Software-Only Hardware Simulation

Simulating embedded hardware in a Windows environment can significantly reduce development time. In this article, Michael provides techniques for the software-only simulation of embedded hardware. He presents a simple example of an RTOS-less embedded system that uses memory-mapped I/O to access a UART-like peripheral to serially poll a slave device. The simulator is capable of detecting bugs and troublesome design flaws.

In this article, I will describe techniques for the software-only simulation of embedded hardware in the Windows/PC environment. Software-only simulation implies an arrangement with which the embedded application, or parts of it, can be compiled and run on the Windows platform (host) talking to the software simulator as opposed to the real hardware. This arrangement doesn’t require any hardware or tools other than a native Windows development toolset such as Microsoft Developer Studio/Visual C++. Importantly, the same source code is compiled and linked for both the host and the target. It’s possible and often necessary to simulate more complex aspects of the embedded target such as interrupts and the RTOS layer. However, I will illustrate the basics of simulating hardware in the Windows environment with an example of an extremely simple hypothetical target system (see Figure 1).

Assuming that the source code of the embedded application is basically the same whether it runs in Windows or the embedded target, the simulation offers several advantages. You have the ability to develop and debug device drivers and the application before the hardware is ready. An extremely powerful test harness can be created on the host platform, where all code changes and additions can be verified prior to running on the actual target. The harness can be used as a part of software validation. Furthermore, you have the ability to test conditions that may not be easy to test using the real hardware.

In the vast majority of cases, debugging tools available on the host are far superior to those offered by cross-development tool vendors. You have access to runtime checkers to detect memory leaks, especially for embedded software developed in C++. Lastly, note that where the final system comprises a number of CPUs/boards, simulation has the additional advantage of simulating each target CPU via a single process on a multitasking host.

FIRST THINGS FIRST

Before you decide to invest in simulation infrastructure, there are a few things to consider. For instance, when the target hardware is complex, the software simulator becomes a fairly major development task. Also, consider the adequacy of the target development tools. This especially applies to debuggers. The absence, or insufficient capability, of the debugger on the target presents a strong case for simulation.

When delivery times are more critical than the budget limitations and extra engineering resources are available, the additional development effort may be justified. The simulator may help to get to the final product faster, but at a higher cost. You should also think about whether or not it’s possible to cleanly separate the application from the hardware access layer. Remember that when exact timings are a main design concern, the real-time aspects of the target are hard to simulate, so the simulator will not help. Moreover, the embedded application’s complexity is relatively minor compared to the hardware drivers, so the simulator may not be justified. However, when the application is complex and sitting on top of fairly simple hardware, the simulator can be extremely useful.

You should also keep in mind that when it’s likely that the software application will be completed before the hardware delivery date, there is a strong case for simulation. (If only to keep software engineers from getting bored and, more seriously, to familiarize them with the hardware before it arrives.)

In the majority of cases, simulation on the host does not mean that target debugging and testing are completely eliminated. Inevitably, the behavior of the target differs from even the most complete simulator. Some amount of target debugging and testing is required, but it may be significantly reduced by simulation.
Another important point is that it may not be feasible at times to simulate the entire target system. Nevertheless, it still can be worthwhile to simulate a part of the system. In this case, only some of the embedded target software modules would be included in the host builds.

There is a case for which simulation is particularly advantageous: when the target has multiple CPUs and interaction between the microprocessors is being developed. For instance, you may be developing software for a system comprised of similar boards communicating with each other over the Ethernet. Every change in software would require a separate download to each board. Also, should there be an issue with interboard communications, it would be nearly impossible to set up and simultaneously debug software in multiple target boards. This is not to mention the cost of multiple ICE and debuggers. On the host, however, it is simply a matter of starting as many applications as needed, with each application simulating a single target board.

