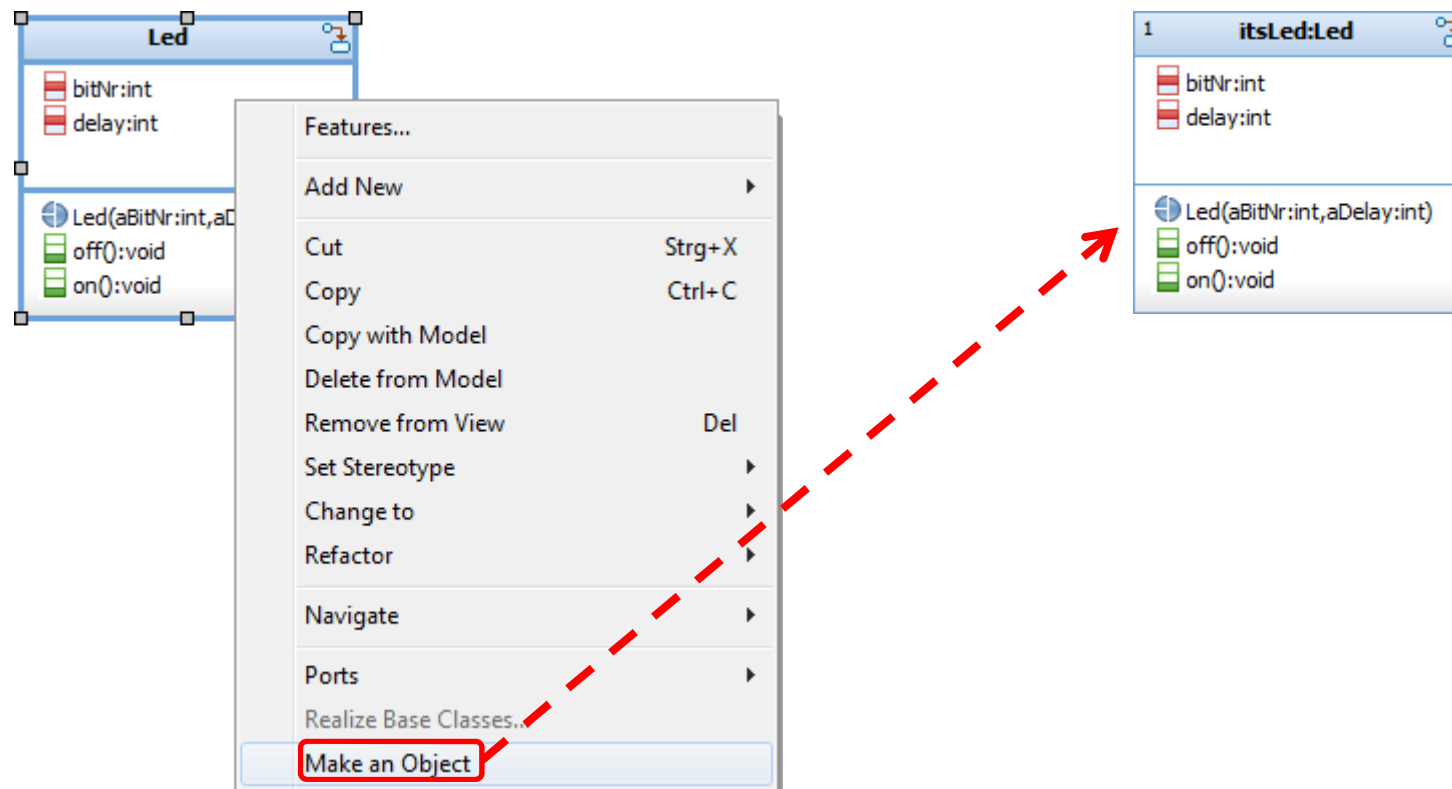
Blinky – Instances of a class

Now drag the class `Led` from the model browser and drop it in the *Runtime OMD*:



