

Real-Time Architecture Design - Juggling with Time Newsletter No 21

WILLERT.



The main loop, together with interrupt service routines, keeps being used as central runtime control in the architectural design of embedded software.

According to the “2008 Embedded Market Study“, however, the increase in code complexity is one of the main challenges in modern software engineering. In this survey, 81% of all developers not using an RTOS state that they don’t see any reason for using an RTOS.

Do these software developers understand that the architectural design is one of the key instruments for mastering complexity? I’m afraid they don’t, and this is why we highlight this issue in a newsletter.

Today, architectural design is one of the most powerful mechanisms of software engineering. It is the basis for enhanced software quality in terms of reusability, understandability, maintainability, robustness, testability, etc. How come?

To answer this question, let us first of all look into the topic ‘complexity’. What exactly does complexity mean?

Latent Growth of Complexity

Looking at today’s software systems with respect to the functionality to be implemented, we will come across several aspects to be considered, in the following referred to as levels. Basically, every software system comprises these 4 levels:

1. Logical behavior
2. Timing behavior
3. Data flow
4. Priority between concurrencies

These four levels are to be found in all software systems. A change occurring in one of the levels can affect any other area of the software through the other levels. Let’s take a look at the main loop, for example. If the logical behavior of a code sequence within the main loop is changed, the timing behavior of all code sequences will be changed. The resulting time shift, no matter how insignificant, can sometimes cause the minimum reaction time of another sequence to be exceeded.

A system is considered to be complex if it comprises different levels and if changes in one of the subareas

of the system can affect completely different areas through the other levels.

In real systems, further levels have to be considered in case of changes, expansion or reuse.

The software might know different operating states, for example, a service and diagnostic mode or software update mode running in parallel to the actual operating mode.

Many devices are available in different variants, such as a low-cost and a high-priced variant with enhanced comfort. If possible, both variants should be supported by one software version.

Our software might also have to be compatible with older device versions. For example, if any hardware components were changed in new devices but previous hardware is still used in the old devices, the software shall work with both device generations.

If we continued our analysis, we would come across even more additional levels.

Looking at a single line of source code in our application, all levels must basically be taken into consideration when a change occurs. For example, conditional compilation based on the compiler directives `#ifdef`, `#ifndef` and `#else` is often used to be able to represent individual levels in the source code.

In practice, there are 7-12 levels of this type today that have to be considered, which accounts for most

of the complexity. The standard solution to address complexity is a proven trick - "divide and conquer". By splitting up a complex system into several less complex units, it is much easier to manage these units. The split-up of software into individual components is also referred to as software architecture design.

The most commonly used mechanism to this end is a split-up into modules and subprograms. Modules provide for a logical structure, but in many cases the contained subprograms are still processed centrally in a main() routine. This type of architectural design is not very useful with regard to levels.

If changes have an impact, through the different levels, from one module or subprogram to other subprograms, the "divide and conquer" trick won't work. For this reason, developers have introduced the so-called encapsulation method for object-oriented programming.

This term has mostly been used in the context of programming data-centric software (databases, booking systems, etc.), mainly focusing on data consistency and encapsulation between individual system units at data flow level.

However, most technical systems are based on runtime oriented software where common encapsulation methods are not really efficient. However, that doesn't mean that encapsulation is not possible in this context at all.

Keep in mind that the following developments lead to an increase in complexity:

1. Higher complexity of the actual logical behavior.
2. Higher amount of concurrencies in our systems (displays, bus systems, safety queries etc. have to be processed in parallel to the actual task).
3. More and more different aspects (also referred to as layers) with which our software has to comply.

In most cases, developers have no impact on items 1 and 2. As far as item 3 is concerned, however, complexity can be addressed through the software architecture design. Of course, the most efficient way is to avoid levels. Take a look at these 2 examples:

1. Timing level:

In case our software is designed as runtime architecture based on a main loop, the timing level can be neglected if the CPU is fast enough to ensure correct response times. If your corporate management can be convinced to spend some more money on the hardware architecture, the timing requirements can be neglected.