**COMMON TERMS**

I assume you’ll use a C or C++ development environment for this project. The computer running Windows/UNIX is the host. The embedded target is the target. The term “process” is used in relation to UNIX processes and Windows applications. The term “thread” refers to both p-threads on UNIX and the Windows API standard thread library. Although the first example is effectively a DOS console application, I still refer to it (correctly or not) as running in the Windows operating environment. Perhaps it also should be noted that Visual C++ does not generate 16-bit code (not since the second version anyway), so it will not run on pure DOS machines. However, it will run quite happily in a DOS box on Windows.

Before moving on to the actual simulator, let’s take a step back and have a look at why the two main cross-development platforms (UNIX and derivatives versus Windows/PC) have such different approaches to the software simulation of the target hardware.

**UNIX**

It appears that on the UNIX (and derivative) cross-development platform, especially in the military and telecommunications industries, the development teams are much more likely to spend significant effort on the creation of a proper simulation harness. There are several reasons for this. Keep in mind that the same tools (GNU) are often used for both host and cross-compilation. In addition, the target and host builds are typically controlled by the same GNU make-driven build infrastructure, so you can easily determine which modules are included in a build. Also, the projects tend to be larger and run in a more formal fashion.

Telecommunications products are often composed of a number of boards, which, in the final product, are likely to talk to each other via a TCP/IP stack. This maps well into the simulation platform for the entire system on the host, where each target CPU can be simulated via a standalone UNIX process. Finally, the target RTOS is often UNIX-inspired or even fully POSIX-compliant. As such, it is neatly mapped into a UNIX lightweight thread library running within the same process on the host.

After this fairly neat infrastructure is created, it’s completely natural to extend the ability of the host builds to run as much of the target application as possible. This can be achieved by different levels of simulation of the hardware present on the target boards.

**PC/WINDOWS**

When the Windows/PC platform is used for embedded software development, the situation is completely different. You will most likely use a proprietary and hopefully well-featured toolset for the project’s microcontroller. The set would typically include an IDE with a cross compiler/assembler/linker/locator, a symbolic debugger, and an evaluation board with the target CPU.

---

**Listing 1—Take a look at the declaration and implementation of the simulated and real hardware access layer.**

```c
//****************************************************************************
// Header file
#ifndef SIM
// Simulator
extern BYTE read_hw_byte(volatile BYTE *addr);
extern void write_hw_byte(BYTE data, volatile BYTE *addr);
#else
// Real target
#define read_hw_byte(x) (*(volatile BYTE *)x)
#define write_hw_byte(d, x) (*((volatile BYTE *)x) = (d))
#endif

//****************************************************************************
// Simulator implementation
BYTE read_hw_byte(volatile BYTE *addr)
{
  intercept_read(addr);
  return *addr;
}

void write_hw_byte(BYTE data, volatile BYTE *addr)
{
  *addr = data;
  intercept_write(addr);
}
```

**Figure 2—There is a parallel among the embedded target and the UNIX and Windows environments. Again, equivalent entities are shown on the same level.**

---

www.circuitcellar.com
on it. All of this allows you to get up and running quickly.

Often, only critical parts of the application are initially developed in the host environment, and then the module is ported across to the target. (The ability to run the code under the host is often lost after porting.) For instance, if your target system features text-driven screens, it would be completely logical to develop and debug a screen module on your desktop and then port it across. This is as far as most projects would go with host development.

Because of the dissimilarity between the target-specific cross compiler and toolset and native Windows development tools, the division between the host and the target software is much greater. That’s why developers rarely build into their designs the ability to run embedded applications in Windows.

The other equally important reason is the perceived fundamental dissimilarity between Windows and the embedded RTOSs and also between their application interfaces (API). However, Windows provides support for threads that can provide an adequate simulation platform for an embedded RTOS. As on the UNIX platform, it is possible to map the embedded environment in the Windows host environment as shown in Figure 2.

Note that this is a neat parallel because Windows threads share the memory space of the parent process. As such, they are ideal to represent the target RTOS. The processes in Windows have separate memory space and can communicate only with each other via the special means of interprocess communication, which is not unlike the target CPUs/boards.

SOFTWARE DESIGN GUIDE

Now let’s focus on what makes embedded software adaptable for simulation. It’s hardly surprising that the following guidelines closely resemble those for writing portable code.