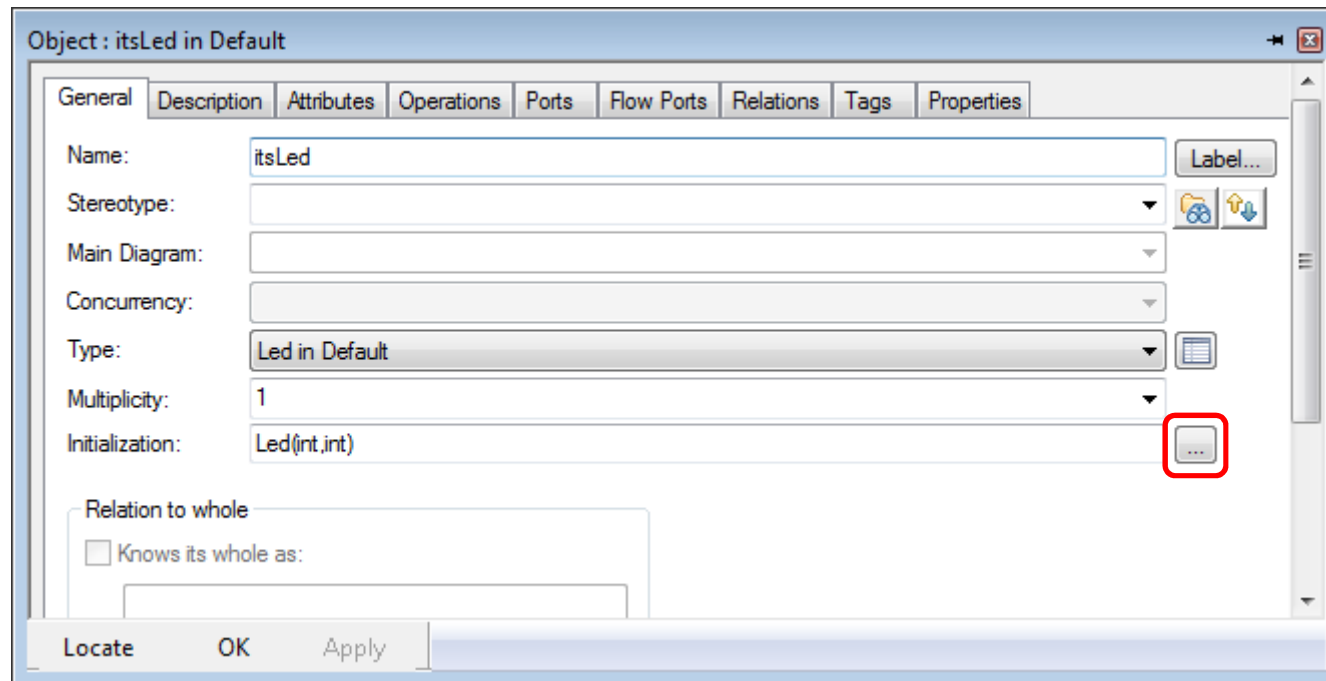
Blinky – Instances of a class

Right-click on the class `Led` in the *Runtime* and select: Make an Object
At runtime an object will be created at run-time out of our class `Led`. It is called `itsLed`.



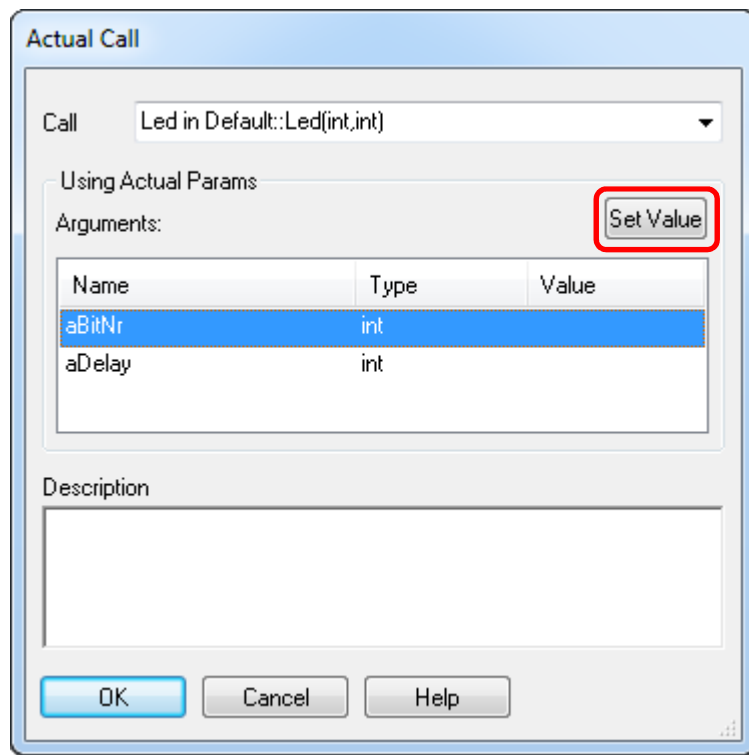
Blinky – Instances of a class

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.

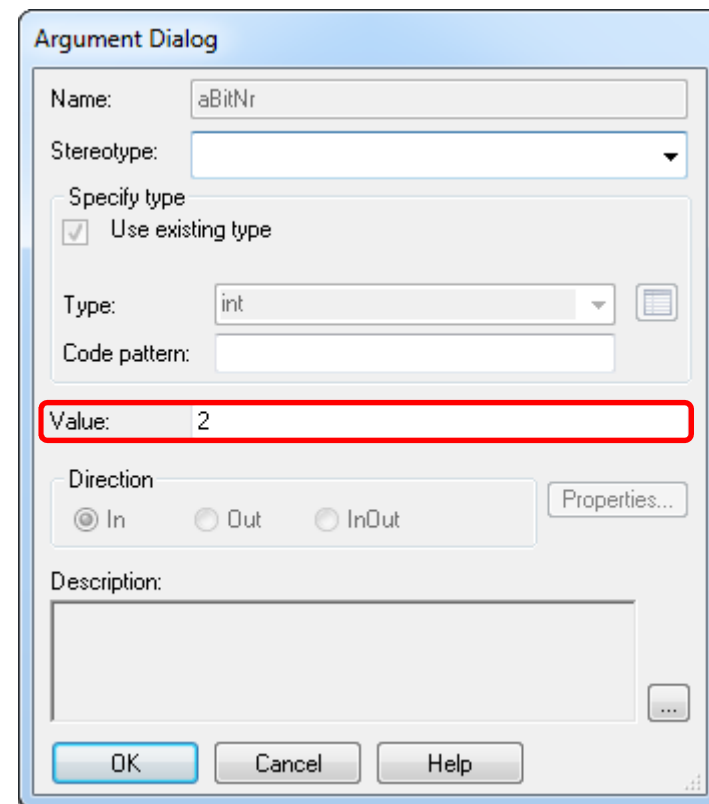


Blinky – Instances of a class

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



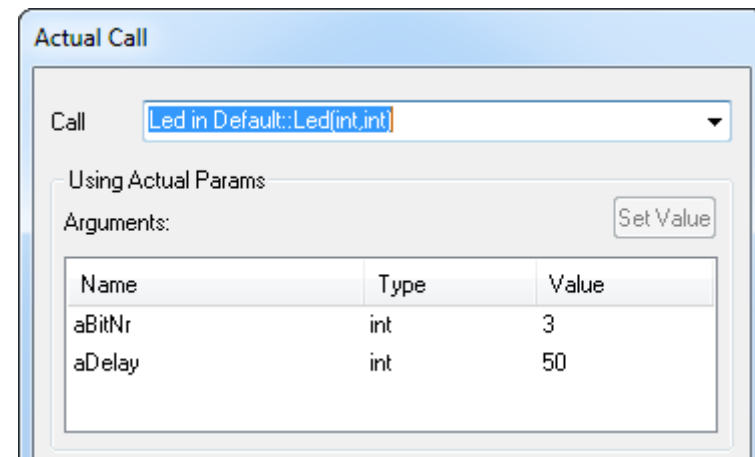
To address the *LED P2.2* on the *Keil MCB1700* we enter the value 2, in the Argument Dialog under *Value*.