2. Device variants:

Today, it is often possible to provide different devices with the same hardware basis. The most expensive and the most cost-efficient device variants then have

identical hardware, which eliminates differences in the software systems.

Such options should always be discussed when considering the purchase of new devices. This article, however, also describes how an optimized software architecture design provides for an efficient encapsulation of individual levels.

Encapsulation in Runtime Oriented Systems

Software complexity can be reduced by creating the basic requirements of the architecture such that one of the levels becomes negligible. In this context, it is useful to review the basic requirements of embedded real-time systems regarding the timing level.

Ideally, our system responds to events from the outside world at any time without time delay. This ideal situation cannot be achieved, of course, because our CPU reacts according to a time pattern (the theoretically smallest time pattern would be clock frequency). What we can achieve, however, is a time pattern that is small enough for the response times of our application to be adequately met.

An example: The refresh rate of a TV set is 50 images per second. At this speed, it is possible to trick our eyes and make films look like flowing movements rather than a sequence of individual images. Thus, our microcontroller can carry out monitor control based on this frequency.

Minimum response times for horizontal and vertical deflection can be calculated from this frequency. Different processing steps have to be executed during these times. A very simple architectural design would consist of a main loop which is executed cyclically at the required minimum cycle time and carries out the required signal processing within one cycle.

The prerequisite for such a simple design is that all required signals are applied to the CPU periphery continuously (at any time). The signals are queried, analyzed and processed cyclically.

There is only one basic precondition for this runtime design: The processing speed of the CPU has to be high enough to ensure the cycle time of the main loop. In this case, the threshold frequency of the system has to be determined once for the runtime architecture design. In case of system changes, it would only be necessary to measure the cycle time of the main loop. The timing level can be neglected as long as the threshold frequency of the system is not under-run.

If the threshold frequency is exceeded, one could simply use a higher-performance CPU (from a software engineering point of view).

However, this is exactly where we sooner or later meet the limits of our system in reality. A faster CPU could call for a different microcontroller family which would necessitate a hardware redesign.

Such measures require a lot of time and effort. Therefore, it is sometimes more efficient to carry out a timing analysis of the software architecture, with the result that some software operations do not have to be executed at the ideal cycle time within the main loop and require a much shorter cycle time instead.

This opens up potential for an optimization of the timing behavior, and a software change eliminates the need for a laborious hardware redesign.

A consistent timing analysis of the system always results in different tasks at different cycle times (e.g. different intermittencies between line frequency and refresh rate) as well as event-driven tasks without cycle time (e.g. TV channel switching or volume control). Thus, by modifying the architectural design such that tasks are carried out only when needed, relating to their own threshold frequency resp. response time, the CPU load could be optimized in terms of runtime.

Based on this idea, a completely different type of architectural design is possible. For each individual CPU task, we focus on the individual threshold frequency resp. response time. The result is an event-driven system. Events can be viewed at data level, and timing information is transformed to the data level at the outer viewing level. What we need is a mechanism that retransforms this information to the execution time of the CPU without any need for interaction on our part.

What would be needed for such an architectural design?

1. Changes in the outside world have to be identified and have an event-driven impact on our system.
2. Cyclic system requirements also have to be converted to events. A superior timer is needed in order to create events based on the respective cycle time.
3. We need a runtime system that manages both occurring events and the CPU runtime.
4. There is another critical issue: Our interface to the outside world is no longer represented in the form of signals (i.e. continuous-time based) but in the form of events (discrete-time based). Therefore, the system has to memorize independently, from one cycle to the next, the respective states of the outside world.

For many developers, the last item is very difficult to comprehend. Here is a summary of the two different design approaches:

1. Our outside world is connected in the form of signals, i.e. continuously in terms of timing. In this case, the execution time pattern is irrelevant as long as the execution time it is sufficiently short. The sensor signals can be queried at any time. Such a system would query and analyze each sensor signal in each cycle time and redetermines the states of the outside world repeatedly.
2. The outside world is event-driven, i.e. connected discretely in terms of timing (e.g. through a bus system). A change would result in a one-time impulse to the system at a specific point of time. The state of the outside world cannot not be queried again, because our system is effected only by a

change in state (of the outside world) and not by the state itself.

