# How to Pick an RTOS

by Ralph Moore

smx Architect

## Simple RTOS Kernels

There are many simple RTOS kernels on the market. The presumed advantage of these kernels is that they are easier to learn than full kernels. The downside to this is that missing capabilities that are needed by an application end up in the green application code rather than in the proven commercial kernel code. The less you have to design, code, and debug, the more likely you are to meet project goals. The cost of reinventing the wheel, so to speak, is likely to exceed the difference in cost between a full kernel and a simple kernel — even if the simple kernel is free.

It may not be obvious what features your application needs. For this reason, it is recommended that you download evaluation kits of simple and full kernels and try structuring your application for each. A little homework up front may save a great deal of unnecessary development work later.

## Big OSs

At the other end of the spectrum are big operating systems, such as Linux and Windows, which are much too complex for typical embedded systems. Unlike RTOSs specifically designed for embedded systems, these OSs have functionality that is not needed for them. Although big pieces of unneeded code can be omitted, a great deal cannot. Hence you are left in the final product with much more code than you actually need. The main downsides to this are: (1) higher bill of materials (BOM) cost — more memory, faster processor, etc., (2) much slower boot time, and (3) much more to learn and deal with.

A good RTOS comes with all of the middleware and drivers that most embedded systems need. Of course, OSs like Linux and Windows have all of the functionality that a product could possibly need, now or in the future. From a marketing perspective, that may seem to present less future competitive risk. However these OSs also come with a lot of excess baggage that drive up BOM costs and increase support cost and liability risks. So which should you pick — protection against potential escalating requirements or the best fit to today's requirements?  Discussion with Marketing may be helpful — maybe they do not see the need for, nor want, limitless features, or maybe they do. Someone needs to make realistic tradeoffs. You and they should not make critical decisions in the dark.

If your company's competitors have wisely picked RTOSs that fit product requirements well, they will be able to establish a price point, which may be marginal for your company, but good for them. Hence the fruit of your work may be a marginally profitable product that does not contribute to your company's success and thus is of little interest to Upper Management. Here, massive functionality does not necessarily provide the best solution.

Of course, if the product you are developing is very expensive relative to its electronics cost, then a big OS is unlikely to be a significant BOM cost burden. However, it still may not be the best solution if its complexity drives up support or liability costs. Linux, for example usually requires adding an OS specialist to the support team to keep up with revisions, and Windows does not provide full source code. Using something that you do not fully understand is risky. If source code is not available or it is too complex to understand in one lifetime, there could be trouble down the road.

## Time to Market

The concern about time to market is based upon the assumption that for a given product there is a market window, which starts when the first competitor gets its product to market and which ends a fixed number of months or years later. In the simplest model it is assumed that a certain average number of units will be sold per month during this window and that none will be sold when the window closes. Hence time lost at the start of the window results in lost sales that cannot be made up. Worse, it may also result in less market share, which means fewer sales per month after the product does get to market. Thus a schedule overrun may cost your company many times the increase in development cost that you see.

Of course, the actual situation is much more complex than this simple model, and you should consult with your Marketing Department to determine the true cost of missed time to market for your project. Do this before picking an RTOS and other software development tools so you can make proper cost vs. risk tradeoffs. The investment that can be justified for good commercial software and tools may surprise you.

## BSPs and Drivers

This is another area of great potential labor savings for you. Of course, the big OSs are likely to have everything you need. Simple kernels tend to be quite bare, leaving the heavy lifting to the user. Naturally, the more code that is already written and debugged, the easier your job will be. This is especially true for low-level code, some of which may need to be written in assembly language and all of which requires intimate hardware knowledge. Many SoCs are ornery little critters, no doubt “written” by engineers like you who do not have adequate time to perfect their work. Hence, this is an area to check carefully for all prospective RTOSs. Look for RTOSs that already have your processor supported. Ideally the RTOS vendor will have a ready-to-run evaluation kit that you can download and test.

Running evaluation kits is also a good way to pick a processor. Hopefully the RTOS vendor has time measurement tools that will enable you to measure and compare critical times for your application. Check that the RTOS vendor has already written the difficult drivers — e.g. Ethernet and USB. Drivers such as UART and I2C are easier to write and probably will be customized to the application, anyway. Also check into what BSP functions and subroutines are offered with the RTOS that may make writing small drivers easier.

## API Usability

As previously noted, some APIs are too sparse and some are too complex. You probably need something in between. You should consider these questions when evaluating an RTOS API:

- Do you feel comfortable with it? Little things like naming, return values, and the number of arguments do matter.
- Do the manuals document operations well?
- Do they document side-effects well? (If not, you could be in for some surprises.)
- Are there good examples to follow?
- What safety and debug features are built into the API? — i.e. does the API help you to avoid mistakes and to find those that do creep in?

You should also consider the completeness of the API. Does it have everything you need or that you are likely to need? It is especially useful to look into what happens when things do not go as expected. The interplay of the RTOS with the development tools you have selected is particularly important. Try introducing some bugs and see how easily you can find them. Here again, doing your homework up front may pay big dividends later.