Blinky – Instances of a class

We repeat the same now with the argument `aDelay`. As a value we take *100* (milliseconds). To fully exploit the advantages of object orientation, we will immediately **add a second instance** of our class. So we will again drag the class `Led` from the browser to the *Runtime* diagram, next to the other instance, we right click the class `Led` again and from the context menu select *Make an Object*.

We then click the *General*-Tab under Initialization on the Extend button and enter now for the `aBitNr` a *3* and for the `aDelay` a *50*.

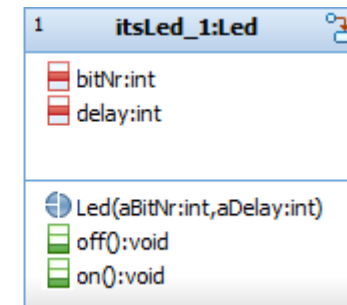
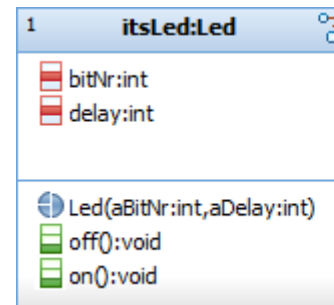


Blinky – Instances of a class

That's it!

We have just, in a very simple way, created two objects from the same class.

Both objects can be simply distinguished using the name and the attributes.



Alternative: Choose the *Object* template from the Elementpool, double-click on the new object and assign Type `Led` in Default.

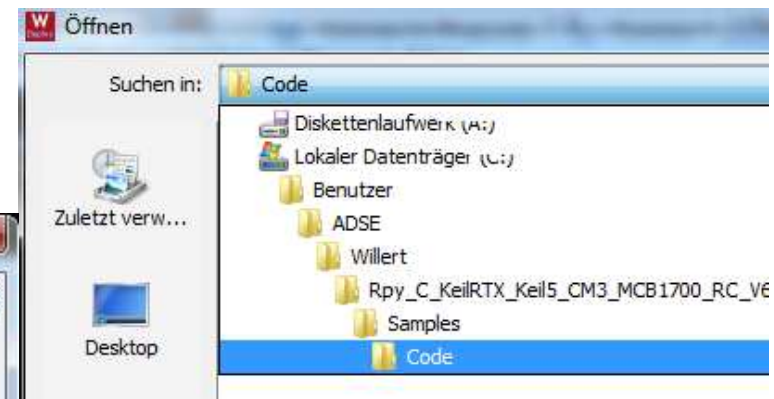
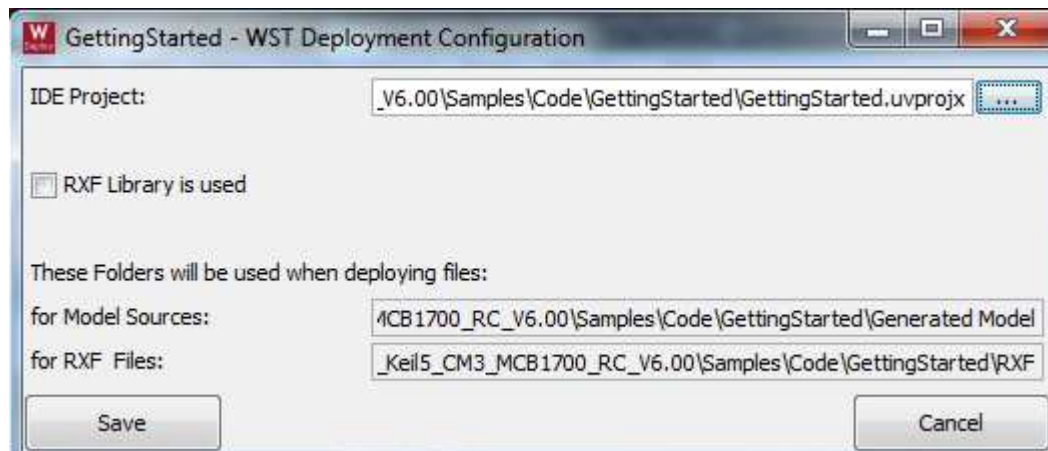
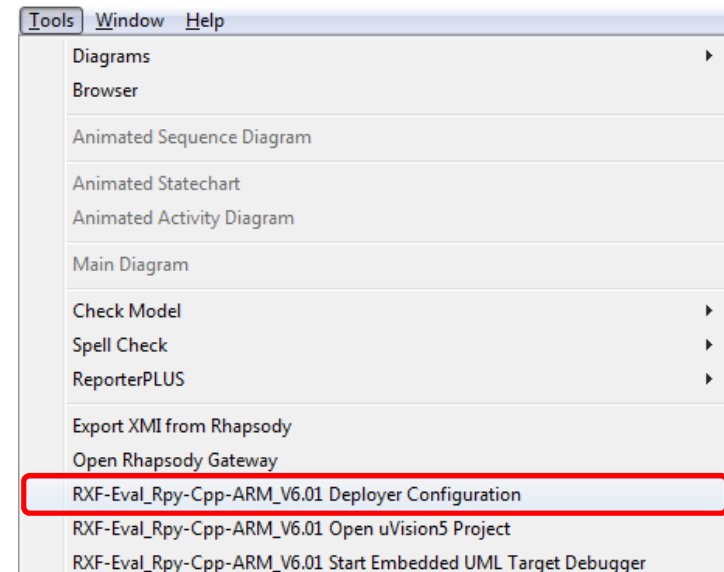
Generate / Make / Run

We now want to generate Code and compile this code with Keil μ Vision

For the Keil μ Vision Project Willert provides a **deployment**. Choose *Tools* → *Rpy-Eval_Cpp... Deployer Configuration*.

