

Use an RTOS on Your Next MicroBlaze-Based Product

An RTOS called $\mu\text{C}/\text{OS-II}$ has been ported to the MicroBlaze soft-core processor.

by Jean J. Labrosse
President
Micrium, Inc.
Jean.Labrosse@Micrium.com

Designing software for a microprocessor-based application can be a challenging undertaking. To reduce the risk and complexity of such a project, it's always important to break the problem into small pieces, or tasks. Each task is therefore responsible for a certain aspect of the application. Keyboard scanning, operator interfaces, display, protocol stacks, reading sensors, performing control functions, updating outputs, and data logging are all examples of tasks that a microprocessor can perform.

In many embedded applications, these tasks are simply executed sequentially by the microprocessor in a giant infinite loop. Unfortunately, these types of systems do not provide the level of responsiveness required by a growing number of real-time applications because the code executes in sequence. Thus, all tasks have virtually the same priority.

For this and other reasons, consider the use of a real-time operating system (RTOS) for your next Xilinx MicroBlaze™ processor-based product. This article introduces you to a low-cost, high-performance, and high-quality RTOS from Micrium called $\mu\text{C}/\text{OS-II}$.

What is $\mu\text{C}/\text{OS-II}$?

$\mu\text{C}/\text{OS-II}$ is a highly portable, ROM-able, scalable, preemptive RTOS. The source code for $\mu\text{C}/\text{OS-II}$ contains about 5,500 lines of clean ANSI C code. It is included on a CD that accompanies a book fully describing the inner workings of $\mu\text{C}/\text{OS-II}$. The book is called *MicroC/OS-II, The Real-Time Kernel* (ISBN 1-5782-0103-9) and was written by the author (Figure 2). The book also contains more than 50 pages of RTOS basics.

Although the source code for $\mu\text{C}/\text{OS-II}$ is provided with the book, $\mu\text{C}/\text{OS-II}$ is not freeware, nor is it open source. In fact, you need to license $\mu\text{C}/\text{OS-II}$ to use it in actual products.

Micrium's application note AN-1013 provides the complete details about the porting of $\mu\text{C}/\text{OS-II}$ to the MicroBlaze soft-core processor. This application note is available from the Micrium website at www.micrium.com.

$\mu\text{C}/\text{OS-II}$ was designed specifically for embedded applications and thus has a very small footprint. In fact, the footprint for $\mu\text{C}/\text{OS-II}$ can be scaled based on which $\mu\text{C}/\text{OS-II}$ services you need in your application. On the MicroBlaze processor, $\mu\text{C}/\text{OS-II}$ can be scaled (as shown in Table 1) and can easily fit into Xilinx FPGA block RAM.

$\mu\text{C}/\text{OS-II}$ is a fully preemptive real-time kernel. This means that $\mu\text{C}/\text{OS-II}$ always attempts to run the highest-priority task that is ready to run. Interrupts can suspend the execution of a task and, if a higher-priority task is awakened as a result of the interrupt, that task will run as soon as the interrupt service routines (ISR) are completed.

$\mu\text{C}/\text{OS-II}$ provides a number of system services, such as task and time management, semaphores, mutual exclusion semaphores, event flags, message mailboxes, message queues, and fixed-sized memory

partitions. In all, $\mu\text{C}/\text{OS-II}$ provides more than 65 functions you can call from your application. $\mu\text{C}/\text{OS-II}$ can manage as many as 63 application tasks that could result in quite complex systems.

$\mu\text{C}/\text{OS-II}$ is used in hundreds of products from companies worldwide. $\mu\text{C}/\text{OS-II}$ is also very popular among colleges and universities, and is the foundation for many courses about real-time systems.

	Code (ROM)	Data (RAM)
Minimum Size	5.5 Kb	1.25 Kb (excluding task stacks)
Maximum Size	24 Kb	9 Kb (excluding task stacks)

Table 1 – Memory footprint for $\mu\text{C}/\text{OS-II}$ on the MicroBlaze processor

RTOS Basics with $\mu\text{C}/\text{OS-II}$

When you use an RTOS, very little has to change in the way you design your product; you still need to break your application into tasks. An RTOS is software that manages and prioritizes these tasks based on your input. The RTOS takes the most important task that is ready to run and executes it on the microprocessor.

