Introduction to Real-Time Operating Systems: Part 5

Call: +44 (0)23 8098 8890
E-mail: sales@itdev.co.uk (mailto:sales@itdev.co.uk)

Posted 3rd November 2017, By James H (/company/about/team/james-h)

Introduction

In the embedded world, the use of a Real-Time Operating System (RTOS) is commonplace, and with the advent of the Internet of Things (IoT) they are becoming more so. You might be deciding to use an RTOS for the first time, or perhaps you are thinking of moving to a new RTOS. This is the fifth article in our series on RTOSes to help you to get started.

RTOS Considerations

The previous articles in this series have explained what an RTOS is and why you would want to use one for responsive scheduling. However, this is not the only factor that must be considered when looking at RTOSes.

Firstly, real-time systems normally imply some kind of consequence to a failure that is more severe, to some degree, than your music player “skipping a beat”. When system memory has been exhausted, the RTOS should guarantee that it will still service the most critical inputs.

For example, in Green Hills INTEGRITY there is no dynamic allocation in the kernel [1], which means that whilst a kernel task could run out of the memory it has statically reserved for itself, this won’t affect the amount of memory available to other tasks. Thus, a memory leak in our storage driver isn’t going to crash the OS, and isn’t going to affect our car’s collision subsystem – this example was originally included in part 3 of this series. This is also the case for user space applications: the system architect determines how much memory each process can have. The upshot of this is that no buggy user space task can consume all of the system’s resources, starving other tasks.

Secondly, because RTOSes are so pre-emptible, they are also normally optimized for fast-as-possible context switches. When a process is interrupted, quite a lot of work must be done: the register sets must be saved, instruction caches flushed, the next runnable task selected, which in itself can be lengthy process if there are long pre-emption chains, and so on.

RTOSes will usually try and do this as quickly as possible. One optimisation used by RTOSes like MQX[2], is to give tasks flags that the designer/coder must set. For example, if an MQX task doesn’t use floating point maths, it won’t set the task’s floating point flag so that when the task is taken off the CPU, the OS knows that it doesn’t have to save/restore the FP registers, thus saving some valuable time.

Thirdly, with the advent of the IoT age, other concerns have also become more pressing. RTOSes targeted at IoT devices also need to be very space efficient and scalable. For example, the Zephyr OS, and many such RTOSes have gone down the nanokernel route, where the kernel is very small and simple: extras are built on top of it in the form of various modules. The Zephyr kernel and software subsystems can run on systems as small as 8 kB of memory! Other newcomers like RIOT OS can run on systems with as little as 1.5 kB RAM and has a minimal footprint of 5 kB.

The IoT phenomenon has also ushered in new forms of communications. Although communications stacks are not part of the OS kernel, they are part of the OS infrastructure as a whole. The Barr Group Embedded survey 2017 [3] found that over 80% of embedded projects are now using wired Ethernet and over 50% are using WiFi.

According to 2015 UBM Market Survey [4], WiFi and Bluetooth LE are the most popular interfaces with a 61% and 35% share of the projects surveyed. Zigbee and Cellular are the next biggest contenders as 20% and 10% respectively.

This connectedness is only predicted to increase: according to ABI research, 21 billion IoT devices will ship with an embedded RTOS by 2022. This means that other wireless interfaces will increase in popularity, such as NFC, LoRa, 6LoWPAN and more. RTOSes that wish to compete in the IoT space will have to provide a configurable modular communications stack.

Security has also become an ever increasing concern as more and more of today’s devices are connected to the internet. The effect of security breaches can cost companies an estimated 13% of revenues in some cases [3]. Device vulnerabilities can be introduced by insecure web interfaces, a lack of authentication and authorisation, as well as many other sources. A quick internet search will reveal some of the IoT vulnerabilities that have been exploited: everything from information theft to the potential hacking of cardiac devices [4]!

