WILLERT.
pioneers in embedded software engineering

Author	Clemens Maas Eike Roemer	**RXF for OORTX Debugging Guide** **Debug control flow and data in a simple example**
Document Version	1.0	
Document Status	Release	

WILLERT.
pioneers in embedded software engineering

Change History

Version	Date	Author	Description
1.0	May 2008	C. Maas	Initial Version

Table of Contents

1 Introduction 3

1.1 Pre-requisites 3

2 the Blinky Model 4

3 How to Debug 5

3.1 Overall flow 6

3.2 Timing 7

3.3 Event Processing 9

1 Introduction

The purpose of this document is to help you understand where things go wrong, if the Blinky example used in the Getting Started Guide [1] does not work. The Embedded UML Studio is a very complex product, and you could be struggling caused by an erroneous Bridge implementation, a malfunctioning heartbeat or a failure in the toolchain you are using.

Using this Guide, you should be able to pinpoint the problem you are faced with. It is not meant as extensive "How To" debug an embedded system or the Framework by Willert Software Tools. This Guide merely helps you to debug a simple application and understand which control flow and data are to be expected.

1.1 Pre-requisites

Before you can debug the model, there are a few pre-requisites:

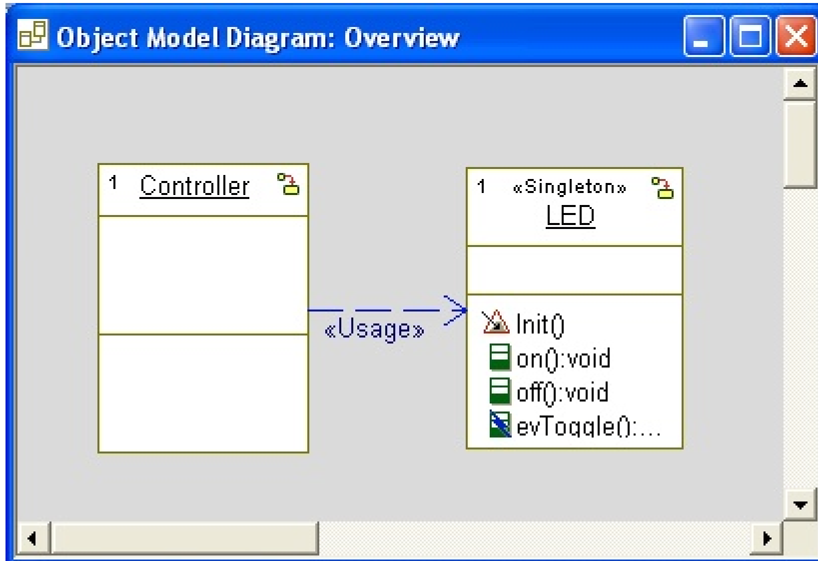
- knowledge of the target platform and your toolchain.
- the C code must be generated from your Rhapsody model without warnings or errors.
  This proofs that the installation of your product by Willert Software Tools is in order.
- the RXF libraries must be build, see Getting Started [1].
  Disable any optimizations for the compiler!
- the generated C code must be deployed and you must be able to build the corresponding IDE project without errors. Again, disable any optimizations for the compiler!

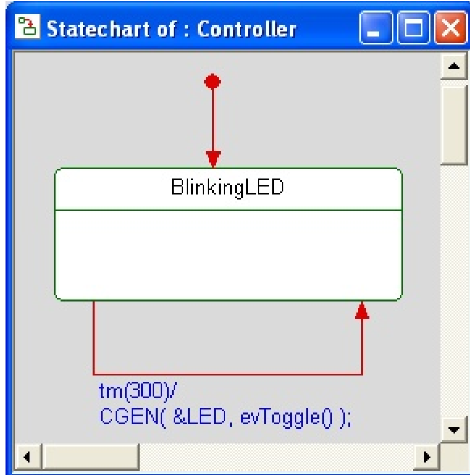
  The linker/locator may not throw any warnings when building the application in your IDE.