For our event-driven architectural design, this means that the system has to memorize the outside states independently. The states are therefore mapped internally based on the actual situation of the outside world.

One more example: Our system knows two operating modes: the actual operating mode in which the device is normally operated, and a service mode in which basic settings can be made. A system button has a different effect depending on the selected operating mode. A monitor, for instance, has a mode for setting image width, image height, synchronization etc. To this end, the normal control buttons are usually assigned another function. To carry out the settings as required, the suitable operating mode has to be selected.

If the button is activated and generates an event in the system, the system has to know implicitly whether it is in normal operating mode or in service mode.

Here is another comparison of the two architectural approaches:

1. Continuous-time system (analog nature) - conventional switches only.
2. Discrete-time system - buttons only.

Older analog systems had a conventional switch with two switch positions for operating mode selection. This switch could be queried at any time in continuous-time systems. Today, discrete-time buttons prevail. The system has to implicitly remember the system state.

In a timing-driven design, all external sensors would be queried and evaluated in a time loop. Results would only be valid if all sensor signals were evaluated within one cycle.

An event-driven system would only respond to incoming events. The type of activity initiated by an event depends on the internal system states.

What is the difference between the two design approaches in view of complexity?

We have learned that the amount of levels to be considered in the development of a software system affects the degree of complexity. Consequently, complexity is mastered much more efficiently if levels are eliminated.

Let's take another look at the two basic runtime architecture design options in terms of the timing level and data level.

1. Timing-driven approach, e.g. in the form of a main loop

Communication between concurrencies is handled through global variables and continuously connected periphery. Consequently, any data is valid at any time. There is no coupling between timing behavior

and data level that would have to be considered.

The situation changes as soon as interruptibility (preemptive behavior) is required - data access has to be mutually protected and the timing and data levels are coupled. Consequently, this is a suitable architectural approach as long as no preemptive behavior is required.

2. Event-driven approach

Communication between concurrencies is asynchronous. Concurrencies are driven by data changes (events). The timing level is subordinate to the data level. This mechanism also works with preemptive behavior and should basically be chosen if preemptive behavior is required or if there is an event-driven connection of the periphery, e.g. through a CAN bus.

In asynchronous runtime architecture designs, the timing level can be neglected over a long period of time. They comply with the following two features:

- Transformation of the timing requirements to the data flow level. The timing level can be neglected in the context of the software system.
- In reality, the abovementioned feature of course only applies up to a specific threshold frequency. However, as event-driven runtime architectures provide for optimized designs in terms of timing, the threshold frequency is much higher than in synchronous runtime architecture designs.

These examples have shown that a suitable architectural design helps to design either the data level or the timing level such that it can be neglected. Of course, there are further architectural design options to master complex software systems more efficiently, for example, by decoupling levels from each other.

Moreover, levels can be visualized independently of each other through graphical notation (such as UML), providing for enhanced understandability.

The basis for this approach is a so-called runtime system. It can be self-programmed or purchased as real-time operating system (RTOS). Many different operating systems are available today, ranging from open source systems or lower-priced systems for only a few thousand Euros to extensive, higher-priced systems. A simple RTOS ensures initial success. Considering today's price structures, the effort to program such a system by oneself would not be worth it in most cases. In terms of the labor cost for a software engineer, we are talking about 10 man days. The time and effort for even a simple scheduling algorithm would exceed this by far.

But which RTOS is most suitable? The following article offers guidelines for making the right decision.

Moreover, RTOS and UML trainings offer orientation and decision guidelines.

Training dates are listed on <http://www.willert.de/lp-events>

Selection Criteria for an RTOS System

A design decision usually starts with a system analysis. The same is true for the decision whether to use an RTOS, and if so, which one. The basis for selecting an operating system is the architectural design and the related application requirements.

