AVIX Real Time Operating System

AVIX-RT Tutorial Series
Tutorial 1

Why use a Real Time Operating System

AVIX-RT

AVIX
Advanced
Virtual
Integrated
e**X**ecutive

AVIX-RT
Maïsveld 84
5236 VC ‘s-Hertogenbosch
The Netherlands

phone +31(0)6 15 28 51 77
e-mail info@avix-rt.com
www www.avix-rt.com

Table of Contents

1 Introduction

Embedded system developers must find solutions for many problems like ‘how to implement correct system timing’, ‘how to deal with external events, often through the help of interrupts’, ‘how to test using complex test scenarios’ and so on.

Even the smallest embedded systems are becoming more complex because of the use of complicated communication interfaces like USB or TCP/IP. The ‘lifetime’ of embedded systems can be quite long, often many years. During this lifetime the embedded system is enhanced with new functionality, again and again challenging the developer’s skills to keep the system working.

Many embedded systems are developed without applying an RTOS (Real Time Operating System). Different reasons exist for doing so. ‘An RTOS is complex’, ‘An RTOS introduces too much overhead, both in timing and memory consumption’, ‘An RTOS is expensive’ and many more.

Whether or not these reasons are valid, they bypass many of the advantages an RTOS can offer. Instead of using the features an RTOS offers to ease systems development, all relevant aspects are custom developed, over and over again.

This article is the first in a series of articles with the goal of explaining how an RTOS can be beneficial to embedded systems development. A number of aspects relevant for embedded systems development will be presented where a comparison will be made as how these aspects influence development both with and without applying an RTOS.

This first article focuses on the effect an RTOS can have on the structure of the software and thereby have a positive influence on testability, reusability, extendibility and modularity of the system.

Subsequent articles will cover how an RTOS can help with:

- Timing aspects to let the system fulfil its timing requirements
- Interrupt management to deal with external events
- Communication between the system modules
- Resource management to deal with memory and processing power

2 The software structure of an embedded system

This chapter presents a basic embedded system not using an RTOS. The system evolves through three subsequent generations where in every generation, functionality is added.

Like every software system, an embedded system needs a certain structure in order to streamline development and keep the system as a whole comprehensible. Quite often functional aspects present in the system drive the division of the systems software in separate modules.

Generation one: The first generation of the embedded system has basic functionality. An analogue value is converted and some processing is done to the values read from the ADC. This is done at a frequency of 1000Hz. The resulting design is shown in Figure 1. It contains three software modules, one for converting the analogue values, one to do the processing on these values and a third to activate the other two. The result is manageable and easy to construct.

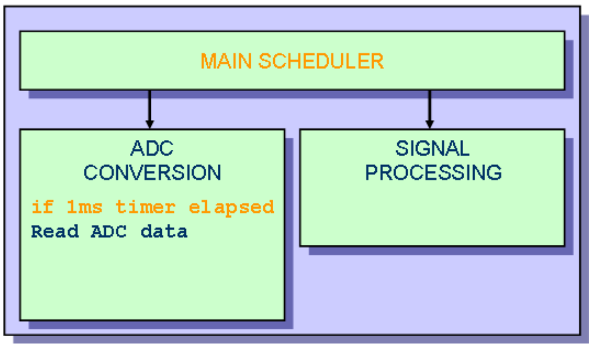

Figure 1: Basic system generation 1

Generation two: Systems tend to evolve. Functionality is added, increasing their complexity. So is the case with this system. A second generation is created with more elaborate processing. As it appears, processing time becomes so long that the cycle of one millisecond with which the analogue values must be read can no longer be met. On average, processing is still below one millisecond but sometimes processing takes longer. To solve this problem, the Signal Processing module is split in three parts. The processing time of each of these parts does not exceed one millisecond. A state machine is added to the Signal Processing module, responsible for successive activation of each of the three sub modules when the Signal Processing module is activated. To guarantee no samples are lost, a second extension is required in the form of a buffer. Here the analogue values are stored until they are processed. The resulting system is shown in Figure 2. Both the state machine and the buffer mechanism have nothing to do with the main goal of the system which actually is unchanged.

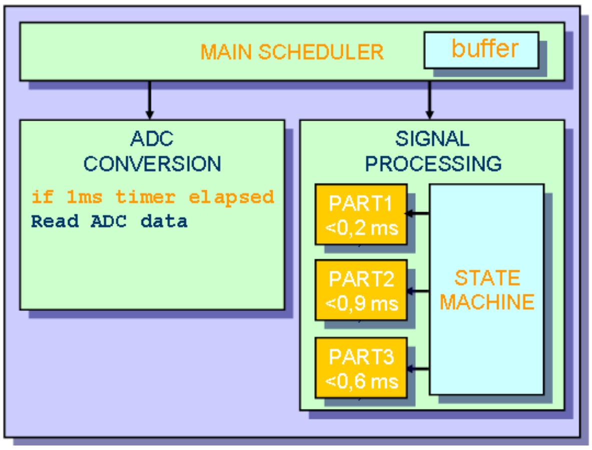

Figure 2: Basic system generation 2

Generation three: A new requirement leads to a third version of the system It is extended with an LCD display. A new software module is added, responsible for controlling this display.