With most RTOSs, a task is an infinite loop, as shown here:

```
void MyTask (void)
{
    while (1) {
        // Do something (Your code)
        // Wait for an event, either:
        // Time to expire
        // A signal from an ISR
        // A message from another task
        // Other
    }
}
```

You're probably wondering how it's possible to have multiple infinite loops on a single processor, as well as how the CPU goes from one infinite loop to the next. The RTOS takes care of that for you by suspending and resuming tasks based on events. To have the RTOS manage these tasks, you need to provide at least the following information to an RTOS:

- The starting address of the task (`MyTask()`)
- The task priority based on the importance you give to the task
- The amount of stack space needed by the task.

With $\mu\text{C}/\text{OS-II}$, you provide this information by calling a function to “create a task” (`OSTaskCreate()`). $\mu\text{C}/\text{OS-II}$ maintains a list of all tasks that have been created and organizes them by priority. The task with the highest priority gets to execute its code on the MicroBlaze processor. You determine how much stack space each task gets based on the amount of space needed by the task for local variables, function calls, and ISRs.

In the task body, within the infinite loop, you need to include calls to RTOS functions so that your task waits for some event. One task can ask the RTOS to “run me every 100 milliseconds.” Another task can say, “I want to wait for a character to be received on a serial port.” Yet another task can say, “I want to wait for an analog-to-digital conversion to complete.”

Events are typically generated by your application code, either from ISRs or issued by other tasks. In other words, an ISR or another task can send a message to a task to wake that task up. With $\mu\text{C}/\text{OS-II}$, a task waits for an event using a “pend” call. An event is signaled using a “post” call, as shown in the pseudo-code below:

```
void EthernetRxISR(void)
{
    Get buffer to store received packet;
    Copy NIC packet to buffer;
    Call OSQPost() to send packet to task
    (this signals the task);
}

void EthernetRxTask (void)
{
    while (1) {
        Wait for packet by calling OSQPend();
        Process the packet received;
    }
}
```

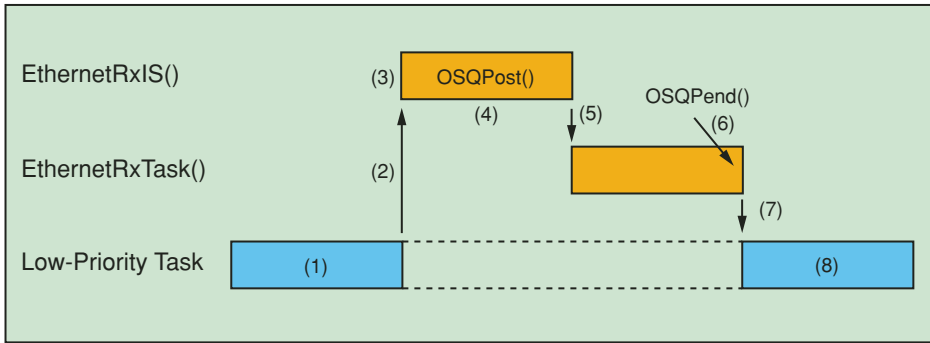


Figure 1 – Execution profile of an ISR and $\mu\text{C}/\text{OS-II}$

The execution profile for these two functions is shown in Figure 1.

Let's assume that a low-priority task is currently executing (1). When an Ethernet packet is received on the Network Interface Card (NIC), an interrupt is generated. The MicroBlaze processor suspends execution of the current task (the low-priority task) and vectors to the ISR handler (2).

With $\mu\text{C}/\text{OS-II}$, the ISR starts by saving all of the MicroBlaze registers, taking a mere 300 nanoseconds at 150 MHz. The ISR handler then needs to determine the source of the interrupt and execute the appropriate function (in this case, `EthernetRxISR()`) (3).

This ISR obtains a buffer large enough to hold the packet received by the NIC. It copies the contents of the packet received into the buffer and sends the address of this buffer to the task responsible for handling received packets (`EthernetRxTask()` in our example) using $\mu\text{C}/\text{OS-II}$'s `OSQPost()` (4).

