What’s your rtos’ footprint?

Verifying performance metrics for embedded real time operating systems. By Colin Walls.

All embedded systems have some limitations on the amount of memory which can be included, which means the requirements of the system’s real time operating system – on a given cpu – need to be understood.

Normally, an operating system will use both rom and ram. ROM, usually in the form of flash memory, will store the kernel code, code for the runtime library and any middleware components. RAM, meanwhile, will be used for kernel data structures, including some or all of the kernel object information. There will also be some global variables stored.

When you are looking at the performance and usage characteristics of an rtos, there are three main areas of interest:

• Memory. How much rom and ram does the kernel need and how is this affected by options and configuration?
• Latency. Broadly the delay between something happening and the response to that occurrence, latency is a particular minefield of terminology and misinformation. However, there are two essential latencies to consider: interrupt response; and task scheduling.

• Performance of kernel services. How long does it take to perform specific actions?

Dependencies
There are a number of factors that affect the memory footprint of an rtos and the cpu architecture is key. The number of instructions can vary drastically from one processor to another, so looking at figures for, say, a PowerPC based device, will give no indication of what the ARM version might be like.

Embedded compilers generally have a large number of optimisation settings. While these can be used to reduce code size, that will most likely be at the expense of performance. Data size can also be affected by optimisation, as data structures may need to be packed or unpacked. Again, both rom and ram can be affected. Packing data has an adverse effect on performance. Most rtos products have a number of optional components. Obviously, the choice of those components will have a very significant effect upon memory footprint. Most rtos kernels are scalable, which means that, all being well, only the code to support required functionality is included in the memory image.

Measurement
Although an rtos vendor may provide or publish memory usage information, this data might be particularly hard to interpret and, hence, misleading. Vendors do not mislead their customers intentionally; it is simply that there are a lot of variables and assumptions may be made. That means you may wish to make your own measurements in order to ensure the figures are representative of the type of application that you are designing.

Taking these measurements is not difficult. Normally, the map file – generated by the linker – gives the necessary memory utilisation data. Remember that different linkers will produce
different kinds of map files, each with varying amounts of information provided in a variety of formats. Some specialised tools can extract memory usage information from executable files: an example is objdump.

**Interrupt latency**

The time related performance measurements are probably of most concern to developers using an rtos. A key characteristic of a real time system is its timely response to external events. An embedded system is typically notified of an event by an interrupt, so the delay between the interrupt occurring and the response to that interrupt – the interrupt latency – is critical (see fig 1).

Interrupt response is the sum of two times:

\[ t_{IL} = t_{I} + t_{OS} \]

where:

- \( t_I \) is the hardware dependent time, which depends on the interrupt controller on the board as well as the type of the interrupt, and
- \( t_{OS} \) is the overhead induced by the operating system.

To measure a time interval, like interrupt latency, with any accuracy, requires a suitable instrument. The best tool to use is an oscilloscope. One approach is to use one pin on a GPIO interface to generate the interrupt. This pin can be monitored on the ‘scope. At the start of the interrupt service routine, another pin, which is also being monitored, is toggled. The interval between the two signals may be easily read from the instrument.

The main problem with interrupt latency is the interpretation of published figures. For hardware, you need to know precisely which platform and interrupt controller is being used for measurement, along with such factors as clock speed and cache configuration. The frequency of the timer is also relevant, as its interrupt ‘tick’ is competing with other interrupts for attention. It is also important to know what kind of memory from which the code is running and how the kernel was built. For example, was the code optimised for speed? Knowing which interrupt is in use is also important, as different interrupts may be handled in different ways on different devices.

Lastly, you need to know whether the supplied figure is the best or the average.

**Scheduling latency**

A key part of the functionality of an rtos is its ability to support a multithreading execution environment. Being real time, the efficiency at which threads or tasks are scheduled is of some importance.

The scheduling latency is the maximum of two times:

\[ t_{SL} = \max(t_{SO} - t_{CS}) \]

where:

- \( t_{SO} \) is the scheduling overhead; the end of the ISR to the start of task schedule, and
- \( t_{CS} \) is the time taken to save and restore thread context.

Measurements may be made in a similar way to the interrupt latency timings. The key stumbling block when it comes to looking at scheduling latency is ignoring the starting point. From system idle, there is only the time to set up a task’s context, there is no time taken to save context. When trying to interpret quoted scheduling latency, the key hardware and software factors to verify are exactly the same as with interrupt latency.

**Timing kernel services**

An rtos is likely to have a great many API calls, probably numbering into the hundreds. To assess timing, it is not useful to try to analyse every single call; it makes more sense to focus solely on the frequently used services. Quoted timing figures for kernel services need to be read with the same care as the other numbers that have been reviewed above. Again, attention must be paid to the configuration of the hardware and the software used to perform the measurements.

A particular point to note is the usage of kernel error checking. Many rtos can be built with varying degrees of error checking included. This checking is likely to have an effect on the timing characteristics of API calls.

**Conclusions**

All rtos vendors provide performance data for their products. This information may be very useful, but can also be misleading if interpreted incorrectly. It is important to understand the techniques used to make measurements and the terminology used to describe the results. There are also trade offs – generally size against speed – and these also need to be thoroughly understood. Without this understanding, a fair comparison is not possible.

If timing is critical to your application, it is strongly recommended that you perform your own measurements. This enables you to be sure the hardware and software environment is correct and the figures are directly relevant to your application.

**Author profile:**

Colin Walls is an embedded software technologist with Mentor Embedded [http://blogs.mentor.com/colinwalls].