DEBROS: DESIGN AND USE OF A LINUX-LIKE RTOS ON AN INEXPENSIVE 8-BIT SINGLE BOARD COMPUTER

Mark A. Davis, NSCL, East Lansing, MI 48824, U.S.A.

Abstract

As the power, complexity, and capabilities of embedded processors continue to grow, it is easy to forget just how much can be done with inexpensive Single Board Computers (SBCs) based on 8-bit processors. When the proprietary, non-standard tools from the vendor for one such embedded computer became a major roadblock, I embarked on a project to expand my own knowledge and provide a more flexible, standards based alternative. Inspired by the early work done on operating systems such as UNIX™, Linux, and Minix, I wrote DEBROS (the Davis Embedded Baby Real-time Operating System), which is a fully pre-emptive, priority-based OS with soft real-time capabilities that provides a subset of standard Linux/UNIX compatible system calls such as stdio, BSD sockets, pipes, semaphores, etc. The end result was a much more flexible, standards-based development environment which allowed me to simplify my programming model, expand diagnostic capabilities, and reduce the time spent monitoring and applying updates to the hundreds of devices in the lab currently using such inexpensive hardware.

INTRODUCTION

Since I first joined the Control Systems group at the NSCL, my primary focus has been on designing and developing software for embedded controllers. My first project was to upgrade several existing controllers from an in-house SBC design based on the Motorola 68701 micro-controller. These controllers used a low-speed RS-485 multi-drop network that resulted in frequent communication outages whenever someone disconnected one of the controllers or one of the connections failed. The goal of the upgrade project was to replace the RS-485 networks with a point-to-point Ethernet network.

To that end, an inexpensive SBC was chosen that incorporated the 8-bit Rabbit 2000 micro-controller and a 10base-T Ethernet interface. The SBC was the BL2105 from Z-World, Inc. (later purchased by Rabbit Semiconductor, and now part of Digi International Inc). This SBC includes 512K of RAM and 512K of flash memory and runs the Rabbit processor at 22.1MHz. At the time, this was a huge step up from the 68701 design which had only the built-in 128 byte RAM, 2K UVEPROM, and a top speed of 2MHz.

Another major improvement in this move was the change from programming entirely in assembly language to writing almost all the code in C.

With an Ethernet interface included on the SBC and a library of functions to support IP networking, the change in the communications interface was not a difficult task. Once that was accomplished, it was time to consider the many other enhancements made possible by the much more capable hardware.

EARLY ENHANCEMENTS

Compared to the Rabbit-based SBC, the earlier 68701 design had some major limitations:

- The tiny memory severely limited code size
- Modifying UVEPROM requires physical access
- All code was written in assembly language

Given these limitations, there was no way to do regular firmware upgrades, much less add functionality beyond the bare requirements.

But the new hardware did not have these limitations. With vastly more RAM, software rewritable persistent storage (flash), and a high-speed communication interface, increased functionality and frequent updates were now possible. As the number of controllers and the complexity of their software grew, it quickly went from being possible to being necessary.

And so it began: More processing power at the controller level meant the controllers could do more of the work and could respond much faster to changes than the control system could. The size and complexity of the code on the controllers grew exponentially, which required additional diagnostic capabilities (adding yet more code) and an efficient way to update the firmware on the rapidly growing number of controllers.

It wasn't long before the inevitable happened: The new controller's larger memory space was used up, and there was no more room for enhancements.

A SQUARE PEG IN A ROUND HOLE

Several factors contributed to our problem of not being able to continue expanding the capabilities of the new controllers, including imitations in the vendor's development tools. But it also became apparent to me that a more basic issue needed dealing with: The programming model I was using was resulting in large, complex code that was difficult to maintain and was eating up memory much too quickly. Something needed to be done.

At the time I was using a cooperative multitasking model to keep my code organized in to separate "tasks". Scheduling consisted of the simple round-robin method of a series of function calls inside an endless loop, with each function representing one task.

In the simplest cases, each function will run from beginning to end each time it is called. This "run-to-completion" model can be very efficient. It can also greatly simplify concurrency issues because no "task" is every pre-empted.