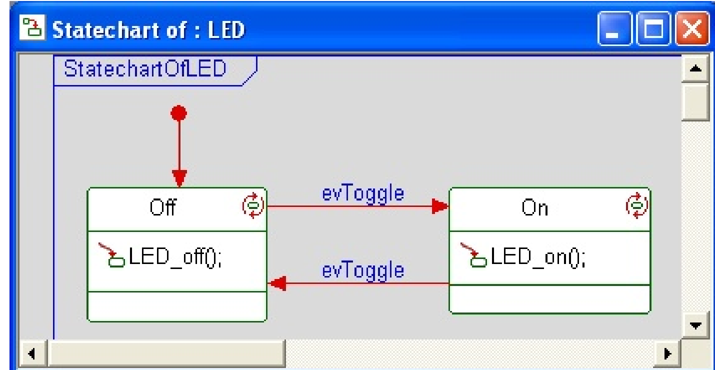
- debug information inserted by the compiler and a linker map file, generated by your linker are mandatory.

2 the Blinky Model

The Rhapsody Blinky model which is part of your release, consists of two objects `Controller` and `LED`:







The Rhapsody generated code from the `Controller` object results in a timer event, which is sent to the LED every 300 msec. This enables you to verify the heartbeat and proper handling of a timer event.

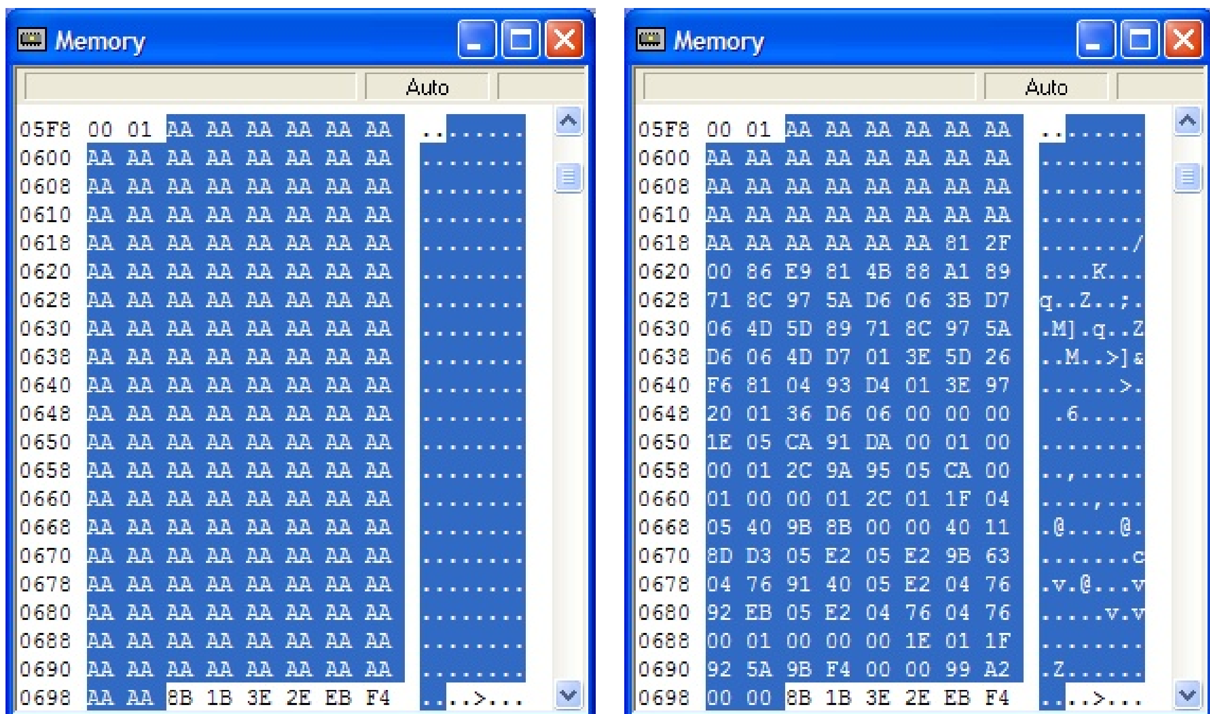
The generated code of the `LED` object sends an event to itself to toggle between the states off and on.

3 How to Debug

After you have generated the C code from the model, deployed it and build it successfully in your IDE, you can use a simulator or debugger to verify its behavior. We will do so by setting breakpoints and verify these being hit and some data, and we will do this at several levels.

When debugging on hardware, most environments allow for 2 simultaneous breakpoints only, and if you are performing single steps, one will be used by your debugger - leaving you a single breakpoint only.
Therefore, we have taken this limitation in account and use a single breakpoint when single stepping.