First, you need a centralized access mechanism to the hardware (read_hw and write_hw macros). Second, the application code and device driver code must be separated. Third, you must use a thin operating level interface. Finally, avoid using the nonstandard add-ons that some cross-compilers may provide.

HARDWARE INTERFACE LAYER

In this example, I assume that all the peripherals are interfaced via memory-mapped I/O, although the techniques I described are equally applicable to port-mapped I/O or a combination of the two. (Obviously, the hardware interface macros for the two types of I/O will be different.) Also, for the sake of simplicity, I assume that all accesses to the hardware are byte-wide. Again, it is clear that all other types of accesses (16- and 32-bit) are easy to implement via simple extensions of the technique discussed.

The code fragment in Listing 1 illustrates the real and simulated hardware interface layer. The compile-time preprocessor flag SIM differentiates the host builds from the target builds.

The other equally important reason is the perceived fundamental dissimilarity between Windows and the embedded RTOSs and also between their application interfaces (API). However, Windows provides support for threads that can provide an adequate simulation platform for an embedded RTOS. As on the UNIX platform, it is possible to map the embedded environment in the Windows host environment as shown in Figure 2.

Note that this is a neat parallel because Windows threads share the memory space of the parent process. As such, they are ideal to represent the target RTOS. The processes in Windows have separate memory space and can communicate only with each other via the special means of interprocess communication, which is not unlike the target CPUs/boards.

SOFTWARE DESIGN GUIDE

Now let’s focus on what makes embedded software adaptable for simulation. It’s hardly surprising that the following guidelines closely resemble those for writing portable code.

First, you need a centralized access mechanism to the hardware (read_hw and write_hw macros). Second, the application code and device driver code must be separated. Third, you must use a thin operating level interface. Finally, avoid using the nonstandard add-ons that some cross-compilers may provide.

HARDWARE INTERFACE LAYER

In this example, I assume that all the peripherals are interfaced via memory-mapped I/O, although the techniques I described are equally applicable to port-mapped I/O or a combination of the two. (Obviously, the hardware interface macros for the two types of I/O will be different.) Also, for the sake of simplicity, I assume that all accesses to the hardware are byte-wide. Again, it is clear that all other types of accesses (16- and 32-bit) are easy to implement via simple extensions of the technique discussed.

The code fragment in Listing 1 illustrates the real and simulated hardware interface layer. The compile-time preprocessor flag SIM differentiates the host builds from the target builds.

For the application and device drivers, any accesses to the hardware must be performed via these macros (target) or functions (host). It is no longer permissible to access the memory-mapped I/O hardware directly via constructs such as the following:

```c
byte data = *(byte *)addr;
```

and

```c
*(byte *)addr = data;
```

Use the following instead:

```c
byte data = read_hw_byte(addr);
```

and

```c
write_hw_byte(data, addr);
```

Note that there are no conditional compilation statements around any of the hardware access lines of code. This means cleaner and maintainable source code.

The overhead on the target is kept to

A Visual C++ Environment

Before getting started, you need to install Visual Studio or .Net software on your PC. Using the downloadable Visual Studio project is straightforward. First, copy the SimPolledUart directory to a suitable location. (A new folder in DevStudio/MyProjects is a good idea.) Second, open Microsoft Developer Studio Version 5 or 6. Next, open the SimPolledUart workspace. Finally, press the F5 key to compile and run the program without leaving the IDE.

Creating a new project from the provided source code isn’t too difficult either. Start by opening Developer Studio Version 5 or 6. Then, select File, New, and Win 32 Console Application.

In the Project Name and Location sections, select a convenient location for your project. Use the following example as a guide:

- **Project:** SimPolledUart
- **Location:** C:\Embedded

Visual Studio creates a SimPolledUart directory for you.

Next, close the project and copy the following seven files to the SimPolledUart directory: Hw.h, Sim_Uart.c, Sim_uart.h, Main.c, EmbTypes.h, Uart.c, and Uart.h. Select Build and Set Active Configuration [this should be set to Debug]. Then, select Project, Settings, C/C++, and add flag SIM at the end of the Preprocessor Definitions box. This will ensure that the SIM preprocessor flag is defined.