The problem with the run-to-completion model is that the average response time of the system for any single function grows larger and larger as the number of functions and/or the time spent in each function grows. When the average (or maximum) time for one pass through the main loop becomes too long, you are forced to break functions into smaller and smaller pieces to maintain acceptable response times.

At that point, the simplicity of the run-to-completion model falls apart. Breaking each function in to smaller and smaller parts and executing just one part each time the function is called stops the average time between calls from growing too large. But it also means it takes longer for each function to do the work it previously did during a single call.

In addition, breaking each function into pieces only works if each piece can pick up where the last one left off. It means every piece must be responsible for maintaining the internal state of the function between calls.

So every function becomes larger, slower, and more complex, even if you aren't adding any functionality to it. And each additional function or enhancement to an existing one potentially requires you to adjust the number, size, and/or content of the pieces in other functions.

And this doesn't even address the issue of different priorities for each task, and how to minimize the time between when an event occurs and when the function that needs to process it is called.

Eventually I realized that this was not a sustainable model. To run smoothly, the expanded capabilities of the controller required the behaviour of a pre-emptive, priority-based multitasking kernel, and I was trying to achieve that using a cooperative multitasking model. The result was that every task had to take on the responsibilities that are normally handled for them by a pre-emptive kernel. Once I realized this, it was painfully obvious why my code had become so large and complex. The question now was what could I do about it?

A PARADIGM SHIFT

At this point I knew a fundamental change was needed. The obvious alternative to the cooperative multitasking model was the pre-emptive one. If I had a pre-emptive kernel, there would be no need for every task to do the kernel's job. The mass of complex code I had to add to each task could be removed, which would free up a lot of memory and make them run faster. But of course, recognition of a problem doesn't automatically mean there is a practical solution.

When considering my options, my first thoughts were of MINIX and Linux. Even early versions of these operating systems where "complete" in the sense that, in additions to having pre-emptive, multitasking kernels, they also supported login shells and user commands. And I knew from personal experience that the original version of MINIX could run reasonably well on a 4.77 MHz IBM PC with only 256K memory. So I was confident that what I needed was possible.

Unfortunately, even the smallest Linux distributions I could find were far too large for my use. I briefly entertained the idea of trying to locate an electronic copy of the original source code for MINIX, but the design of MINIX varied in some very significant ways from what I needed (e.g. it used run-time loaded re-locatable binaries and a message-passing microkernel, both of which pose problems for real-time scheduling). [1]

Having failed to find a "complete" solution (a kernel, user interface, support for sockets), I decided the next best thing would be to use a pre-made kernel and write or port the rest of what I needed.

DEBROS IS BORN

I knew that, even if I found a suitable ready-made kernel that supplying the rest was going to require a lot of work. But in addition to the very real need for a more practical runtime platform, I was motivated from the start by the desire to learn about how pre-emptive multitasking actually works. When I was getting my degree in Computer Science the primary focus was on formal languages and writing compilers. There is no end to how useful the insights I gained from that knowledge have been over the years. But I am finding that the things I learned while doing this project have been just as fundamental to how I think about programming. In retrospect I am surprised that this topic was all but ignored in my college courses.

So I eagerly began my new programming and self-learning project by purchasing a pre-made kernel named TurboTask from Softools, Inc. along with a copy of their compiler.

TurboTask is a very small and efficient kernel that supported all the basic features I needed. The Softools compiler and linker produced smaller and faster code than the hardware vendor's tools, and included a copy of the hardware vendor's network stack (whose API I was already familiar with). All my early development was done using the TurboTask kernel, and the first version of DEBROS used on controllers put into production was one that incorporated it.

Unfortunately, as things progressed, it became apparent that the TurboTask kernel was not as good a fit as I had hoped it would be. It was indeed small, fast, and highly optimized. But it really was designed for someone programming at a much more hardware-oriented level. It just didn't mesh well with the needs of the UNIX-style OS I was developing. One limitation in particular - the requirement that the stacks for all tasks had to reside in the first 64K of memory – became a deal-breaker.

Eventually I had to replace the TurboTask kernel with my own task switcher, which was itself derived from an example provided by SHDesigns, the firm that sells the Download Manager/boot-loader we now use. The Softools compiler and tools, however, have continued to be critical to the success of DEBROS.