## Middleware

In 1975, when the embedded systems industry was in its infancy, a typical embedded device had 8 KB of code, 1-2 KB of RAM, and a handful of switches and LEDs for the operator interface. There was no "middleware." Today, that picture has radically changed, and now middleware is a dominant part of the RTOS selection process. For embedded devices, there are four main categories: networking, file system, USB, and user interface.

**Networking:** A major industry trend is to network all embedded devices. This is useful to download software upgrades, for remote diagnosis, and to upload data and operational logs. The goal is to reduce technician visits. The foundation for networking is the TCP/IP stack offered with the RTOS. These days, it should be a dual IPv4/v6 stack in order to work with either type of network. Some stacks are very large and slow and were not really intended for embedded systems. For embedded devices a dual stack should fit in about 75 KB of ROM and be able to give reasonable performance with about 20 KB of RAM. (Of course, as other protocols are added these footprints may go up.) RAM usage tends to be a sticking point because most SoCs have small on-chip RAM, yet TCP/IP stacks typically need a basic 12 KB plus about 8 KB per active session. Using off-chip RAM may not be desirable because performance can go way down and it adds to BOM cost.

A plethora of protocols are available for TCP/IP. Most popular among these for embedded devices are web servers, also known as HTTP servers. A web server permits accessing an embedded device via a standard browser, such as Internet Explorer, in order to change settings or view device operation. Beware, however, that modern browsers are designed to operate with web servers running on powerful processors having large memories. They, for example, attempt to initiate multiple sessions for better performance. A small SoC-based web server, with minimal RAM, cannot do this and thus performance may suffer unless web pages are kept simple.

Other useful protocols are HTTP client, which allows accessing a home-base website for configuration parameters or to report problems, SNMP to manage embedded devices, DHCP to obtain temporary IP addresses, FTP for file transfers, and many more that may or may not be useful for your application. It is important to pick an RTOS that has a broad offering of TCP/IP protocols because of the difficulty to decide what to use. Customers often order additional protocols long after their projects have started.

When an embedded device becomes network accessible, network security becomes important, even if the device is on a private network. Imagine if a competitor's spy could access secret information, or if a hacker could cause damage to your device or to your customer's equipment, product, or reputation. SSL (Secure Socket Layer) is extensively used for financial transactions and is increasingly being used in embedded systems. Through the use of "certificates," your device is able to verify that you are you, and you are able to verify that it is it, after which you are able to exchange safely-encrypted information back and forth.

Many SSL packages on the market are too large (e.g. 500 to 1000 KB) for small SoC devices. However, some vendors offer SSL packages in the 50-75 KB range, which are obviously much more suitable. Another consideration is performance. Encryption or decryption is basically division of an entire message by a 2K-bit or 4K-bit divisor. This is a bit too much for a 70 MHz processor and message throughput will not be good. Some SoCs offer encryption hardware to ameliorate this problem. Even if you initially do not plan to use SSL, it is advisable to make sure that your hardware and RTOS selection will allow adopting it in the future without great difficulty. Trying SSL out might be an eye-opener.

A final major embedded networking trend is toward wireless devices. These can greatly reduce installation costs, avoid bringing a connector out of the package, and are much more convenient for servicing via a laptop, pad, or cell phone. (One can access a piece of equipment from a safe and convenient distance.) IEEE 802.11, also known as WiFi, is the leading protocol for this purpose. Some RTOSs offer it; most do not. The WiFi stack should support both peer to peer and access point communication. Security is especially important for wireless communication. WPA2 (WiFi Protected Access) is needed for real security and should be available with the WiFi stack.

**File Systems:** Some sort of file system is needed by most embedded devices. There are many alternatives: (1) FAT file system used to read and write popular media such as SD cards, USB thumb drives, CompactFlash, and others. Generally speaking, the FAT file system must be compatible with Windows so information can be transferred between a PC or laptop and the embedded device. FAT file systems generally are not power-fail safe, except when extended with non-standard techniques such as journaling. (2) Flash file system for use with NAND, NOR, or serial flash chips. This kind of file system handles the complexities of dealing with flash devices and offers a low-cost solution to storage of large amounts of data within a device. A good flash file system is power-fail safe and uses only a very small amount of RAM to handle very large flash devices. NAND flash is normally used when there is a large storage requirement. SLC (Single Level Cell) NAND devices are best for embedded systems; MLC (Multi-Level Cell) devices are too sensitive — cells can be disturbed by reads and writes to neighboring cells. MLC devices require large ECCs (Error Correction Codes) to function reliably. To be practical for SoC devices, ECC hardware assistance is needed for MLC chips, but not for SLC chips. (3) Loggers. Some RTOS vendors offer very simple, low-footprint flash loggers that are useful for things like down-hole instruments, where the device is incommunicado while gathering data, but data can be dumped after it returns.

**USB:** USB has replaced serial in the commercial world and is rapidly doing the same in the embedded world. It comes in three flavors:

(1) A USB host stack enables connecting to your embedded device a USB device such as a Thumb drive, serial device, audio device, HID (Human Interface Device), printer, modem, hub, WiFi, etc. To do so, the appropriate class driver runs on top of the host stack. A major prospect for your product can unexpectedly need to connect to an unusual device such as a USB modem. This is no problem if your RTOS vendor has the needed class driver.