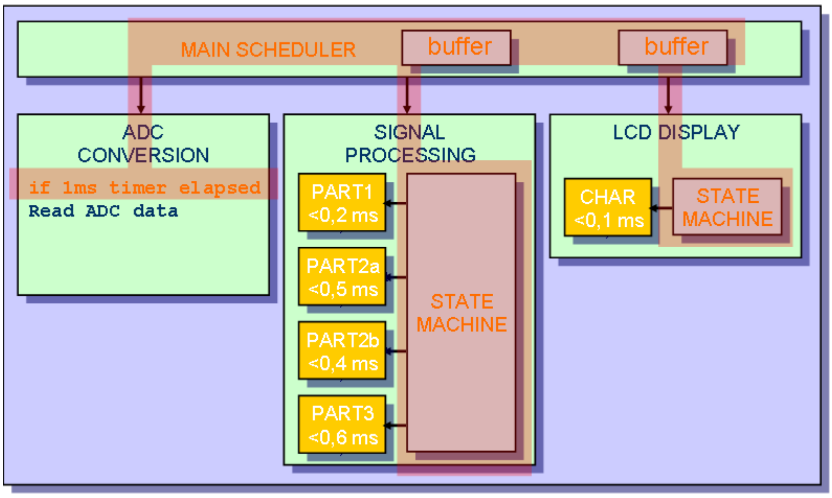

Figure 3: Basic system generation 3

Problem with LCDs is they need a minimum time for every character to be written. This implies that the displayed data cannot be written at once. This would again violate the 'one millisecond' requirement with which the analogue values must be read. Therefore the LCD module is developed with a local state machine taking care of writing one character at a time. To store all characters until they are processed, a second buffer is added. Adding the LCD display does introduce a new problem. One of the three sub modules of the Signal Processing module takes a maximum of 0.9 milliseconds. Adding the LCD processing time to this figure, again the total system timing requirement is no longer met. To solve this, the Signal Processing module is revised and split even further to contain a total of four sub modules now. The resulting system is shown in Figure 3.

The shaded area in Figure 3 contains the modules added in order for the system as a whole to meet its timing requirements. As shown this makes up quite a substantial part of the system.

Still no RTOS is used or needed but what is happening here? Which problems are introduced?

- More effort is needed to create all additional software
- Correct timing is accomplished in a trial and error fashion
- The modules are not autonomous. Adding a module (LCD) breaks the timing of other modules
- It is virtually impossible to reuse modules for other systems due to the strict timing relationships that exist between them
- The development process as a whole is cumbersome. During testing and bug fixing, processing times change, and again potentially break the systems timing
- Even the simplest change, like a new version of the compiler, might break the system due to different code sequences being generated with different timing.

Without doubt the system will work correctly. The question one may ask however is whether this approach, and the negative side effects introduced by it is acceptable and, even more important, how this all can be prevented.

One of the possible answers to this question is by applying an RTOS.

3 The effect of an RTOS on the software structure

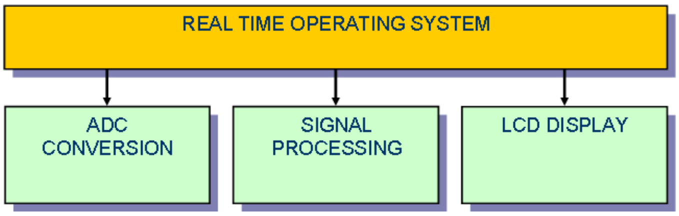

Figure 4: Basic system using an RTOS

If the system uses an RTOS, all supporting functionality, not directly related to the systems functional behaviour, would not have to be custom created. Instead it would be provided by the RTOS. The individual modules could be created as if they were stand alone autonomous modules and the relation between them is taken care of by features and mechanisms offered by the RTOS. The resulting structure of the sample system when applying an RTOS is shown in Figure 4

The timing requirement of the overall system remains the same. Modules taking too much processing time still need to be split in multiple parts in order for another module to be given precedence. With an RTOS however, it is the RTOS that stops one module in favour of another (pre-emption), should the second have to be given precedence. *The largest positive effect on the systems structure is that each individual module is written as an autonomous piece of software. No custom state machines are needed to 'split' a modules processing in multiple parts. This enhances the readability, testability and not in the last place reusability of the individual software module. It will lead to faster development, less errors and fewer dependencies between the different modules.*

Also the buffering mentioned in the 'non-RTOS' scenario is still needed. Buffering mechanisms are now however offered by the RTOS and can just be used out of the box instead of being custom developed.

Not detailed in the previous scenario is how the modules communicate with each other. Here too functionality is offered by the RTOS that otherwise would have to be custom developed. Finally the RTOS will have a positive influence on the available processing power of the system by not using polling and active wait mechanisms. These last two aspects are detailed in forthcoming articles.

4 Conclusion

I have suggested that developing an embedded system without an RTOS requires effort to be spent on software components not directly related to the functional behaviour of the system. Not only does this imply that development takes longer. Also the interdependencies between the individual modules make the system as a whole more complex and have a negative influence of future expandability of the system and reusability of the individual components. Even for the basic system illustrated here, using an RTOS has a positive influence on the software structure thereby streamlining development and maintenance. Also by delegating system timing to the RTOS, correct system behaviour is much less dependant on all kinds of variations like:

- Adding/changing code
- Using a microcontroller with a different speed
- Changing compiler optimization levels
- Etc.

The system as a whole becomes more robust, easier to develop and easier to test. Of course, when using an RTOS, there is no such thing as a free lunch. Using an RTOS requires some familiarisation to learn how to deploy the RTOS and its mechanisms, such that the highest possible benefit is gained. As stated in the introduction, a number of topics related to this will be the subject of forthcoming articles appearing as part of this tutorial series.

About the author:
Leon van Snippenberg is founder and owner of AVIX-RT (www.avix-rt.com). AVIX-RT is a company developing and marketing an RTOS specifically developed for Microchip 16 and 32 bit microcontrollers belonging to the PIC24F, PIC24H, PIC30F, PIC33F and PIC32 families.