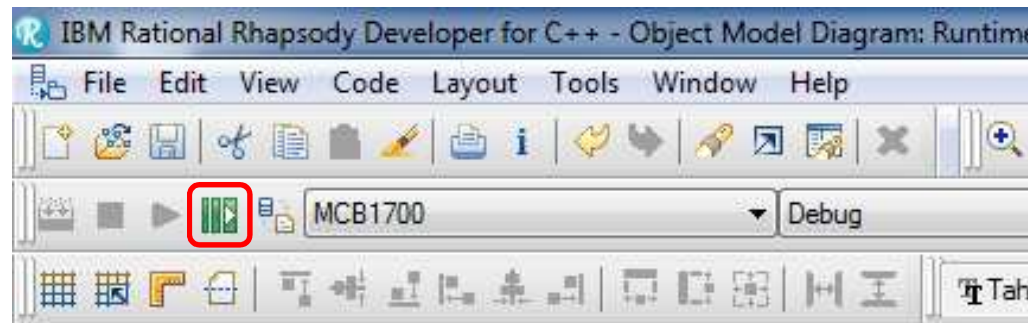
There you have to insert the Path to the Keil Project. Choose the folder Code under:

C:\Benutzer\ADSE\Willert\Rpy_C_KeilRTX_Keil5_CM3_MCB1700_RC_V6.00\Samples\Code\GettingStarted\GettingStarted.uprojx



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 two steps in succession.



In the first step Rhapsody generates 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:

`<yourProjectDirectory>\MCB1700\Debug`

Generate / Make / Run

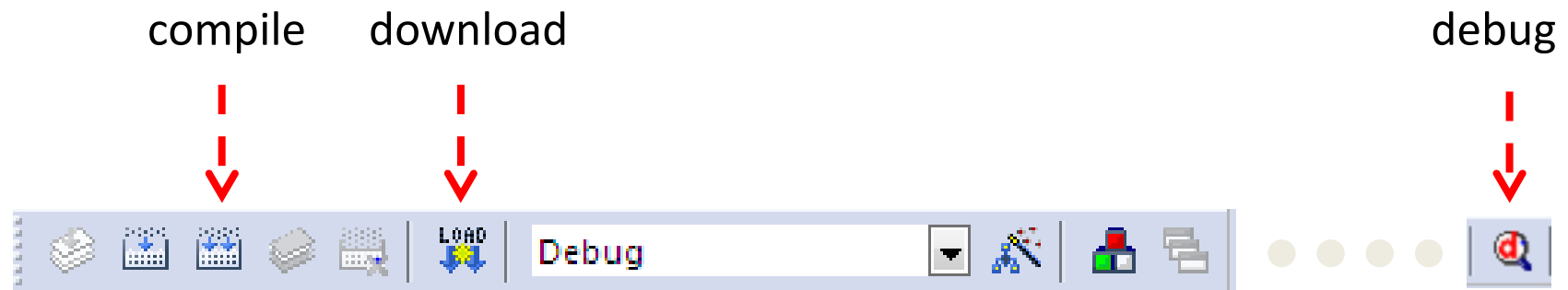
Now start the project in the Keil μ Vision IDE.

C:\Users\ADSE\Willert\Rpy_C_KeilRTX_Keil5_CM3_MCB1700_RC_V6.00\Samples\Code
\GettingStarted\GettingStarted.uprojx



Alternative in Rhapsody: select *Tools* \rightarrow *Rpy-Eval_Cpp... Open uVision5 Project*.

Compile it and load your executable to your virtual device with *Load*. Start the debug mode with the debug-button to start the simulator or to debug your program.



Keil µVision Development Enviroment

Of course, now we want to see if our LEDs will flash in rhythm.
We can **download the program** and see the blinking LED.

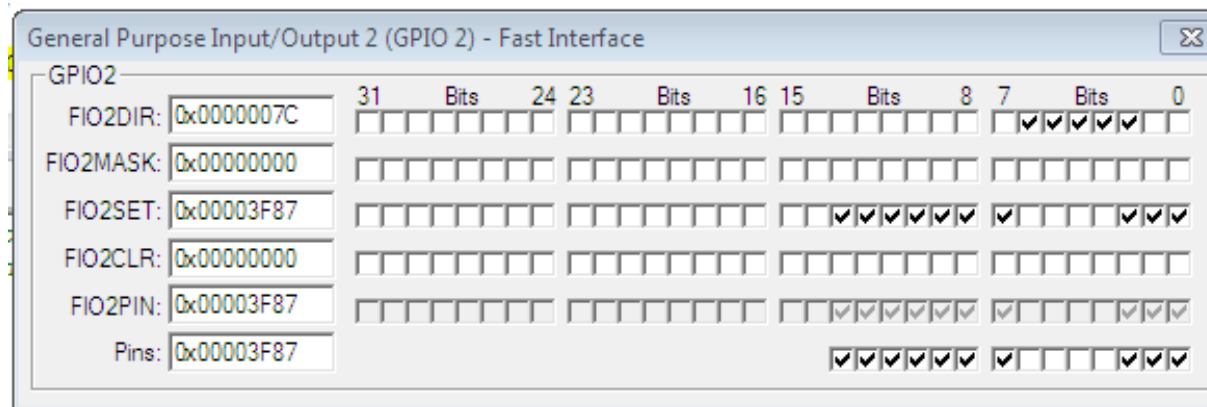


In case of an error or missing target, we can use the Keil Simulator:
We have to activate the debug mode of the Keil IDE.



