One-click distributed real-time in just 10KBytes

Eric Verhulst

Developing embedded real-time applications can be a daunting task. The engineer sits on top of hardware, which has limited memory and processing cycles available, but still the system must meet stringent real-time specifications. If size and power consumption are less of an issue, it can help to use a supersized processor, but that implies that the system is connected to the mains. Often, this is not an option. Power consumption is an environmental issue and size matters. Smaller often means less cost and allows to build smartness in the actuator or sensor itself.
In addition, GHz notwithstanding, the faster the clock rate, the more external memory is a serious bottleneck and using caches kills the real-time predictability. On-chip SRAM is much better suited for this, but that's a precious resource. As a rule of thumb, the solution is then to distribute the application over multiple, slower processors. This eliminates the bottleneck, can reduce power consumption and provide for redundancy. When done well, the application might even be faster than when using just a single high speed processor. In addition, this allows us to use processors that are fit for the application. Use a microcontroller for a sensor, a high-end DSP for performance and an embedded host for connecting to the external world. But how do you program such a mixed real-time embedded system?

The worst case solution is to handcraft the code. It can mean using a RTOS, no RTOS at all or a host-OS like Linux or Windows and then knitting it all together. This is an error-prone and complex job. Worst of all, making changes is hard and there is no 100% certainty it will meet all specifications. The reasons are that such an approach lacks a clear and coherent systems view and that the communication must be programmed at the application level.

But, what if there was a solution with which you could simulate and develop it all on your desktop PC? And when the development is done, with a few clicks the application could be distributed over the available processors with no changes in the source code at all? You could run the application and profile it, make a few changes, recompile till you get it fully right? There is such a solution and it is called OpenComRTOS. It was designed as a network-centric RTOS and can be deployed from small limited memory microcontrollers to widely distributed systems. It was also developed using formal methods and as a side-effect it is very small. No JVM measured in Mbytes, but compiled C code in KBytes. How does it do that trick?

Contrary to many RTOS and OSes, OpenComRTOS was not handcrafted, but formally designed from the ground up, although the team had a long term experience developing a parallel processing RTOS, called Virtuoso, (later on aquired by Wind River Systems) . Given that in multi-processor systems, whether it's a multi-core chip or a distributed one, communication is the main issue, it was designed as a communication system using prioritised packet-switching. Packets are used everywhere, at every level. This is convenient for several reasons. Packets are easy to manipulate, because they have fixed size. Internet is based on it. Packets can also carry arguments and data. The packet handling layers don't need to know what is inside the packet. They just pass packets around. In OpenComRTOS, packets are composed of a header and a payload. They carry a priority and they carry routing information. Everything else is service or application specific. But what's unique in OpenComRTOS is the fact that packets are used for task interaction. This applies also for kernel and drivers tasks and interfacing with interrupts. It even applies for interfacing with the kernel task between Interrupt Services Routines and the kernel. It makes the whole system uniform and straightforward.

While traditional RTOS have separate services for e.g. events, semaphores, fifos, resource locks, etc. in OpenComRTOS these services share a common so-called “hub” functionality. Such a hub is a generic "Guarded Atomic Interaction" entity, a core concept in formal methods. To put you in the picture, think of Chicago airport. Airplanes come in and offload their passengers and/or cargo. If Chicago is your destination, you have reached your destination with success. If not, you have to wait till your connecting flight arrives. Both flights "sync" in the hub, exchange passengers and/or cargo and both flights can depart again.

What are the benefits of this architecture? First of all, it cleanly separates communication and processing. As a result, it also makes the program independent from the underlying hardware. Of course, one still has to compile for each processor and assign the tasks to a processor in the network, but that's it. The priority of each packet also provides system-wide real-time behaviour, because QoS is a matter of assigning the right priority, including support for priority-inheritance.

But there's more. Hubs can be user defined without changing the kernel task. Task interaction is essentially using a hub as an intermediate entity, but based on some conditions. These conditions are called the "synchronisation predicate". Often it simply means that there must be a corresponding pair of requests from two tasks. When that condition is met, a Hub specific "action predicate" will be called. It can be as simple as setting an event status to true, but it can be as complicated as starting up a motor controller. Of course, both predicates are implemented as plain C functions, so it is easy to program new ones. The default ones, that come with OpenComRTOS, are the same as you find with any other RTOS, except that they work independently of the hardware topology and they carry a low overhead. Tasks can synchronise will being placed on different processors while the Hub entity they use (e.g. a semaphore) is placed on a third processor. Actually, it even allows to process an interrupt another processor where the interrupt was received.