Regardless of the implementation details, by this time I had achieved my primary goal: The individual tasks no longer had to do the work of a kernel. I was able to strip out all the extra code that had nothing to do with each task's primary purpose. The resulting reduction in size and complexity was so significant it was if a thick fog had lifted and revealed a sunny sky.

But as important and fundamental as that change was, there was another one that has, in many ways, been just as important: My adoption of the UNIX standard.

THE ADVANTAGE OF STANDARDS

Like anyone that enjoys their work, I try to keep tabs on what is going on in the industry. For me this includes looking at the endless stream of new computer devices both big and small and seeing what kind of hardware and software they use.

One of the things that has become very apparent is that whatever the computing platform, be it a tiny computer no bigger than an Ethernet connector that serves up web pages, the latest smart-phone or e-book reader, or a building-size server cluster, there is one thing they nearly all have in common: Some level of compatibility with UNIX.

Linux and its offshoots (e.g. Android) are probably the best known examples. Various flavours of the Berkley Standard Distribution (BSD) are also quite common, especially when you include derivations like the open-source Darwin OS and its ancestors that form the core of Apple's OS-X operating systems. Even Microsoft provides UNIX compatible subsystems for its Windows operating systems.

The bottom line is that 40 years after it was born, the Application Programming Interface (API) spawned by UNIX has become an almost universal standard. Any programmer who has ever written a command-line application has at least some familiarity with the stdio and stdlib libraries. And if they have written a network server they know what sockets are and have almost certainly used the standard BSD socket API. So when I started writing my own OS, it only made sense that it too should be based on such a widely supported and familiar model.

Adopting a proven, widely supported standard can have many advantages over creating a proprietary interface. One that was critical to my own success was the availability of some very well written textbooks on subjects that were at the core of what I was trying to achieve. For my work on DEBROS, there were two in particular that I referred to constantly. The first is "UNIX Network Programming", by W. Richard Stevens [2]. It has since been expanded to multiple volumes and continues to be highly regarded. The second is "Understanding the Linux Kernel", by Daniel P. Bovet and Marco Cesati [3]. While the latter is a detailed look at the Linux kernel in particular, the topics and issues covered apply to any multitasking kernel. Both books are well written and I highly recommend them for anyone wanting to learn more about network programming or multitasking in general.

For programmers, the use of the UNIX standard means that writing code for DEBROS is no more difficult than writing code for any UNIX compatible OS. While DEBROS will never be fully UNIX compliant, the functions I have implemented use the same parameters and provide the same behaviour whenever practical. So learning to program for DEBROS does not require learning a new API. As proven by another person in our lab, it is possible to compile and test code written for DEBROS on other operating systems like Linux with few if any changes. Probably the most significant difference is simply learning to think on a smaller scale.

KEY LESSONS LEARNED

The primary purpose of an operating system kernel is to manage a set of shared resources. And probably the most important resource that it manages is CPU time.

In the process of writing DEBROS I learned a lot about the concepts and methods that make it possible for a kernel to manage resources effectively. Even the briefest discussion of them all would not fit in this paper. But there is one that, probably more than any of the others, serves to demonstrate both the complexity of the issues an efficient kernel must address and the ingenuity that has gone in to addressing them.

This concept is blocking vs non-blocking function calls. Anyone who has used the UNIX poll() or select() functions has been exposed to this concept to some degree. But what I found most surprising is that it applies to so many other functions, including ones I have used for decades, and yet I was barely aware of it or its importance.

Blocking and the Scheduler

The scheduler is the part of the kernel that is responsible for deciding which task should get the CPU next. While scheduling methods vary widely, they share a common goal: Don't waste CPU cycles.

The most basic piece of information a scheduler uses when choosing a task is what "state" the task is in. There are many different states a task can be in, but the key ones relevant to this discussion can be summarized as "Ready", and "Blocked".

A task is "Ready" if it currently has the CPU or has been pre-empted. The only thing a "Ready" task is waiting for is the CPU.

A task may become "Blocked" when it calls a function to perform an operation and the operation cannot be completed immediately (e.g. writing to a socket). The blocking function initiates the requested operation and then loops, waiting until the operation completes. The simplest possible wait loop looks something like this:

```
while (! eventOccured) { }
```

The problem with this very simple approach is that "blocked" tasks continue to use as much CPU time as the scheduler will give them. Blocking is very common, so this approach wastes large amounts of CPU time.

