WIZnet 2010 iMCU DESIGN CONTEST

SPECIAL FEATURE

by Dave Tweed

iMCU W7100

Embedded Networking Made Simple

The hardware TCP/IP stack of the W5100 has been enhanced in the W7100 with the addition of an on-chip 8051 application processor core, eliminating the need for a separate processor chip in many applications. Here's an introduction to the new chip and an evaluation module that's based on it.

Ethernet connectivity for embedded systems has been a hot topic for a while now, and WIZnet has a nice family of products that makes Ethernet and TCP/IP accessible to any microprocessor that has at least an SPI interface. Their latest offering, the W7100 chip, takes it one step further by integrating a general-purpose 8051 CPU core onto the same die, creating the possibility of truly single-chip implementations for many low-end applications.

This article will take you through some of the details of the new chip and the development tools for it, and then show you a complete application—a GPS-disciplined Internet time server—that takes advantage of its features.

THE W7100 CHIP

The W7100 chip is a combination of the same hardware TCP/IP core used in the W5100 along with a high-performance 8051-compatible CPU core. The TCP/IP core includes 32 KB of data buffer memory and supports eight simultaneous sockets. In addition to the standard 8051 features, the CPU core includes 64 KB of XDATA memory (SRAM), 256 bytes of nonvolatile XDATA memory (flash), 64 KB of code memory (flash), and 2 KB of boot code memory (ROM) (see Figure 1).

The TCP/IP core in the W7100 has basically the same functionality as the standalone W5300 chip. However, instead of an SPI or parallel interface, it uses a dual-port memory arrangement with the CPU core that can support higher performance. Both the registers and the buffer memory of the TCP/IP core are mapped into the 0xFExxxx block of the CPU core's 24-bit XDATA memory space, and a special routine (called `wizmemcpy()`) is provided in the boot ROM that supports a high-speed memory-to-memory transfer between TCP/IP core memory and CPU memory.

Just to give you an idea of the levels of performance you can expect, I tried out the WIZnet-supplied TCP loopback server example. This is a simple server that sets up all eight sockets in TCP mode, listening on port 5000. Any data received on any socket is immediately sent back to the originator. WIZnet also supplies a desktop program called AX1 to communicate with the server. It has the

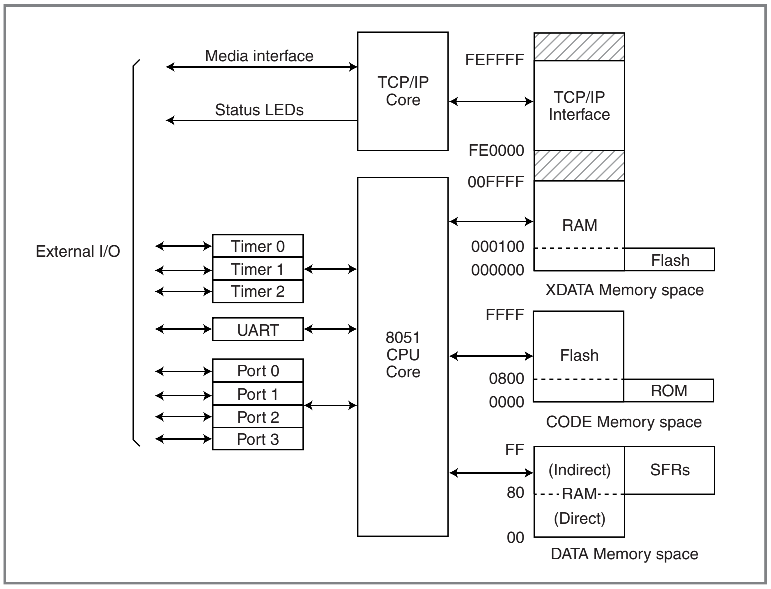

Figure 1—This shows two types of information, the block diagram of the W7100 chip along with information about how the 8051 memory spaces are laid out.

ability to send a file to the loopback server and measure the overall throughput.

Right out of the box, this setup achieved about 1.6 Mbps overall, transferring a 1-MB file in about 5 seconds. However, I took a look at the code, and it turns out that for every packet received, it was sending some debug information out the UART port, and this turned out to be slowing things down. When I removed the diagnostic messages, the throughput approximately doubled, to about 3.3 Mbps for the same size file. In the sample application that we'll get into later on, I've left the loopback server in place on the unused sockets so that you can see this for yourself.

