

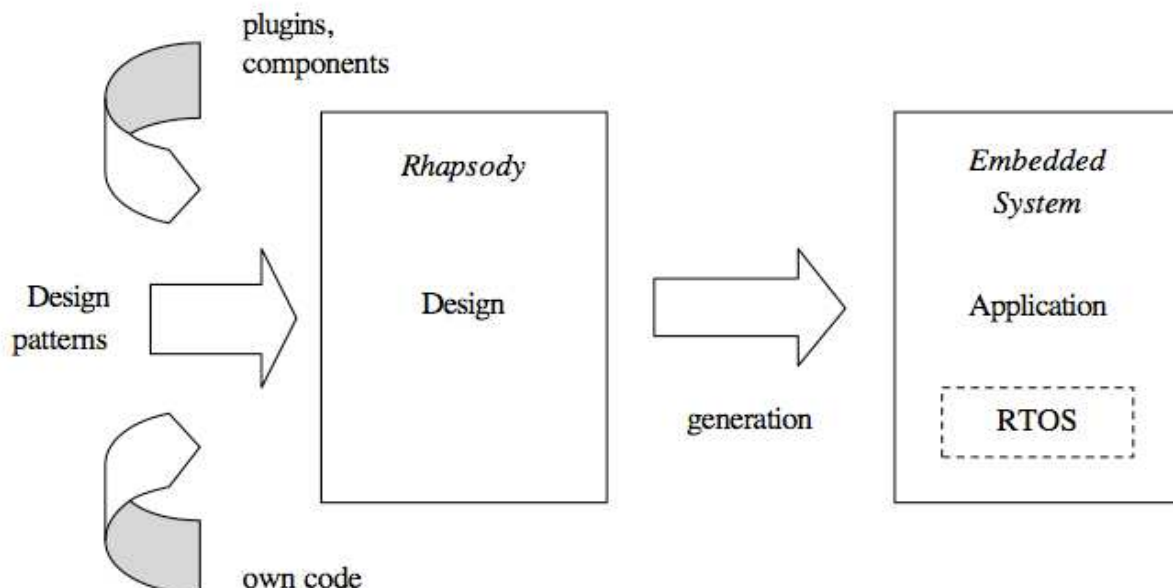
# UML and Embedded

This article is not about UML - there are plenty books out there which explain UML and how to use it. But you'd be surprised about the combination UML and embedded systems.

There are tools out there which let you design an application for an embedded system, which generate the complete framework and all logic. You can insert your specific code, and create components which are reusable such as drivers for some hardware or design patterns. One of these tools is Rhapsody by iLogix Inc, which comes in several flavors regarding the language. Using it for embedded applications, we will focus on Rhapsody for C.

Although you will have massive legacy code, it is a good idea to start with a new design from scratch for whatever you are trying to create. The reason is not that your new application should be Object Oriented to the maximum, but after you have realized an application on a proper design, it is easy to fix problems when referring to the design.

The UML approach brings support for both design as well as analysis, including analysis of a running system. We will come back to this later. First we will discuss how to design and code your system.



You create your design with Rhapsody. You bring in design patterns, as to translate common problems into a solution. Apart from the design you do yourselves, you will insert your own C code. You can compare this as the body of a function: the skeleton of the function is created by Rhapsody including the proper arguments etc.

Although we are not programming in C++ but in C, Rhapsody will translate *classes* into structures including attributes and methods. Some aspects of C++ are not available as overloading and inheritance, but the advantage is a remarkable small footprint and speed. As the code is being generated, it is easy to call some objects methods, while the UML design will support you in attacking a problem and support you with its many views.

Apart from inserting your code like the body of a function, you can use plugins in the form of *components* and *units*. A unit is a collection of one or more objects which can be used as a plugin. Typical examples are a driver for some hardware and previously developed pieces of code ready for use. A component is an abstraction at a higher level, and usually is the realization of a fair piece of your embedded application.

During your work as a designer or inserting code, Rhapsody offers a window called “Active Code View” which shows the result of your actions immediately. The reason is twofold: it offers you an in-depth view on what is going on “under the hood” as well as it shows how your code hooks in. At regular intervals, you will press the “Generate, Make and Run” button, to validate the code you inserted and verify the behavior of your design.

### With or without RTOS

The result of the code generation (and inserting your code snippets) is an application which can run on a target, with or without an RTOS. The running environment can be selected within Rhapsody and offers you various possibilities.

The standard Rhapsody in C by iLogix Inc is perfect for creating and maintaining an application for 32 bit targets, including support for popular RTOSses. The interface between your application and an RTOS is called a *Bridge*. Willert Software Tools GmbH has adapted existing bridges for smaller targets so you can use Rhapsody in C for 16 and even 8 bit targets, supporting popular RTOSses like emboss by Seggert and CMX-RTX by CMX Systems. For those environments where there is no need for an RTOS or it will introduce too much overhead in functionality or size, a so called Interrupt Driven Framework has been developed.