When we are in debug mode, an IO-window automatically opens or we choose the *General Purpose IO Fast Interface* and Select *GPIO2*. There we can monitor the state of the ports.

If we now click on the *RUN* button, we can see how bit 2 of *FIO2SET* toggles in the 100ms rhythm and bit 3 in 50ms intervals.



We can switch from Simulator to Hardware with Project/OptionsforTarget

Department of Electrical Engineering and Information Technology

LABORATORY GUIDE – EMBEDDED SYSTEMS

PROF. M. VON SCHWERIN

PROF. N. NORMANN

On the next few pages several tasks and exercises are described which have to be solved during the laboratory.

Annotations

- Start the exercise part with a **new project**. Do **not use parameters for the initialization of a class** because in the following we will also use other types of instantiation of an object as shown in the example.
- Create a new project for every exercise.
- Therefore you can select a suitable already existing project



Note that the last project is not always the best fit for the next exercise.

- You should copy selected former project by using *File* → *Save As...* and create a new top directory folder.

Exercise 1

- Modify the project from the tutorial, that:

a) all LED's (P2.2 to P2.6) flash in turn and go out in reverse order.



b) all LED's flash in a row to build a 'moving light'.



Exercise 2

- Modify the time, which the "moving light" takes for one iteration. Therefore implement a code that changes the "Delay" variable dynamically during the runtime.

Exercise 3

- Now the iteration time should be modified by using the Poti, which is right beside the LEDs. Its potential is connected via jumper AD0.2 to the A/D Converter channel 2 of the LPC1768. The potentiometer changes should be processed using polling mode.



For the configuration of the A/D Converter the following code can be used. The page references in the comments belong to the *LPC17xx User manual*.

AD Initialization:

```
// Initialize Poti (page 574)
LPC_PINCON->PINSEL1 &= ~(3<<18); // P0.25 is GPIO - Reset (page 108)
LPC_PINCON->PINSEL1 |= (1<<18);  // P0.25 is AD0.2 - Function 01 (page 108)
```

Exercise 3 (continued)

Sampling:

```
LPC_SC->PCONP |= (1<<12);           // Enable power ADC block (page 63)
LPC_ADC->ADCR = (1<<2) |             // Select AD0.2 pin (page 577)
                (4<<8) |             // ADC clock is 25MHz/5 (page 577)
                (1<<21);             // Enable ADC (page 577)

LPC_ADC->ADCR |= 0x01200000;          // Start A/D conversion now (page 577)
do{
    potiVal = LPC_ADC->ADDR2;         // Read A/D Data Register...
                                     // into potiVal (page 579)
}while((LPC_ADC->ADGDR & 0x80000000) == 0); // Wait until A/D conversion...
                                     // completes (page 578)
LPC_ADC->ADCR &= ~0x01000000;         // Stop A/D Conversion (page 577)
delay = (potiVal >> 4) & 0x0FFF;     // Extract result (page 579)
```



Exercises 1 – 3 have to be checked through a lab advisor. Please mail your projects and submit printouts of your implementations, methods and state charts to the advisors.

Exercise 4

- The LEDs should now be controlled by a new class `Board`. This class sends suitable events to the LED instances to change from *on* to *off* state. Try to use reasonable UML relations in your class diagram. The delay does not have to be controlled by the Poti in this exercise.



- Events can be used for asynchronous communication between objects beyond the borders of their state charts. An event is created as an operation of the class that receives the event or as an global operation. The predefined macro `CGEN` defines an event

```
CGEN( &Receiver, sendEvent() );
```

- Parameters can be passed to events similar to functions and so it is possible to transport information. The receiver can access the parameters via:

```
params->parametername
```

Attention: Create an event with parameters as *Operation* and add its parameter. After that, change the Stereotype of the operation to *Reception*.

Exercise 5

- Print now the number of the currently flashing LEDs onto the LCD Display.