Whilst IoT devices are more likely to be low-power, the systems that aggregate and process the data need to be able to crunch large amounts of data. The Barr Group Embedded survey 2017 [3] found that two-thirds of systems used in embedded projects are now multi-core or multi-processor. With this comes the question of symmetric multi-processing (SMP) and asymmetric multi-processing (AMP).

Why has there been this growth? The answer lies in power consumption: simply increasing processor frequencies to get greater “speed” means greater power. The other way in which greater “speed” can be achieved is through parallelisation: running many tasks simultaneously. This allows a system to do more, whilst using less power to do so. Hence the increase in the popularity of AMP and SMP systems.

In SMP systems, for example, there is only one kernel and it must determine which applications are assigned to which processor. Thus, if you want to leverage SMP, you'll need an RTOS which can support it.

One must also consider industrial safety standards that an RTOS may have to satisfy. Examples some of these standards for different markets are given below:

- DO-178B (airborne systems)
- ISO 26262 (automotive)
- IEC 62304 (medical device software)
- EN 50128 (railway control)
- IEC 61508 (industrial control systems)
- EAL 6+ (NSA)

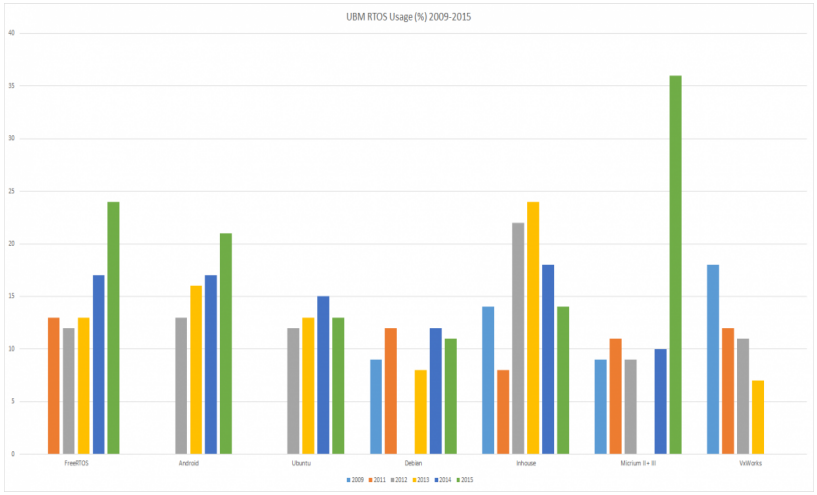
This is where the commercial RTOSes have an edge. They have been designed specifically with many, or all of these standards in mind and so have followed the required planning, development, documentation, testing processes, etc. Often the vendors also provide services to extend the certification to your system (i.e. they'll help certify your own drivers etc.).

Whether certified for one of these standards or not, the RTOS will be following a set coding standard, most commonly MISRA.

Market Shares

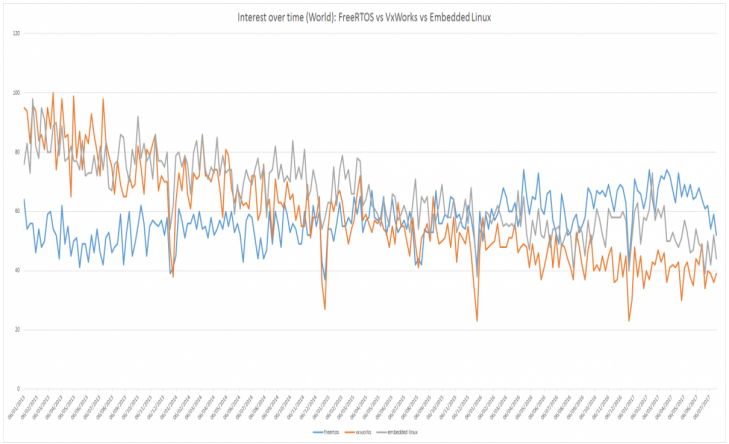
Finally, there is the open source question. In the past, the likes of VxWorks were the "Microsoft" of the embedded world and competitors like Green Hill's INTEGRITY and LynxOS[5] were on the scene and competing hard. The market was very much dominated by commercial RTOSes.