A major benefit of the OpenComRTOS architecture is scalability. To start with, the code size can be very small and still provides a lot of functionality. A basic hub entity is called a "Port" and provides simple packet exchange. This is enough for basic synchronisation and communication, even systems wide, while the developer can define his own protocol conventions . A typical code size for such a configuration is less than 5 KBytes, but by optimising some data-structures and accepting a few limits, it was possible to synthesise it down to about 1 KByte on a single 16bit microcontroller with 2KBytes for the multiprocessor version. Yes, all in C (except context switch and ISR interface routines). Scalability also means that an application, developed on a single processor, can be distributed over a network of processors with no change to the source code or the other way around. Every task or hub can be placed on any node in the network. The current implementation limit is 64K of nodes. Unless hardware dependencies are present, the only precondition is the availability of an ANSI-C compiler.

And last but lot least, OpenComRTOS is a safe RTOS. Besides the fact that it was formally developed during the design stage, the final implementation was formally verified by building formal models of each service. Some of the beneficial properties are the absence of buffer overflows and the capability of auto-throttling. Furthermore, most code and datastructures are generated. Mallocs are not used. The packet architecture also avoids that user tasks can modify other task's data. The crucial pointer manipulations are done by the formally developed kernel. The small code size also helps. Besides that fact that it is easier to squeeze it all in the on-chip SRAM, the architecture is very clean, reducing the probability of unintended side-effects.

Practically speaking, how does a developer program an OpenComRTOS application? The easiest way is to use OpenVE, a generic Visual Development Environment, running under Windows and

Linux. In OpenVE, the user draws his tasks and hubs (events, semaphores, fifo's, etc.) on a canvas and connects them. Such a connection allows him to select from the available services, while OpenVE will insert the service call in the appropriate task source. OpenVE gets this information from an application specific metamodel. On the canvas, he can also define task and hub parameters, like stack space, priority and especially the target node. In a first instance, it is advised to map them all on the host node, because this simplifies simulation. Using a host server task, application tasks can use stdio, file IO and graphics, or the user can develop his own access to host services. He can animate the application and examine the event traces using the event viewer. When satisfied, it is a simple matter of remapping tasks and hubs to the real target. Of course, there is an evident pre-condition: The network must be in place. Besides a physical connection, this requires that communication drivers were developed once. As they only have to be able to send and receive fixed size packets, their development can be straightforward.

So, why should OpenComRTOS make a difference for application developers? There are many reasons. First of all, OpenComRTOS was developed with a real systems' view in mind. Often, engineers will have such a view in mind when the project starts. At that level, a system is composed of entities that interact. Each of the entities will then provide one or more functions to meet the specifications. With OpenVE, it is natural to map these entities to tasks and use the kernel services to synchronise them and to exchange data. Some people would call that modelling. From within OpenVE, the engineer can then start refining his program. He can simulate inputs and outputs on his PC. When done, he can then recompile for his real target, even if that is composed of multiple types of processors. In a first instance, he might even keep the PC in the network.

What this means is that when using OpenVE and OpenComRTOS, there is a seamless path from specifications to executables. This avoids a lot of errors and issues that are unavoidable when having to mix tools, libraries and RTOSes from different vendors. The result is that engineers are free to think about the real application at hand and productivity is high. They barely have to worry about the overhead and hidden bugs. OpenComRTOS was formally developed and as a result is very compact, scalable and reliable at the same time. OpenComRTOS is not just another RTOS. It is an enabler for better applications, whether they are running on a small microcontroller, a multicore CPU or even a network of mixed processors. All that in just a few Kbytes.

The last novelty is it licensing model. While binary licenses are available, Altreonic also introduced a so-called “Open License” model. A growing concern for many industries is long term availability and support of the technology they use. This is in particular important for sectors where safety is an issue or where the lifetime of the product is measured in years if not decades. While sometimes escrow agreements are used for such purposes or open source solutions can help, these solutions do not provide enough flexibility and assurance. For example, even if the source code is available, where are the design documents that allow you to understand it? An Open License goes beyond that stage and provides the licensee not only source code but also design documents, the formal models, the test suites, etc. It also allows to create binary installations.

As a conclusion we can say that OpenComRTOS has taken the concept of what an RTOS is all about to a new level. The use of formal modelling was a major factor in achieving a universal and generic solution for embedded real-time programming. RTOS 2.0 has arrived.

Interested readers can download the Win32 version of OpenVE and OpenComRTOS for free from www.altreonic.com

Eric Verhulst, CEO