(2) A USB device stack allows connecting your device to a PC or a laptop. Function drivers running on the device stack communicate with class drivers running on the PC or laptop. A very useful function driver is the serial driver. When a device with it is plugged into a PC USB port, Windows recognizes the device as a serial device and assigns a COM port to it. Thereafter the device can be accessed like a serial device through that port — by, for example, using a terminal emulator. Other function drivers include Ethernet over USB, mass storage, media transfer, multi-port serial, and video. Ethernet over USB (RNDIS) provides an interesting capability. With it, a web server in a device can be accessed via a browser on a PC as though the device were connected to a LAN. So, for example, a technician, with a laptop, can plug it into the embedded device and access it the same as he would through a network.

(3) An OTG (On The Go) stack can look like either a host or a device. OTG allows switching the role of the peer device (which is also OTG). This is useful for things like digital cameras, but most embedded devices need both interfaces simultaneously and thus need two connectors, two controllers, and both the host and the device stacks.[1]

It is important to learn about the vast array of possibilities that USB offers[2], then to make sure that the RTOS you are considering has them covered. A broad assortment of class drivers and function drivers is important because a device's USB requirements are likely to change with time.

[1] For more discussion see: "When to Use USB OTG" by Yingbo Hu.
[2] For ideas see: "Ways to Use USB in Embedded Systems" by Yingbo Hu and Ralph Moore.

**GUI (Graphical User Interface):** Many embedded devices just need some simple text output and therefore no GUI is required. Good RTOSs provide console functions for sending messages and text to a small display. Devices that have 1/4 VGA and larger displays can benefit from a GUI. A good GUI allows easily creating attractive interfaces using window builder, font capture, and image conversion tools. It provides black and white, gray scale, or color, touch screen, widgets, and more. The RTOS vendor should offer a range of GUIs from minimal to elegant. However, expecting smart-phone equivalence is not realistic for most embedded devices. (Android® requires 100's of megabytes of memory.) Nonetheless, a good GUI can come close — it is possible to get very nice screens with a small-footprint GUI that has been designed for embedded systems. As with other RTOS features, it is good idea to try evaluation kits to see which achieves what you need with the least effort on your part.

A final point on middleware is that it is best if the RTOS vendor developed the middleware, itself, so it will have the in-house expertise to provide you with good support.

## Tool Integration

An often-overlooked consideration for picking an RTOS is its quality of tool integration. Good integration with quality development tools allows starting quickly and being more productive. The quality of tool integration can be determined by downloading an RTOS evaluation kit and seeing how easy or difficult it is to use. Most RTOS vendors offer kernel-aware add-ons for tool vendor debuggers. These typically provide event timelines that show when tasks and ISRs started and stopped and why they did so. They also may provide stack and processor usage graphs and permit inspecting RTOS control blocks, event buffers, error buffers and other RTOS objects. The more system-level information you have available, the easier it is to track down system-level problems such as task starvation, prioritization problems, priority inversions, stack overflow, etc. Some RTOS vendors have extended their kernel-awareness tools to their middleware, which is very helpful in dealing with networking, file, and USB problems. Good error reporting by the kernel (e.g. OUT OF TCBS) is also a major help during debugging. Stack overflow and high watermark reporting are crucial to saving one's sanity.

## Other Factors

Other factors are also important to get a quick start. It helps if the release is pre-configured to run on a standard evaluation or development board for your processor; this permits starting to use the software immediately and to build confidence that everything works as advertised. Low-level BSP notes are a big help, not only for understanding how the BSP code works on the evaluation board, but also for migrating the BSP (and all other software) to the final production board. Good BSP notes explain configuration settings and point to the locations of important constants, files, etc.

Good BSP code backed up with strong support has become very important because SoCs are now so complex. Typical data sheets are well over 1000 pages of excruciating detail that is often incorrect and incomplete. In order to produce a reliable BSP and drivers, the RTOS vendor must blaze a path through the SoC documentation. Like an early pioneer, it is good for you to have a path to follow. Help in the form of notes and responsive technical support, can keep you from going off the path into the weeds. Assess the support you receive during evaluation as an indicator of support you will receive after you buy.

## Conclusions

Picking a simple RTOS kernel because it easy to learn is not a good solution for most projects, because needed capabilities that are not in the kernel end up in the application as green code rather than proven code. Going to the other extreme of picking a complex OS because it is future-proof against anything Marketing may require, is also not a good solution because it burdens the project with excessive complexity and cost. As a consequence, the very success and on-time delivery of the project is jeopardized for future eventualities that may never happen. It is better to talk with Marketing and make realistic decisions up front. (If this doesn't work, you may need to enlist the help of the businessman who runs your company.) Finally, we have reviewed the many considerations that go into creating a modern RTOS that is good for now and for the future. Such an RTOS is properly viewed as an important member of your team, worthy of lavish salary and benefits.

Ralph Moore  ralphm@smxrtos.com
Micro Digital, Inc.
www.smxrtos.com