You want to exclude problems with respect to the stack size you configured in your IDE or startup file.
For this, fill the stack with a value like `0xAA` and inspect regularly between breakpoints if the stacksize is sufficient by inspecting how much of this `0xAA` filled area is overwritten with calls and local data.
You can find the exact location of the stack in your map file.



When you inspect the stack via a memory window in your simulator or debugger when hitting breakpoints, it will show that for some part `0xAA` is overwritten by stack frames including local data. There must still be an area with `0xAA` left!

In the example above, the stack is allocated from `0x5AA` up until `0x699`. After a while the stack has been used until 0x61E.

3.1 Overall flow

First we want to check if the model functions.

- set a breakpoint in `RiCErrorHandler_error()` in `RiCErrorHandler.c` if you can spare a breakpoint. Hitting 'Halt' when nothing seems to happen in your simulator or debugger may not always lead you to a meaningful code area; it could be an Interrupt Service Routine - even when you ended in the error handler.

  See the online Help documentation [2] for details on your error code, section *Usage | Error handler*. When you inspect the call stack you might understand why you ended in the error handler, but if not please follow the instructions in the following paragraphs.

- set a breakpoint in `WSTTarget.c` in the function `WSTTarget_Init()`.

  > Please make sure you are in the proper code section enclosed between `#ifdef WST_COMPILER_xxx` and `#endif`. The appropiate constant is set in `WSTProduct.h`

  Set a second breakpoint in `LED.c` in the function `LED_Init()`.

  - when you hit the breakpoint within `WSTTarget_Init(),` you can inspect if the initialization of your hardware looks OK. This is a breakpoint that must be hit, and you may remove the breakpoint when hit.
  - if the breakpoint in `LED_Init()` is hit, this proves that the RXF is started and at least the very first housekeeping seems OK. You may remove the breakpoint when hit, and set a breakpoint at the function `LED_on()` in `LED.c`.
  - if this breakpoint in `LED_on()` is hit, set a breakpoint at the function `LED_off()` in `LED.c`. If the breakpoint in `LED_off()` is also hit, `LED_on()` should be hit next, etc.

If the overall flow is working, than all you need to do is verify the exact timing - apart from the stack usage.

3.2 Timing

There are two issues we can check:

- if the `tm(300)` in the Rhapsody model matches the timing of the toggle between `LED_on()` and `LED_off()` Even if the overall flow like described in the previous paragraph does not work, we can verify if the heartbeat which the Realtime eXecution Framework needs is not a single shot, and that its frequency matches the constant `RIC_MS_PER_TICK` in `WSTTarget.h` which is used to translate the timer ticks into miliseconds as used by the RXF.

- if a timer event is properly generated and consumed.

3.2.1 Heartbeat

The RXF is usually triggered at a regular interval via a timer Interrupt; in that case the Interrupt Service Routine must call the function `WSTRTOS_incrementRxfTicks()` in `WSTRTOS.c`
Alternatively, you may call this function from an RTOS task or from an SPS or IEC1131 environment. In any case, the function `WSTRTOS_incrementRxfTicks()` must be called at a regular interval:

- remove any breakpoints and set a breakpoint in `WSTRTOS_incrementRxfTicks()` in `WSTRTOS.c` If the overall flow like described at page 5 does not work you may need to reset the target and even flash the application again when debugging on hardware, to make a fresh start.

> Please make sure that you use `WSTRTOS_incrementRxfTicks()` within the proper code section enclosed between `#ifdef WST_COMPILER_xxx` plus `#endif` and the proper `#ifdef WST_TARGET_xxx` plus corresponding `#endif`. The constants are set in `WSTProduct.h`

- each time the function `WSTRTOS_incrementRxfTicks()` is called, an internal counter in the RXF is incremented:

  - the constant `RIC_MS_PER_TICK` in `WSTTarget.h` should match the milliseconds between the ticks or calls to the function `WSTRTOS_incrementRxfTicks()`.

  - if the function `WSTRTOS_incrementRxfTicks()` is only called once, verify why.

    > You may need to search for the call in all source files, because the call stack window in your simulator or debugger will not be updated when this function is called from within an ISR.

3.2.2 Timer event

When a timer goes off, like specified after the `tm(300)` in the Rhapsody model, an event is generated. The difference with an 'ordinary' event and a timer event, is that the event data itself is allocated statically; the timer events do not use parameters and are always equal in size, so the maximum number of timer events which must be supported by the RXF can be configured at compile time.