As described before, the runtime architecture design is an efficient instrument for mastering the latent growth of complexity. The following aspects have to be considered when choosing a suitable architectural design:

1. Which concurrencies shall be implemented in the system?
2. What is the response time resp. intermittency of each concurrency?
3. What is the maximum runtime of the individual concurrencies?
4. What is the main nature of the sensor and actuator systems?

Aspects 1 and 2 can usually be analyzed in detail when the project starts. However, the runtime of the individual concurrencies can often be determined only later at the time of implementation. As we will see later on, this issue is not critical. Rough estimations will do for the time being.

The sensor and actuator systems are classified into timing-driven or event-driven designs.

System Analysis

Step 1: Preliminary decision - Timing-driven or event-driven design

First of all, the nature of the sensor systems has to be determined - predominantly time-driven or event-driven. This gives a first indication regarding the internal runtime architecture which should always be designed in analogy to the nature of peripheral connection. One important note: For many years, the trend has been towards an event-driven connection of the outside world. In case of doubt, I would therefore recommend an event-driven architecture.

Here are some basic guidelines:

- Passive sensors are more of continuous-time character, active sensors more of event-driven character.
- Sensor systems connected through a bus system are event-driven, with just a few exceptions.
- Switches on a digital I/O port are of continuous-time nature. Buttons at an interrupt input are event-driven.

The internal architecture should also be based on the type of connection between the periphery and the outside world, as this provides for a basically more harmonic architectural design. But please note: Over time, most designs have been implemented on a continuous-time basis. Consequently, connection of peripherals is often also of continuous-time nature even though the inherent character is event-driven. Even today, for historical reasons, sensors are often queried through a bus even though the sensor could send a signal. Such systems need to be thoroughly cleared out.

Step 2: Analysis of concurrencies

Now, the concurrencies to be fulfilled by our system are analyzed. To this end, the concurrencies, the related timing requirements and estimated runtime could be entered in a simple Excel table.

Here are some definitions:

Response time resp. intermittency: The time in which a concurrency has to react. This can be event-driven or based on a recurring time pattern in which case this time is also called cycle time.

Runtime: Refers to the time required to process the actual concurrency.

Implementation level: We basically know the interrupt level and the program level. The program level is referred to as RTOS level in this case.

Additional scheduling time: Every interrupt of one concurrency by another implies runtime overhead. The period of time caused by an interrupt is also referred to as scheduling or context switch time and needs to be considered when a scheduling pattern is selected.

It might range from only a few μs to several ms depending on the RTOS system used.

The following system requirements are now derived from the exemplary runtime architecture (figure):

1. Are there any runtimes that are longer than the shortest response times? Each overlap requires system interruptibility (preemption).

In the following table, we can see that the runtime of the FFT is 1000 times longer than the reaction times of the motor control, for example. If this was the only overlap in time, it would also be possible to implement motor control as ISR (interrupt service routine) at interrupt level and to do without a scheduling pattern at program level.

But there is another overlap: Pressure monitoring and display control, with estimated runtimes in a 1ms range, overlap with the response time of pneumatic valve power control. The same is true for the pneumatics of the pressure monitoring system.

Concurrency	Required response time resp. intermittency	Estimated processing runtime	Implementation level	Additional scheduling time
Motor control	100 μs cycle	?	?	?
Display control	100 ms ev.	1 ms	RTOS	10 μs
Keyboard control	100 ms cycle	100 μs	RTOS	10 μs
Pressure monitoring, pneumatics	10 ms ev.	1 ms	RTOS	10 μs
Temperature monitoring, motor protection	100 ms ev.	100 μs	RTOS	10 μs
FFT offset, self-learning routine	10 s ev.	100 ms	RTOS	10 μs
Power control for pneumatic valves	1 ms cycle	?	RTOS	10 μs
...
CAN transmitter and receiver routines	10 μs ev.	?	ISR	800 ns
PWM motor control	100 μs cycle	?	ISR	800 ns
Digital input query	1 ms ev.	1 μs	ISR	800 ns
AD conversion	1 μs cycle	1 μs	ISR	800 ns
...