When this operation is completed, the ISR invokes $\mu\text{C}/\text{OS-II}$ (calls a function) and $\mu\text{C}/\text{OS-II}$ determines that a higher-priority task needs to execute. $\mu\text{C}/\text{OS-II}$ then resumes the more important task and doesn't return to the interrupted task (5). With $\mu\text{C}/\text{OS-II}$ on the MicroBlaze processor, this takes about 750 nanoseconds at 150 MHz.

When the packet is processed, `EthernetRxTask()` calls `OSQPend()` (6). $\mu\text{C}/\text{OS-II}$ notices that there are no more packets to process and suspends execution of this task by saving the contents of all MicroBlaze registers onto that task's stack. Note that when the task is suspended, it doesn't consume any processing time.

$\mu\text{C}/\text{OS-II}$ then locates the next most important task that's ready to run (the "Low-Priority Task" in Figure 1) and switches to that task by loading the MicroBlaze registers with the context saved by the ISR (7). This is called a *context switch*. $\mu\text{C}/\text{OS-II}$ takes less than 1 microsecond at 150 MHz on the MicroBlaze processor to complete this process. The interrupted task is resumed exactly as if it was never interrupted (8).

The general rule for an application using an RTOS is to put most of your code into tasks and very little code into ISRs. In fact, you always want to keep your ISRs as short as possible. With an RTOS, your system is easily expanded by simply adding more tasks. Adding lower-priority tasks to your application will not affect the responsiveness of higher-priority tasks.

Safety-Critical Systems Certified

$\mu\text{C}/\text{OS-II}$ has been certified by the Federal Aviation Administration (FAA) for use in commercial aircraft by meeting the demanding requirements of the RTCA DO-178B standard (Level A) for software used in avionics equipment. To meet the requirements for this standard, it must be possible to demonstrate through documentation and testing that the software is both robust and safe.

This is particularly important for an operating system, as it demonstrates that it has the proven quality to be usable in

any application. Every feature, function, and line of code of $\mu\text{C}/\text{OS-II}$ has been examined and tested to demonstrate its safety and robustness for safety-critical systems where human life is on the line.

$\mu\text{C}/\text{OS-II}$ also follows most of the Motor Industry Software Reliability Association (MISRA) C Coding Standards. These standards were created by MISRA to improve the reliability and predictability of C programs in critical automotive systems.

Your application may not be a safety-critical application, but it's reassuring to know that the $\mu\text{C}/\text{OS-II}$ has been subjected to the rigors of such environments.

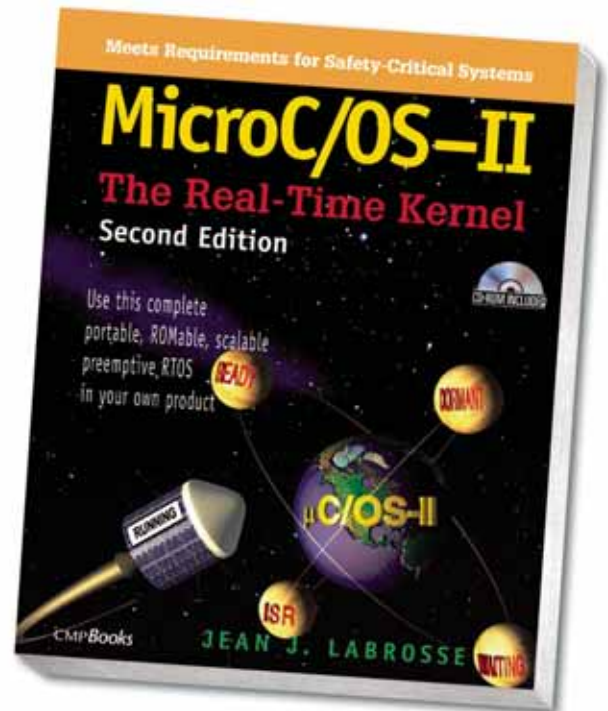


Figure 2 – *MicroC/OS-II* book cover

Conclusion

An RTOS is a very useful piece of software that provides multitasking capabilities to your application and ensures that the most important task that is ready to run executes on the CPU. You can actually experiment with $\mu\text{C}/\text{OS-II}$ in your embedded product by obtaining a copy of the *MicroC/OS-II* book. You only need to purchase a license to use $\mu\text{C}/\text{OS-II}$ for the production phase of your product. 🌟