If the overall flow like described at page 5 does not work, we must verify the proper handling of a timer event:

- first we verify the transformation of a timer event. Remove any breakpoints and set a breakpoint in the function `rootState_entDef()` in `Controller.c`, at the call to `RiCTask_schedTm()`:
    - o when hit, remove this breakpoint and step into `RiCTask_schedTm()` in the file `RiCTask.c` and step into the call to `RiCTimerManager_add()` in `RiCTimerManager.c`
    - o the local variable `aTimeout` must get a value or address which sits in the array `RiCTimerManager_itsTimeout[]` which address you must lookup in the linker map. (This array is declared in `RiCDimension.c).` Step over until you see where an address is assigned to `aTimeout`. Remember this address; it is actually the first element in the array.
    - o the timer event will be transforned into an 'ordinary' event: set a breakpoint in `RiCTask.c` in the function `RiCTask_Timeouts()` at the call to `RiCEvtQueue_put()`.
      Verify the value of `aTimeout` which must be the address remembered and if you can step over this function, i.e. if this function returns and remove the breakpoint.
- now we must verify the consuming of the timer event. Set a breakpoint in `RiCTaskExecute.c` in `RiCTask_execute()` at the call to `RiCTask_Timeouts()`:
    - o remove the breakpoint when hit and step into this function in `RiCTask.c`
    - o set a breakpoint at the call to `RiCTimerManager_getExpiredTimeout().`
      When hit, remove the breakpoint and do a step over.
    - o set a breakpoint at the call `RiCEvtQueue_put()` which must have as argument an address which is an element of `RiCTimerManager_itsTimeout[].` Remove the breakpoint when hit and you verified the address with `RiCTimerManager_itsTimeout[]` in your linker map.

3.3 Event Processing

An 'ordinary' event is allocated dynamically, but actually for event objects we use the static buffer pools (small-, medium- and large pool). The size of an event may vary depending on its arguments, if any, so the RXF uses a static array for pointers to the events and the buffer pools for storage of the events themselves.

We want to verify the producing of an event:

- remove any breakpoints, and set a breakpoint in `Controller.c` in the function `rootState_dispatchEvent()` at the call to the `CGEN()` macro.
- when hit, remove the breakpoint and single step. You will reach `RiC_Create_evToggle()` in the file `ExamplePkg.c` and step into the macro `RIC_MEMORY_ALLOCATOR_GET()`:
    - you will end up in `RiCAllocator_getMemory()` in the file `RiCDimAllocator.c` where you must verify its argument `requestedSize` which must be in the range of 8 to 16 bytes - the actual number depends on the platform and compiler bridge being used.
    - now step further; the requested size is smaller than `RIC_SMALL_BUFFER_SIZE` so you should reach the call to `RiCMemoryPool_getMemory()` with `RiCAllocator.itsSmallPool` as argument.
    - step over this function `RiCMemoryPool_getMemory()`. The address returned must be a block within `RiCSmallMemoryBlocks[]` which you must verify in the linker map. (Actually the last block will be returned, so do not panic).
      Remember this address.

Now we must verify the consuming of this event:

- remove any breakpoints, and set a breakpoint in `RiCTaskExecute.c` in the function `RiCTask_execute()` at the call to the function `RiCEvtQueue_get()`.
- remove the breakpoint and step over this function. The return value must be the address of the event which was stored in the small buffer pool.

> If you needed to reset your target, this breakpoint is first reached finding the timer event.
> That address is an address in `RiCTimerManager_itsTimeout[]`.
> The second hit will show you the remembered address which lies in `RiCSmallMemoryBlocks[]`.

- set a breakpoint in `RiCDimAllocator.c` in the function `RiCAllocator_returnMemory()`. The argument `memory` must be the same address and `RiCMemoryPool_returnMemory()` should be called to return a block to the small buffer pool where the event was allocated previously.

> Note that there are two implementations of `RiCAllocator_returnMemory()`; one used with the Telelogic Container Classes (standard), one for special cases (`WST_PMF_CONTAINERCLASSES` set). You must use the `RiCAllocator_returnMemory()` with a single argument if the constant `WST_PMF_CONTAINERCLASSES` is not set in `WSTProduct.h`

References

[1] RXFGettingStarted.pdf – RXF Getting Started Guide.

[2] Rhapsody\Share\WST_RXF_V5\<your product>\Doc\Help\index.htm
Online documentation for your product.