Figure: Example - System analysis of the runtime architecture

(First system analysis can be based on rough estimation. Values that cannot be estimated even roughly at that stage are marked with ?. In the course of the project, the numbers can be corrected based on more experience. Routines to be implemented as HW drivers at interrupt level anyway are already marked as ISR).

According to experience, additional overlap conditions can be expected in the course of implementation resp. the further project which cannot be identified through rough analysis. I recommend the use of a preemptive scheduling pattern in the case of two or more potential overlap conditions.

2. If the use of an RTOS is considered after the initial analysis, the required response times are ensured based on the scheduling times of available operating systems.

In a current system based on an ARM7 architecture with the embOS RTOS, for example, the context switch time would be approx. 8 ns. With the same RTOS on an Atmel AVR, it would be approx. 50 ns. Several different approaches are feasible, and here are two examples:

- A. An RTOS shall be used from a software engineering point of view. This would impact CPU selection. In this case, the AVR's processor performance would border on the defined limits.
- B. For reasons of HW costs, the AVR was chosen as CPU, and the response time of the motor control - 100 ns - would thus be borderline related to 50 ns scheduling time. In this case, this function would have to be implemented as ISR. How come? Consider the scheduling time of 50 ns. Based on a cyclic response time of 100 ns, this would result in a 50% load of the CPU computing performance if this routine were to be implemented at RTOS level. According to rule of thumb, the minimum response times should be longer by a factor of 10 than the scheduling time.

The "power control for pneumatic valves" routine also has to be implemented as ISR. To avoid grids in response time, this routine has to be able to interrupt pressure monitoring, display control and the FFT.

For case B, more routines would have to be implemented at interrupt level than at the actual program level.

3. We have found that most of the concurrencies are of event-driven nature; there is a balance regarding sensor system connection. In this case, I would tend towards an event-driven implementation of the runtime architecture design.

In principle, a timing-driven and event-driven transformation of both worlds is possible. This implies substantial effort which can be easily avoided by choosing a suitable runtime design.

Summary

The timing analysis of concurrencies supplies basic information for decisions regarding the HW and SW architecture. Here is a summary of the most

important basic information needed for a decision on architectures:

- An overlap of response time and runtime requires preemptive behavior and, in case of multiple overlaps, suggests the need for an RTOS.
- The response times of concurrencies and the scheduling times of the runtime systems give an indication of the implementation level for concurrencies. In principle, the interrupt level should be reserved for a few timing-critical routines.
- By multiplying the response times with the sum of runtime and scheduling time, one gets a rough indication of the required HW computing performance (in practice, approximate values can often be derived from experience with previous projects).
- The basic nature of peripherals and concurrencies is the decisive factor for a basically timing-driven or event-driven runtime architecture design.
- In the case of more than 5 concurrencies, the use of an RTOS is strongly recommended.

For more information on scheduling patterns, please visit:

<http://de.wikipedia.org/wiki/Prozess-Scheduler>

Which RTOS Hits the Spot?

Response times

Once you decide to use an RTOS, you have to select the right system. (All based on the assumption that the RTOS supports basic scheduling mechanisms, which applies for virtually all available operating systems but should still be verified in case of doubt).

From our system analysis of concurrencies, we know the required scheduling time, which already facilitates a rough differentiation of systems. As the name suggests, real-time operating systems can be basically classified according to their real-time behavior.

1. Resource efficient operating systems with few services and very fast timing behavior.
2. Operating systems that offer numerous services and drivers but require a suitable extent of resources and thus have longer response times.

In the following, the exemplary reaction times of different operating system categories are listed roughly, based on ARM7 hardware architecture:

Scheduling times and interrupt latency times are

within a range of only a few μ s for the following small and efficient operating systems:

ARTX (contained in the RT Lib of Keil/ARM), embOS (Segger), FreeRTOS, (FreeRTOS.ORG), CMX RTX (CMX Systems), NC OS (FaMicryum).