There is a direct relation between your design and the number of tasks in an operating system. A class thus the derived object can be defined as *active*. An active object will run in its own task; Rhapsody generates automatically the proper RTOS initialization. For OSEK compliant RTOSses, an OIL file which dictates the RTOS configuration can be used as input for Rhapsody, preventing you from creating a design which would conflict with the configuration of the OSEK RTOS you need to use.

The operating environment is characterized by several design specifics: each task has a event queue which is used to interact between tasks and objects running in that task. An object may send an asynchronous event to another object, which will handle the request when the event is dispatched. A synchronous event acts like a conventional function call; the addressed object handles the request immediately.

Apart from events there is of course explicit support for timers. Depending on the compiler used, one can declare an object to act as Interrupt Service Routine, for example to send an event to an object which must handle this interrupt.

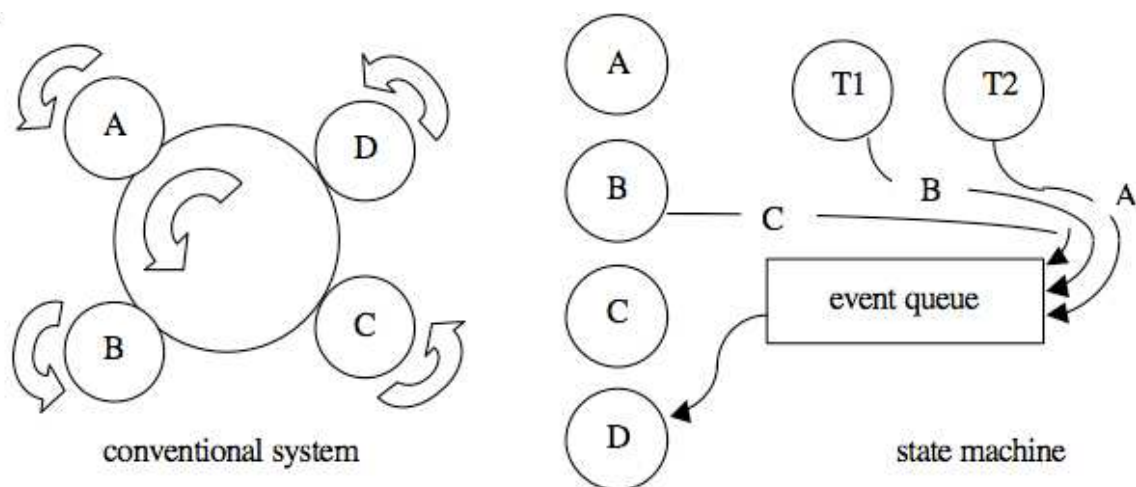
## Conservative versus UML approach

There is both a similarity as well as a large difference between “conventionally” setup of an application and a UML approach.

For memory management sake, one can still use static allocated data structures and objects, preventing calls to malloc() on the fly. The advantage of a “static” approach is that one prevents inefficient use of resources and last but not least the effect of a spurious interrupt in the middle of a malloc(). The downside is that one must pay extra attention to dimensioning the system. This requires careful analysis of actual resources used, comparable with a high water mark on the heap.

The radical difference between a “conventional” setup application and this UML approach is the way the system operates. Instead of some endless loop which acts as main flow, while calling functional chunks of code consecutive even when handling interrupts, one now uses an event driven state machine.

Although at a first glance one may end up with a system which when driven to its extends may lose interrupts or fail to handle something, the opposite is true: careful designing will result in handling what really needs handling and for example skip less important things like updating a display. In the end, the “conventional” system will have collapsed way before.



The conventional system calls functions A, B, C and D in a large loop. In the UML designed system, objects have states and react on events, sent by other objects or timers.

It does take a mind switch to design a system as a state machine, but this comes with a great advantage. One can easily depict from realtime results and the underlying design what is going wrong or where exactly the boundaries are of the properly working system.

## Testing

UML offers a great feature which is being exploit by Rhapsody: the animated sequence diagram. In a sequence diagram you analyse the control flow between objects, using events and timers in a graphical design. When actually running the application, one can see the control flow in an animated sequence diagram. This animation shows you the

exact interaction and flow of events and how the objects interact.

The animated sequence diagram can be saved for further analysis or you can compare it with older versions. And yes, Rhapsody supports the use of a source control system. You can retrieve parts of your design from a source control system; by default packages and units are read-only and clearly marked as such, while you can check packages and units out from the source control system being used.

The interface towards a source control system and use of makefiles make the tool ideally suitable for using regression tests and co-operate use. As a bonus, iLogix Inc has added a test generator which can run tests based on a script while the result is being logged in a trace file. Obviously, the test scripts and trace output can be maintained in your source control system, for optimal use of regression tests and continuous verification.