After you select Project, Add To Project, and Files—and after the file dialog is displayed—select and add all seven source files to the project. Lastly, press the F5 key. This will compile and run the program without leaving the IDE.
a minimum because macros are used (no function call). The overhead is not important on the host, and each hardware access will be translated into a function call. In the C++ environment, read_hw_byte and write_hw_byte can be inlined if required.

All of the hardware access is channeled via two simple macros/functions. Some advantages are immediately obvious. Let’s say all the memory-mapped I/O is addressable at even addresses only. By simply adding an ASSERT in the host version of read_hw_byte and write_hw_byte, accesses at odd addresses are caught easily. Similarly, you could verify that addresses used for memory-mapped I/O accesses are in the valid range and catch all attempts to write to an illegal address. If such an illegal access is detected on the host, it is simply a matter of backtracking to the offending piece of code.

**RUNNING WITH VISUAL STUDIO**

This project was developed using Microsoft’s Developer Studio/Visual C++ environment, which is today’s most popular development environment for creating Windows applications. Contrary to popular belief, you don’t need to be an expert in Windows API, MFC, COM, and the plethora of other mysterious acronyms to use this environment. In fact, you can still create a console Hello World application in Visual C++ by simply writing:

```cpp
int main()
{
    printf("Hello world\n\r");
}
```

This console application will compile and run in a DOS box. The first simulator also happens to be a console application.

If you already have Visual Studio installed, you can download the source code and Visual C++ project files from the Circuit Cellar ftp site, and then load them into Visual Studio. Refer to the sidebar for instructions on setting up your Visual C++ environment. You can use Visual C++ version 5 or 6. I’m pretty sure you can use the .NET environment (it’s backward-compatible with Visual Studio), but I haven’t tried it myself.

**HYPOTHETICAL APP**

The rest of this article is written in a way that allows you to follow the topic even if you don’t have a PC in front of you. (This is not to suggest that this article is a good bedtime read.)

The example is extremely simple. The target hardware board has a UART. It acts as a master in a simple communications protocol, where it polls a fairly dumb slave using an ASCII text string, receives the response, and verifies that it is as expected. The master sends a string [poll] to the slave, which responds with the original string and the - echo string appended at the end. The master makes sure that the received string is indeed the original string plus - echo. The master software must be developed and verified to work correctly with the slave (see Figure 3).

The uselessness of this application is completely understood. Actually, it helps to illustrate the core techniques rather than the application-specific aspects. Moreover, you will see that things can
go wrong, even in this absurdly simple system. A simulator can catch early bugs.

For simplicity, only one thread exists. The simulator and the real application run in the same thread, as does the simple user interface. Normally, with real-world examples, you would need to create a separate thread in which the simulator runs.

The application and simulator then become completely independent and the two interact via three memory-mapped I/O registers, as shown in Figure 4.

Figure 5 is in Real Time/Structured Analysis and/or Structured Design (RT/SASD) notation. Threads/processes are shown with oval shapes. Data stores via parallel lines.

The UART-like interface has three registers: a receive register, a transmit register, and a status register. They are all byte-wide. The receive register is read-only. The transmit register is write-only. The status register is both read and write.

As on normal UARTs, the receive register contains data from when the status register has RX_FULL bit set. When the TX_EMPTY bit is set in the status register, the transmit register can accept new data. Also, I'll add a twist here by saying that writing a zero to the status register initializes it in this hypothetical UART.

Let's take another look at the hardware interface layer and discuss the `intercept_read` and `intercept_write` simulator functions used in the simulated environment (shown in sim_uart.c, which can be downloaded from the Circuit Cellar ftp site). These two functions are the only hooks between the application and the simulator. There is only one thread. Calls to the intercept functions kick the simulator. This is the only chance it gets to analyze the address and data written and read, and do some processing. In the case of more complex and realistic simulators, the simulator would run in its own execution thread and would perhaps maintain the data using a state machine of its own.