The scheduling times and interrupt latency times range from some ns up to several ms for the following more complex operating systems:

Windows CE (.NET), LINUX, VxWorks.

In between, a wide variety of operating systems is available on the market: EUROS, PXROS, QNX, Enea OSE ... just to mention a few.

Coordinating the required response times with the RTOS and a suitably high-performance HW architecture is the basis for any decision regarding the runtime architecture design.

What good is an RTOS that, in combination with the selected HW architecture, does not comply with the required scheduling times? In this case, half of the concurrencies would have to be implemented at interrupt level where most services of an RTOS cannot be utilized.

Coverage of required services and functions

Modern applications are required to communicate through bus systems, shall be PC-compatible in reading and writing on memory boards, shall offer complex display controls, carry out dynamic memory updates, provide web services, etc.

Operating system suppliers already offer standard solutions for many of these requirements. Of course, the coverage of own requirements on services and drivers is also a decisive factor. More complex operating systems such as Windows CE or LINUX can play their cards here.

Proven in Operation

Another important decision criterion is whether the RTOS has proven successful when operated in combination with a compiler, IDE and processor architecture. Moreover, if an RTOS has to be adjusted to a specific combination first, you should consider using the existing combination and probably reconsider the selected compiler instead, or choose a different RTOS.

For example, you may want to use the ARM7 architecture based on the GNU compiler and have short-listed a specific RTOS which, however, only supports the Keil/ARM compiler. (Many customers have been using the RTOS in this combination).

In this case, I would recommend postponing the decision for a compiler and first select the RTOS. Of course, there might be good reasons in favor of a specific compiler, but saving 3,000 Euros due to the free GNU compiler is probably not one. Taking over a project that was developed based on a GNU compiler, however, could justify a specific compiler selection.

More Decision Criteria

Source Code

Is the RTOS available in source code? Many developers think that they need the source code to be able to make changes themselves. I don't think this is a critical issue, as the RTOS supplier is responsible for service and maintenance. The source code is much more interesting when it comes to

debugging, for example. Single stepping through the system, it might be quite inconvenient to end up in the disassembler over and over again when stepping into the RTOS routines.

System Level Debugging

Even if your application is based on a suitable architecture and a proven RTOS, it will not always work error-free from the start. A so-called system level debugger enables you to view the resources and procedures of your architecture dynamically and to identify errors more easily. Such type of debugger is worth the investment in any case, and the selected toolchain should contain a system level debugger. In some cases, the debugger is delivered along with the RTOS, in others, there is a defined interface to the debugger.

Prices

Real-time operating systems do not necessarily have to be expensive any more. Many systems are sold for only a few thousand Euros per seat. This is much less expensive than developing the functionality of a simple RTOS system yourself.

The situation changes if so-called distribution licenses (also referred to as royalties) are added to the development licenses. This price structure is pursued by some RTOS suppliers and can raise the cost for using an RTOS to a range of several ten thousand Euros. If the delivered solution offers the respective benefits, this can still be cost-efficient. Such concepts, however, have been going out of fashion in view of other options.

Many open source operating systems, such as LINUX, are available for free at first glance only. These systems often have to be coupled to proprietary hardware, or a so-called board support package has to be purchased. So this option also entails considerable costs.

Conclusion

Requirements for modern applications have become more and more stringent over time, not obviously but permanently. Higher complexity is

not necessarily only caused by new functionalities themselves but also by their interconnection.

Encapsulation is the most efficient mechanism to master complexity. The basis for an encapsulated software system is the architectural design, primarily the runtime architecture design. For this reason, the use of an RTOS is a very good way of mastering increasing complexity in practice.

If you are not quite sure whether, and how, an RTOS might help you improve the architectural design in your specific situation, you should attend our one-day workshop:



WILLERT.

Publisher:

Andreas Willert

Willert Software Tools GmbH

Hannoversche Strasse 21

31675 Bueckeberg, Germany

www.willert.de info@willert.de

Phone +49 5722 - 9678 60

Fax +49 5722 - 9678 80