We can reduce the waste quite a bit with one small change:

```
while (! eventOccured())  { schedule(); }
```

Calling the scheduler directly allows a task to voluntarily give up the CPU. The affect here is to limit the loop to one execution each time the task is resumed.

A significant improvement, but there is still a lot of CPU time spent checking on events that haven't occurred yet. Having the scheduler choose blocked tasks less often, will minimize the waste, but at the cost of increasing the average response time to events.

The ideal situation would be for blocked tasks to use no CPU time at all. Once blocked, a process should not be given the CPU until the event that will allow it to unblock occurs. While not perfect, there is a method that does approach this ideal.

Wait Queues

A wait queue is a doubly-linked list of entities that all have something in common. The most typical use is to keep track of which tasks are waiting for an event related to a shared resource. The wait queue is created by the driver that is responsible for the resource. Tasks are added to the queue by blocking functions when they need to use the resource, and are removed by the driver when the resource becomes available.

The following shows how wait queues can be used in wait loops to make blocking much more efficient:

```
for (;;)  {
  set state to blocked and add to wait queue;
  if (eventOccured)  break;
  schedule();
}
set state to Ready and remove from wait queue;
```

While it may seem a bit odd, the order shown for these steps is critical. Any other order can result in waste we are trying to avoid or, worse yet, a hung task.

The first step is to set the state of the task to Blocked and add the task to the event's wait queue. Note that pre-empted tasks are treated as being in the Ready state, regardless of what their state is set to, so even if the task gets pre-empted after changing its state, it will not hang.

The next step is to check to see if the event has occurred. If it has, we exit the loop.

If the event has not occurred, we call the scheduler for the reasons explained previously. When we call the scheduler, one of two things will be true.

The first is that the task was not pre-empted between when it checked on the event and when it called the scheduler, or it was but the event didn't occur before it was resumed. In that case the task is still in the event's wait queue and its state is still set to Blocked. Because the scheduler was reached by the task calling it directly, it will not be resumed until the event occurs, at which point the driver will remove it from its wait queue and set the task's state back to Ready.

The other possibility is that the task was pre-empted after the check and the event occurred before it was resumed. In that case, the driver will have removed the task from its wait queue and changed its state back to Ready.

One of the subtle aspects of this logic (which was actually missed in an early version of Linux) was that you have to check on the state of the event after adding the task to the wait queue. Checking before you do so doesn't cause a problem, but checking after is critical. Consider what could happen if the only check is before:

- The check returns false and the task is pre-empted before it can add itself to the wait queue.
- The event occurs while the task is pre-empted.
- The task resumes and adds itself to the wait queue.
- The task is now dormant waiting for an event that has already occurred.

Even if the task does get resumed by a later event, at the very least it will have been delayed. Worse, it will have missed the previous event which, depending on the circumstances, could have fatal side effects.

When properly implemented and used by application code, blocking can be a very powerful tool in the quest to create an efficient system. As shown here, not only can a task block indefinitely with little or no overhead, but it will be resumed with minimal delay when the event it was waiting for occurs.

Until I wrote my own OS, the concept of blocking vs non-blocking calls barely penetrated my consciousness. Even when using calls like poll() and select() it never occurred to me how key they are to the efficient operation of the OS, as well as to writing efficient applications.

SUMMARY

The lack of a standard software platform for 8-bit microcontrollers is an impediment to their use in projects they are otherwise well suited for. Proprietary tools and programming interfaces require longer learning curves and result in non-portable software which increases the cost of using them.

DEBROS provides a way to get the benefits of a familiar standards-based programming API and runtime environment on inexpensive hardware, making them a practical alternative in many cases.

REFERENCES

[1] Andrew S. Tanenbaum, "Operating Systems, Design and Implementation," Prentice-Hall, Inc. (1987).

[2] W. Richard Stevens, "UNIX Network Programming", Prentice-Hall, Inc (1990).

[3] Daniel P. Bovet and Marco Cesati, "Understanding the Linux Kernel", O'Reilly Media, Inc. (2006)