In a larger and more realistic system where multiple simulated devices could coexist, `write_hw_byte` and `read_hw_byte` would work out which simulator to call for a particular address access detected. For some simple examples of how this mechanism could be implemented, read the comments in the source code posted on the Circuit Cellar ftp site.

The `intercept_read` function puts data produced by the simulator in the register pointed to by `addr`, possibly modifies its contents, and then returns the contents of the `addr`. `intercept_read` analyzes the address being read and puts data in the register according to its internal logic. In this case, it puts the next character required to send to the master.

In the case of a write access, note that the `write_hw_byte` function writes data to the register first. Only then is `intercept_write` called. This is so that the simulator is aware of what data is written. The `intercept_write` function is allowed to modify the data in the register after the write action [see Listing 2].

For example, think of a register with a couple of reserved bits that will always read zero no matter what is written to them. This is illustrated by requiring that the UART is initialized by first writing a zero to it. When a write of zero to the status register is detected, the simulator sets the transmitted empty bit, which starts off the transmissions. In other words, immediately after a write to it, the UART status register contains a byte different than the byte that has been just written to it. (Sounds just like real hardware!)

**CODE STRUCTURE**

The hw.h file contains definitions for I/O-mapped hardware access included by the real target and simulator. It declares hardware access layer macros (real target) or functions (simulator). All modules include this header file.

The `sim_uart.c` file is the simulated UART implementation, and it contains the source code for the simulator. It is only included in the simulated/host environment. To emphasize the point, it is fully surrounded by the `#ifdef SIM` preprocessor flag. This illustrates the point much better than showing some obscure make file listings. The SIM preprocessor flag is defined only on the host and not on the target. The initialization function changes the pointers used to access the real memory mapped hardware to point to simulators own storage registers.

The simulator uses the `sim_data` array to buffer the received data and reuses this buffer to transmit the response. It remembers the number of bytes transmitted and received. It assumes that it can start transmitting without waiting for the end of the string from the master.

The `intercept_uart_write` function expects a write either to the transmit register or the status register. A write to any other location in memory-mapped I/O is flagged as an error. After zero is written to the status register, the status register always has the TX_EMPTY bit set. [Remember, this is how the obscure UART is initialized.] The `intercept_uart_read` function makes sure that read is either from the receive register, or the status register, before analyzing the data. The `sim_user_interface` function allows you to change the string used to poll the slave. To change the poll string, press any key on the keyboard and enter the string. Pressing the Escape key followed by the Enter key or `Ctrl+C` will exit the application.

The `sim_uart.h` file is the simulated UART header file for the simulator.
Wireless + Control

Control and communicate with your devices from anywhere in the world with Z-World’s M2M solution.

Imagine devices notifying you when they need servicing; change settings from around the world; upload data wirelessly; send commands from one machine to another; receive vehicle location/conditions via email.

These are all examples that combine control with wireless communications.

Z-World’s M2M Application Kit utilizes GSM/GPRS and SMS wireless technologies to provide the tools for designing a wireless communications link to a Rabbit-based device.

Z-World is a proven leader in the embedded controls industry. Let Z-World’s M2M solution add wireless capabilities to your next control project.

In partnership with: www.zworld.com/wireless
888.362.3387
Naturally, it is not used by the real UART driver code.

Main.c contains the code entry point, initializations, and the forever loop. It includes the start-up code and is used for both simulated and target builds. As such, it has a few lines of code surrounded by #ifdef SIM. The structure of this file would be similar to a much larger and a more realistic project’s main.c file.

The main.c file has three sections in the file. In the first section, all simulators should be initialized first. This section is surrounded by the #ifdef SIM flag. The second main.c section is followed by calls to device drivers/thread initializations. A forever loop follows. Add a single, simulator-specific thread called sim_user_interface that allows you to add a degree of interactivity to the simulator. Naturally, the simulators are initialized first to detect any hardware accesses performed during real initialization. Also, note that the amount of code surrounded by the #ifdef SIM preprocessor directive will not change, and the structure of the file remains clear.