Since then the market has changed dramatically. According to the UBM Market Surveys from 2009 onwards we see that players like FreeRTOS and Android have an increasing share of RTOS usage, whilst the more traditional players have been losing out, as illustrated in the following graph:



(/sites/default/files/embedded/RTOS_market.png)

Using Google Trends to monitor search terms we can see the same trend:



(/sites/default/files/embedded/RTOS_market_google.png)

Over the past 4 years, we can clearly see that FreeRTOS has gained relative popularity at the expense of traditional players like VxWorks, but also open source players like embedded Linux.

The decision to go open source will be determined, mostly, by the level of certification your product requires. If a particular standard needs to be met, you will likely go for one of the proprietary RTOSes. Otherwise, you are likely to go for an open source solution for several reasons. Firstly you have access to the entire OS. When I used to work with RTOS vendors, it was frequently the case that the kernel innards were not available to us. This can very occasionally be an issue when debugging. I had encountered one commercial RTOS where we ended up requesting extra snippets of the kernel's source code because we were debugging a task crash which in the end was found to be caused by the OS's context switch not saving the floating point registers for an FP-enabled task!

The more likely reason people prefer fully open source RTOSes is that everyone can see the code. Take Linux, for example. Because there is such a large and enthusiastic community, everything is reviewed by many, many, eyes so it becomes harder for badly written code to enter the upstream Linux trunk. As people can see the code, security issues and bugs can also be more easily identified and fixed. Also, such vibrant communities are constantly improving their product and everyone using it gets this for free.

How Can ITDev Help?

Since joining ITDev I've seen first-hand the company's experience with embedded Linux, Android and RTOSes like FreeRTOS. As a company of 16 staff, we bring a lot of experience in a small but efficient package. In the last few months alone I've had the opportunity to use my previous RTOS experience to enhance touchscreen drivers, write daemons for Android to enable advanced touchscreen events to be delivered to applications, consult on smart motorways using embedded Linux and more. The knowledge is spread across several developers, so you know that if one team member gets ill, our ability to cover the task is not diminished.

All of our development involves rigorous software development methodologies. Everything is developed under source control and we're proficient in SVN and GIT. All code is reviewed using "Review Board", which is intimately tied into our on-line scrum boards and task tracking systems. We use continuous integration technologies like Jenkins to make sure that all of our solutions are continuously tested, can be delivered in increments, and is regression tested all of the way.

References

1. Green Hills Software, "INTEGRITY-178 EAL 6+ certified, safety-critical RTOS," [Online]. Available: https://www.ghs.com/products/safety_critical/integrity-do-178b.html (https://www.ghs.com/products/safety_critical/integrity-do-178b.html).
2. http://www.nxp.com/products/developer-resources/run-time-software/mqx-software-solutions/mqx-real-time-operating-system-rtos:MQXRTOS (http://www.nxp.com/products/developer-resources/run-time-software/mqx-software-solutions/mqx-real-time-operating-system-rtos:MQXRTOS).
3. BARR Group, "2017 Embedded Systems Safety & Security Survey," 2017. [Online]. Available: https://barrgroup.com/Embedded-Systems/Surveys/2017-embedded-systems-safety-security-survey (https://barrgroup.com/Embedded-Systems/Surveys/2017-embedded-systems-safety-security-survey).
4. UBM & EDN/EE Times, "2015 Embedded Markets Study," 2015. [Online]. Available: https://devzone.nordicsemi.com/attachment/a61052ff4978f8c42b4f6f4b11a1b0e0 (https://devzone.nordicsemi.com/attachment/a61052ff4978f8c42b4f6f4b11a1b0e0).
5. http://www.lynx.com/ (http://www.lynx.com/)

< prev (/blog/introduction-to-real-time-operating-systems-part-4) next >

Latest Blog Posts

(/blog/reflection-2020)

A Reflection on 2020