The processor core itself is a fairly generic implementation with a moderate amount of on-chip I/O, including one UART, three timers, and plenty of GPIO. It has the extensions required to support 24-bit XDATA memory space, including two 24-bit DP registers for memory-to-memory transfers.

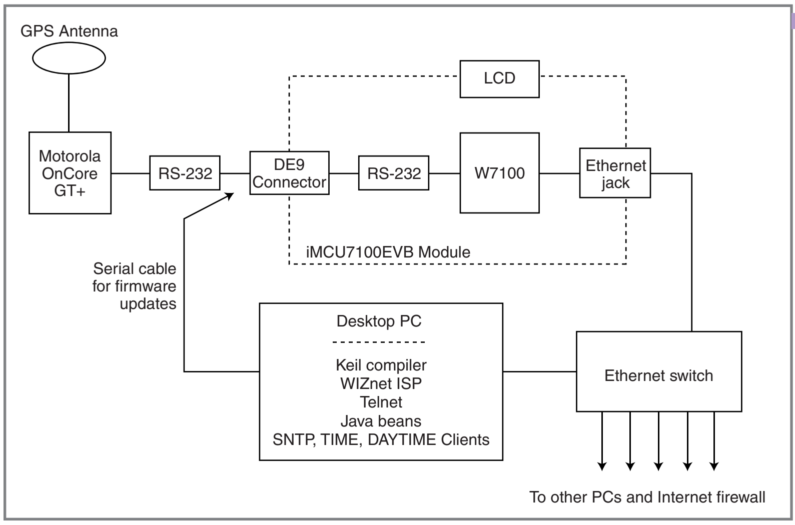

Figure 2—The hardware setup includes the iMCU7100EVB module along with the Motorola OnCore GT+ GPS receiver module. The PC supports both code development and operational testing.

The 64-KB code memory space is completely occupied by on-chip flash memory, plus there's a 2-KB ROM that can be overlaid over part of that space. There's a dedicated "boot mode" pin that determines the initial code memory configuration of the chip—whether it starts by executing the boot loader in ROM or goes directly to the user application in flash.

SOFTWARE DEVELOPMENT TOOLS

The WIZnet folks recommend using the Keil suite of 8051 software development tools (C compiler and assembler, along with their "µVision" IDE), and as it happened, I already had a copy of them installed from another project several years ago, so I was all set.

Each of the demonstration projects comes with a µVision project file, but I ended up setting up a Makefile and building the software from a Cygwin command line. It's probably just my old-school mentality showing through, but generally the only thing I use IDEs for is simulating or debugging. For anything else, they just get in the way.

I was hoping to try out some alternative software tools, such as SDCC, but I ran out of time and didn't get a chance to investigate that. However, based on my observations with the Keil tools, it doesn't look like there's anything in the W7100's CPU that can't be programmed with fairly generic tools.

DEVICE PROGRAMMING & DEBUGGING

The evaluation kit I received has two hardware development interfaces and PC-side software packages. The first is a simple in-system programmer for getting your code into the chip. There's a serial-port bootloader built into the on-chip ROM, and a cable is provided to connect that to a hardware port on your PC. A simple PC application takes your hex file and gets it into the code flash. It can also program the small data flash area if you want.

The second tool is a JTAG-based debugger interface. It comprises a board with a fairly hefty FPGA on it, presumably for better performance. It connects to the PC via USB, and to the target via a small header. Unfortunately, I didn't have enough time to check out this tool.

THE iMCU7100EVB

The iMCU7100EVB evaluation module (mine says iMCU7100API in the silkscreen) includes the W7100 chip and an Ethernet connector (with built-in magnetics), along with an RS-232 level translator for the UART. All of the chip's external I/O is brought out to pads to which you can solder either 0.100" or 2-mm headers, and a special connector along one edge connects to the included 2 × 16 LCD module. There's also an array-of-pads prototyping area that supports both 0.100" and 2-mm grids. (As you may recall, 2-mm headers were used for the W5100-based module used in the 2007 iEthernet Design Contest, causing issues for some contestants. Obviously, WIZnet took that into account here.)

LEDs are provided both for the dedicated status outputs of the TCP/IP core, and for general use by application code on the CPU. A DIP switch sets the Ethernet operating mode, and there are other switches for Power, Reset, and Boot mode.