The embtypes.h file includes common types and definitions. It contains the usual collection of typedefs [e.g., INT and WORD]. In addition, it defines an EMB_ASSERT macro, which works just as ASSERT, but is named differently to avoid confusion with Windows ASSERT.

Uart.c contains the actual polled UART thread/driver. The initialization of pointers is included so that the application doesn’t have an uninitialized pointer even if the SIM flag hasn’t been defined on the host:

```c
volatile BYTE *uart_rx_reg = (volatile BYTE *)&foo; //TODO
```

Clearly on the target, it has to be set to point to a hardware specific memory-mapped I/O address. Note that on simulated builds, the simulator changes these pointers.

The tx_msg is initialized to contain a string with a single character (H). The process_response simply compares the reply string with what’s expected and prints an OK or Error message.

The serial_comms function is built around a simple state machine that can be in one of three states. In the idle state, the tx_count is initialized. In the transmitting state, the data is sent byte by byte to the slave (written to the transmit register) until the null-terminator character is sent. When null character is sent, the state machine switches to the receiving state and reads the data from the receive register. The driver looks for TX_EMPTY_BIT and RX_FULL_BIT in the status register when required (as on a normal polled UART driver). The header files are self-explanatory.

Uart.h is the embedded header file for UART definitions (e.g., register layouts). It contains various definitions for UART bit layouts, and also declares externs required. This is a part of normal embedded software, and the simulator uses it too.

**BUG FINDING**

Visual Studio has two build modes, Release and Debug. The SIM flag is defined for Debug mode builds only, so make sure that you are building the debug version of the executable.

You build and run the application by pressing F5 in Visual C++. You’ll get the following output (repeated many times because polling repeats continuously):

```
Send: H, Received H- echo
```

You can change the string sent to the slave by inputting the string followed by <CR>.

Now, I’ll demonstrate how the simulator can catch silly bugs. To do so, I’ll intentionally introduce a few bugs to the code.
The source code has three highlighted points for your attention. Search for the word “point” in the code to locate these highlights.

The first point illustrates that the simulator will catch the incorrect sequence of incrementing the array index. The following portion of uart.c is correct:

```c
rx_msg[rx_count] = read_hw_byte(uart_rx_reg);
if (rx_msg[rx_count++] == 0)
```

Now let's introduce a bug incrementing the receive counter one step too early:

```c
rx_msg[rx_count++] = read_hw_byte(uart_rx_reg);
if (rx_msg[rx_count] == 0)
```

If you recompile and run the program, you get the following output:

Send: H, Received H
EMB: Response incorrect, expected H - echo

You'll quickly realize that there is a flaw in the program because the null character was erroneously detected too early, which tricked the software into assuming that the slave response was complete. Return the source code to the original bug-free state before moving on to the second point in the uart.c file.

The second point shows that should you forget to initialize the UART by writing zero to the status register, the communications would never start. This is because `TX_EMPTY_BIT` is never set and the transmission from the master never commences.

Comment out the following line in the file main.c:

```c
write_hw_byte(0, uart_status_reg);
```

Then, recompile and run. There is no output in the window. The communications with the slave are not running.

Put the initialization line back in and let’s go to the last and most interesting point. Refer to the sim_uart.c file. This file shows that a simulator can catch a fairly high level, almost a design level problem. In this example, the software design of the actual driver assumed that the communications are half-duplex. In other words, the software assumes that the reception from the slave will start only after the poll message has been fully transmitted. This is effectively how the state machine is written.

On the other hand, the design of the simulator assumes [rightfully] that the communications can be full duplex, and it makes sense to start sending the response string back without wait-

---

**Figure 6**—Notice the difference between the half-duplex and full-duplex (shown here) communications protocol between the master and slave. The slave transmission starts while the master is still transmitting.

---

Copyright Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar, Inc. is prohibited.
PROTEUS
The Complete Electronics Design System

NEW SIMULATION
Models Include
MicroChip PIC 18
Basic Stamp BS1 & BS2
& PIC Basic Plus*

Schematic & PCB Layout
- Powerful & flexible schematic capture.
- Auto-component placement and rip-up/retry PCB routing.
- Polygonal gridless ground planes.
- Libraries of over 8000 schematic and 1000 PCB parts.
- Bill of materials, DRC reports and much more.