- **Hint:**

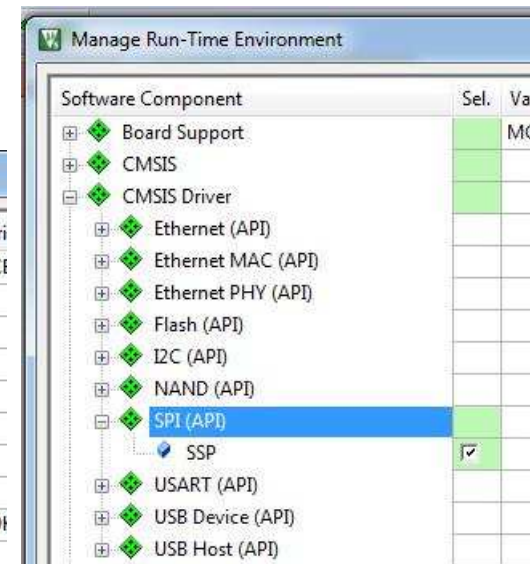
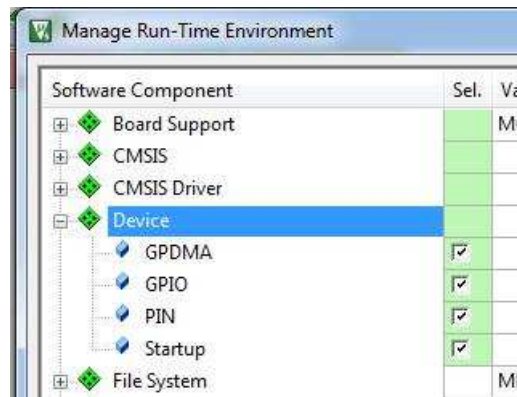
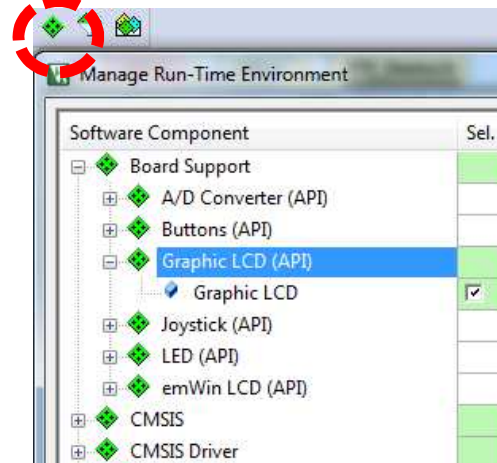
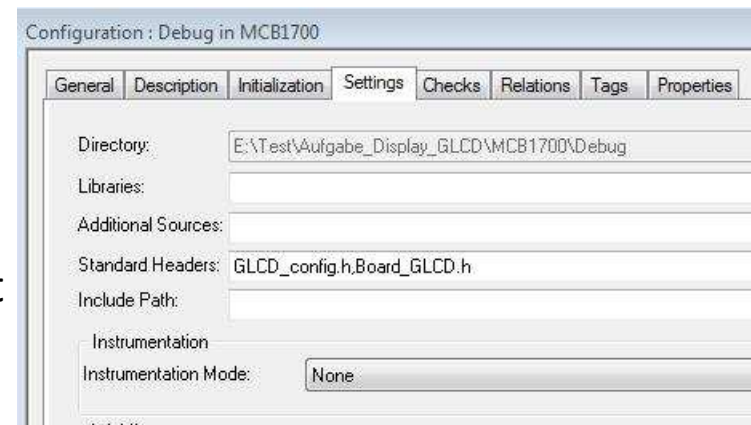
The display requires additional drivers which are provided by the Keil IDE.

In the Rhapsody project you should add two additional headers:

GLCD_config.h, Board_GLCD.h

In Keil choose the Manage Run Time Environment and add

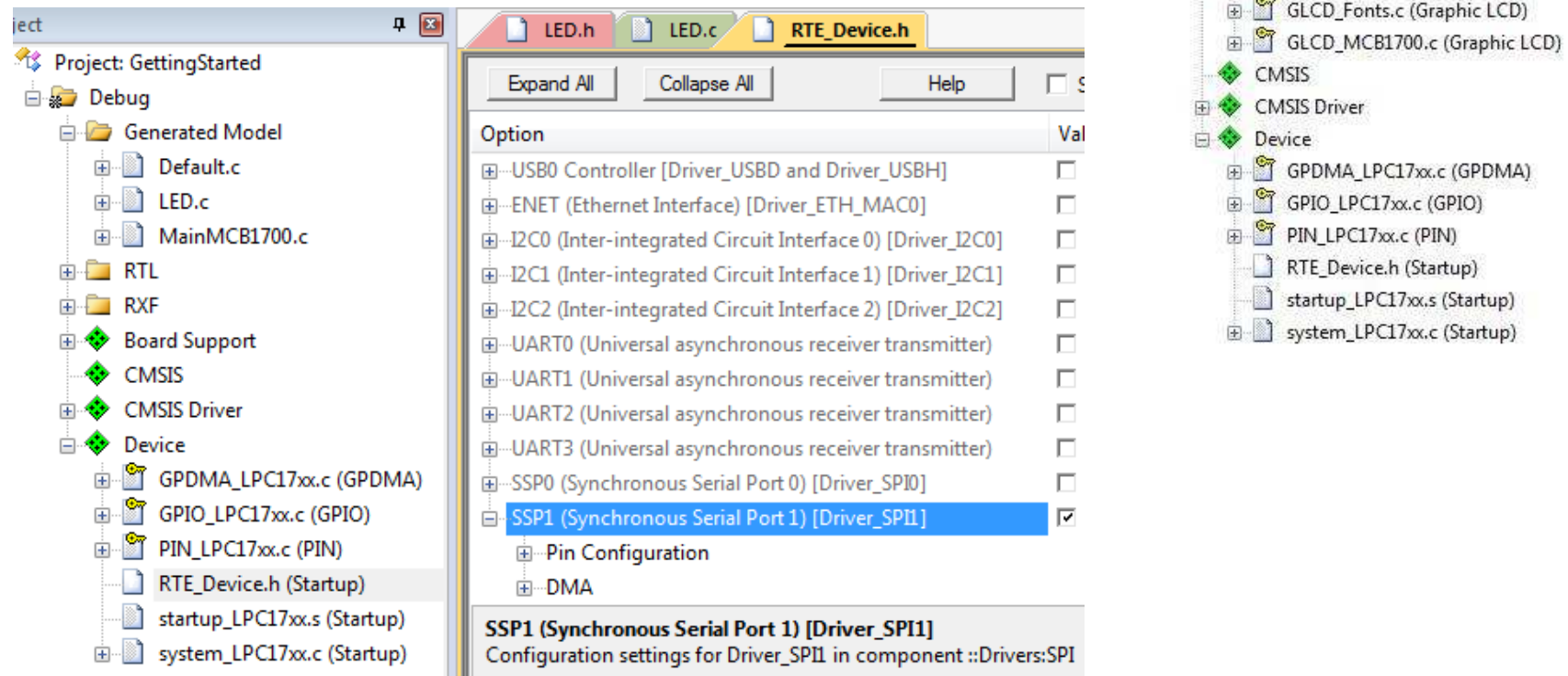
- Board Support – Graphic LCD
- CMSIS Driver – SPI – SSP
- Device – Select all



Exercise 5 (continued)

In Keil:

- You now see the Board Support and CMSIS in your project
- Choose Device – RTE_Device.h and select SSP1

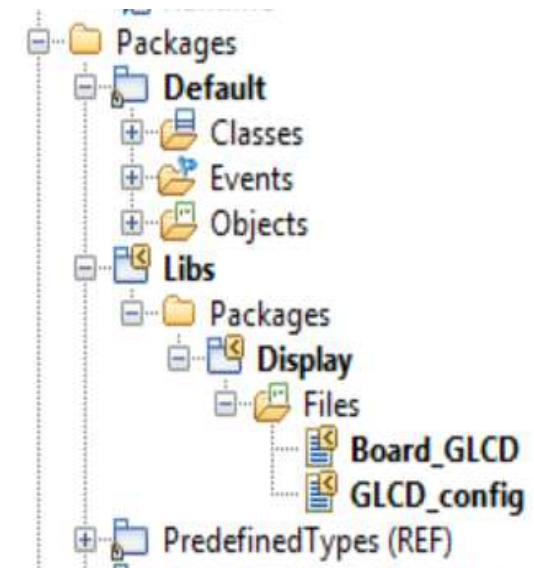


- Save the settings and close the project

Now you can use the GLCD commands in your Rhapsody Project

Exercise 5 (continued)

- In Rhapsody:
 - The display requires **additional files** which contain the display driver functions. To use them in the model do a right click on “Packages” and add a new package “Libs” and set its Stereotype to “External”.
 - Create in “Libs” another package “Display” and add the files “**Board_GLCD.h**” and “**GLCD_config.h**”The explorer should look like this:
- Double click on your MCB1700 component and select the **include path** as following:
C:\Keil_v5\ARM\PACK\Keil\MDK-Middleware\7.4.0\Board



Excercise 5 (continued)

- In Rhapsody:
Create a usage / extern C relation from Class Display to the GLCD libraries

The screenshot displays the Rhapsody IDE interface. On the left, the 'Display' class is shown with methods: `Display()`, `drawString(ledNr:int):void`, and `evSwitchLed(ledNr:int)`. It has two dashed arrows labeled «Usage» pointing to external files: `Libs::Display::Board_GLCD` and `Libs::Display::GLCD_config`. On the right, the 'Dependency : Board_GLCD in Display' dialog is open, showing the 'General' tab with 'Name: Board_GLCD', 'Stereotype: Usage in Predefined Types', and 'Depends On: ☒ Usage in Predefined Type'. Below this, the 'Properties' tab is visible. At the bottom left, the 'View Overriden' dropdown is set to 'CPP_CG', showing a 'Dependency' section with 'SpecificationEpilog' and 'SpecificationProlog'. The 'SpecificationProlog' is highlighted, and a 'Text Editor' window shows the generated code: `extern "C"`. On the right, the generated code in `Display.h` is shown:

```
extern "C"
{
//#]
#include <GLCD_config.h>
}
```

The generated Code in Display.h should look like:

```
extern "C"
{
//#]
#include <GLCD_config.h>
}
```

Exercise 5 (continued)

You can use the following commands for controlling the display:

```
extern GLCD_FONT GLCD_Font_16x24; //in case you don't want to
                                   // use the default string

GLCD_Initialize();
GLCD_SetBackgroundColor(GLCD_COLOR_WHITE);
GLCD_SetForegroundColor(GLCD_COLOR_BLUE);
GLCD_SetFont(&GLCD_Font_16x24);
GLCD_ClearScreen();
GLCD_DrawString(2,3,"Hello"); //line,column, text as char-Array
GLCD_Bargraph(unsigned int x, unsigned int y, unsigned int w,
               unsigned int h, unsigned int val);

/*Print a Value val as bargraph to line x and column y with the
pixel width w and high h*/
```

Annotation: in C `sprintf(str, "i is %i", i)` changes integer `i` to a C string `str` (char array) with the same content (header `stdio.h` necessary)

Exercise 6

- Model a periodic A/D conversion of the potentiometer voltage. The interval between two AD samples should be 10ms. Convert the AD values to a number between 0 to 100. Print the current value to the LCD as number and bargraph.

Exercise 7

- Design a complete LED supervision through the board class. The delay of the “moving-light” should be controlled through the potentiometer in the area 0 to 100.
- **Hint:** Consider again how to use the UML relations reasonably.



Exercises 4 – 7 have to be checked through a lab advisor. Please mail your projects and submit printouts of your implementations, methods and state charts to the advisors.

Exercise 8

- Use the “joystick” on the MCB1700 board to control the „moving light“ through user inputs. Model a LED supervision that allows the user to enable/disable, stop and restart the LED line by using the joystick. The joystick should be prompted by polling.

Joystick initialization:

```
LPC_GPIO1->FIODIR= ~( (1UL<<20) | (1UL<<23) |  
                      (1UL<<24) | (1UL<<25) | (1UL<<26));  
/* Port P1.20 and 1.23 to 1.26 is input (Joystick)*/
```

Identification:

```
position = (LPC_GPIO1->FIOPIN >> 20) & Joystick_Mask ; // last 7 Bits now are related  
                                                         //to Joystick
```

Bit 4 to 7 includes the information about the Joystick position:

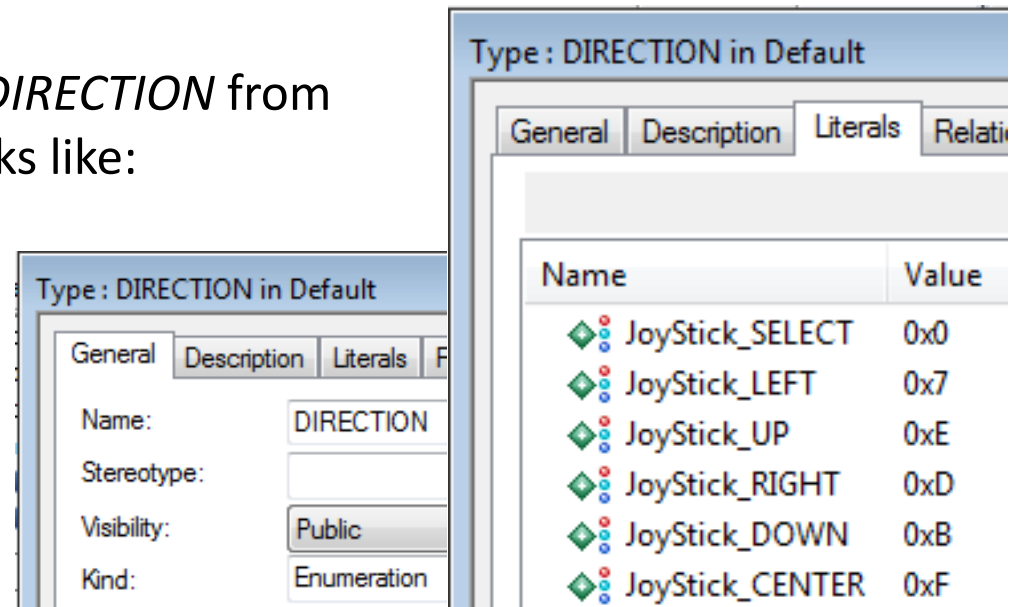
```
position = position >> 3;
```

Exercise 8 (continued)

- Joystick_Mask is a *constant*. A constant can be created as a *Type* through a right-click on the *Default* package and selecting *Add New* → *Type*. Call the new type *MASK* and write to *Declaration*:

```
#define Joystick_Mask 0x79
```

- We need another constant *DIRECTION* from kind *Enumeration* which looks like:



- The variable `position` can now be compared with the *DIRECTION* items.

Exercise 9

- Now use the *EINT0* Interrupt (triggered through button *INT0*) to start and stop the LED line.

EINT0 Interrupt initialization :

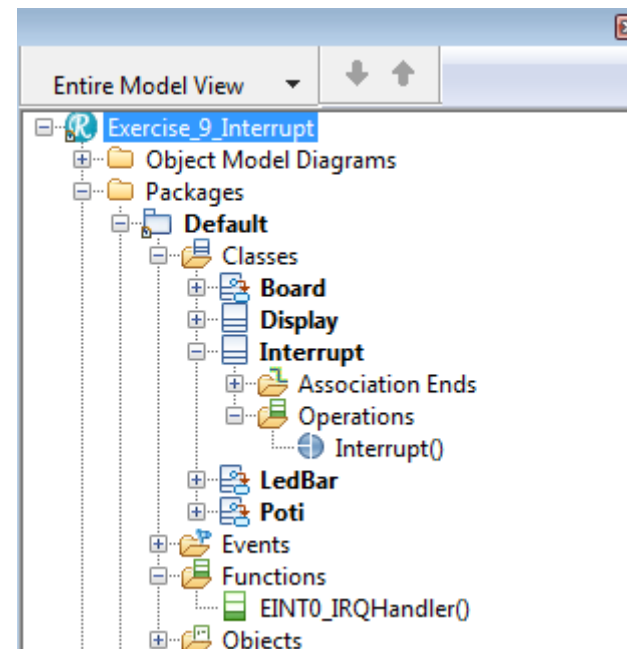
```
// P2.10 is function EINT0 for 01 (page 110)
LPC_PINCON->PINSEL4 |= (1<<20);
// Disable IRQ interrupt for INT0 in NVIC to configure
// EXTMODE (page 25) and EXTPOLAR (page 26)
NVIC_DisableIRQ(EINT0_IRQn);
// To ensure that EINT0 interrupt is cleared (page 24)
LPC_SC->EXTINT |= (1<<0);
// EINT0 is edge-sensitive (page 25)
LPC_SC->EXTMODE |= (1<<0);
// EINT0 is falling edge sensitive (page 26)
LPC_SC->EXTPOLAR &= ~(1<<0);
// Enable IRQ interrupt for INT0 in NVIC
NVIC_EnableIRQ(EINT0_IRQn);
```

Exercise 9 (continued)

- An operation `EINT0_IRQHandler()` is necessary. The core interrupt handler jumps to this function. There the *EINT0* Interrupt Service Routine (ISR) code takes place. In the ISR first clear the pending *EINT0* interrupt flag:
- The operation `EINT0_IRQHandler()` has to be an unqualified function.

Cleaning Interrupt flag:

```
// Clear EINT0 interrupt (page 24)  
LPC_SC->EXTINT |= (1<<0);
```



Exercise 10 Observer

- Use the observer pattern for your design.
- The poti value is used by the Ledbar to define the velocity of the running led.
- The poti value is also provided for the LCD display which shows the actual value.
- Both, display and Ledbar should always work with the actual poti value.

Hint: Rhapsody offers the template type `OMCollection<>Listelementtype>` This is a special list type with method `add` and `remove` to insert and delete elements. The list can be iterated by a `OMIterater<Listelementtype>` which returns a pointer to the actual listelement. Iteration is done with `++` operator.

Exercise 11 (optional) Ethernet

- We now want to use the **Ethernet** communications which is provided by our board. We need an additional platform model file and an idea how to model a class Ethernet. Therefore there is a starting project in `E:Ethernet_Aufgabe`.
Use this as start and deploy the Keil Project to
`E:Ethernet_Aufgabe\GettingStarted_Ethernet\GettingStarted`
- Now develop a model which uses two boards. Connect them with an Ethernet cable. Then model the following scenario:
If you **push the joystick up** on one board, the **other board** shows an **LED light**. If you push the joystick down, the light switches off again.
For the communication between the boards Ethernet is used.