Mixed Mode SPICE Circuit Simulation
- Berkeley SPICE3F5 simulator with custom extensions for true mixed mode and interactive simulation.
- 6 virtual instruments and 14 graph based analysis types.
- 6000 models including TTL, CMOS and PLD digital parts.
- Fully compatible with manufacturers’ SPICE models.

New Features in Version 6
- Drag and drop toolbars.
- Visual PCB packaging tool.
- Improved route editing.
- Point and click DRC report.
- Multiple design rules (per net).
- Multiple undo/redo.

Proteus VSM - Co-simulation and debugging for popular Micro-controllers
- Supports PIC, AVR, 8051, HC11 and ARM micro-controllers.
- Co-simulate target firmware with your hardware design.
- Includes interactive peripheral models for LED and LCD displays, switches, keypads, virtual terminal and much, much more.
- Provides source level debugging for popular compilers and assemblers from Crownhill, IAR, Keil, and others.

Call Now for Upgrade Pricing
* PIC BASIC Plus from Crownhill Associates

Starting as low as
$199
With Manual on CD

Labcenter Electronics
Free DEMO CD  www.labcenter-electronics.com
Contact - R4 Systems at 905 898-0665 or info@r4systems.com
Michael Melkonian studied Electronics Engineering in Moscow and completed his B.S. in 1988. For the past 15 years, he has worked as a software engineer. Currently, he is a senior software engineer at Chubb Electronic Security, which is based in Sydney, Australia. Michael's technical interests include embedded systems and communications protocols, as well as RTOS and GUI development. In his spare time, he enjoys making computer-animated 3-D movies (no major hits so far) and watching soccer games. You may contact him at mmelkonian@tpgi.com.au.

The reason you didn’t see errors before is that the slave only starts sending its response after it receives the first two characters (see Listing 3).

If you think about it, the response o - echo makes sense because the driver misses all the characters that the slave transmits while the master is transmitting too. I failed to implement the driver in a full-duplex capable way because in the implementation it can either transmit or receive, but not both. The simulator caught the design flaw.

A VIABLE OPTION

In many cases, simulating the embedded target hardware on the Windows platform is a viable option that significantly reduces development time. This is especially true when the software development has commenced before the target hardware is available, and when the complexity of the application is significant and the interface to peripheral hardware relatively simple. Being able to compile and run target code for PCs and Windows offers access to a set of excellent debugging and runtime-checking tools that may not be available on the embedded platform.

Simulators can range from simple console-driven applications (as shown in this article) to complete simulators modeling the real-time behavior of the target, interrupts, and target RTOS. Embedded RTOS threads can be mapped into Windows threads. In case of multiprocessor systems, separate Windows processes [applications] can be used to model interaction between the target processor/boards.

In this article, I presented a simple example of an RTOS-less embedded system that uses memory-mapped I/O to access a UART-like peripheral that is used to serially poll a slave device. Even such a simple simulator is capable of detecting both trivial bugs and serious design flaws.

Listing 3—After the first 2 bytes are received by the simulated UART, it starts “transmitting” by putting characters in the receive register and setting the Receiver Full bit in the status register.

```c
if (sim_rx_byte_count > 2)
{
    sim_uart_status_reg |= RX_FULL_BIT;
    sim_uart_rx_reg = sim_data[sim_tx_byte_count++];
}
```

Michael Melkonian studied Electronics Engineering in Moscow and completed his B.S. in 1988. For the past 15 years, he has worked as a software engineer. Currently, he is a senior software engineer at Chubb Electronic Security, which is based in Sydney, Australia. Michael's technical interests include embedded systems and communications protocols, as well as RTOS and GUI development. In his spare time, he enjoys making computer-animated 3-D movies (no major hits so far) and watching soccer games. You may contact him at mmelkonian@tpgi.com.au.