SAMPLE APPLICATION

The sample application is an idea borrowed from the 2007 WIZnet iEthernet Design Contest, which featured the W5100. Contestant Steven Nickels put together an Ethernet Time Server using the WIZnet module coupled with a Freescale microcontroller and a WWVB receiver module. It served up time in three ways, supporting the SNTP, TIME, and DAYTIME protocols. This time around, I'll use the W7100's built-in CPU and a GPS receiver module.

Steven's project only kept track of time down to the second, which makes sense for several reasons. First of all, it's tricky to get more than that level of precision from a WWVB receiver because of the nature of the 1-bps signal. Also, the TIME and DAYTIME protocols only have 1-second resolution anyway.

On the other hand, a GPS receiver can provide sub-microsecond precision on its pulse per second (PPS) output (typically down to ±50 ns in position-hold mode), and the NTP packet structure has timestamps with a resolution of 2^{-32} second (about 230 ps). I've always been interested in precision timekeeping and frequency standards, so I'm going to design my project to not only implement the basic time-server functionality, but also support eventual construction of a full NTP server and a GPS-disciplined reference oscillator.

THE REQUIREMENTS

The hardware requirements for this project are simple. I have some Motorola OnCore GT+ GPS receiver modules that I purchased some time ago. That defines that side of the implementation—the W7100 is going to have to communicate with one of these modules using its binary protocol. The CPU will get the OnCore status messages via its serial port from the receiver, along with the 1-PPS timing signal on a GPIO pin, providing potential accuracy down to the microsecond level.

On the LAN (software) side, we'll be running the TIME, DAYTIME, and SNTP protocol servers, plus a Telnet-based console interface of my own devising that has turned out to be a big help during debugging. Also, keeping in mind the future development of a high-precision system, the software timebase will need a mechanism that allows it to take into account any inaccuracy in the CPU's own clock. More about this when we discuss the `time` module.

A few things to keep in mind for the future would be to add a simple web server for configuration, a DCHP client for getting IP configuration information, and perhaps an external hardware VCXO (voltage-controlled crystal oscillator) that would allow the system to be used as a GPS-disciplined precision timing reference. These are beyond the scope of this article, but they're definitely things I'm interested in exploring soon.

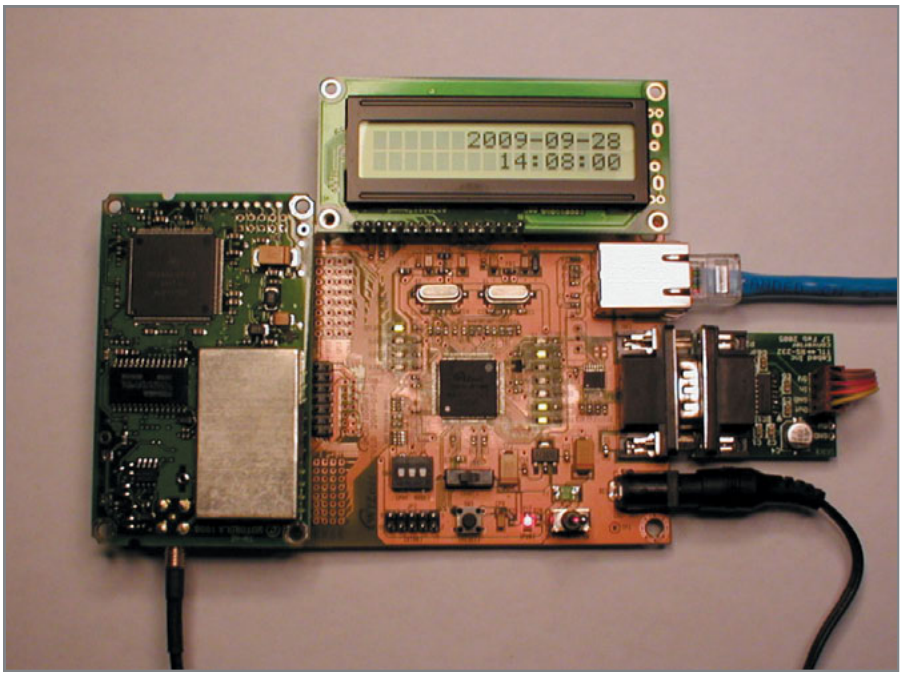

Photo 1—The W7100 chip in the center, which runs the show, is surrounded by the GPS receiver module on the left, the 2 x 16 alphanumeric LCD above (this comes with the evaluation module), and a small RS-232 level converter on the right.

THE DESIGN—HARDWARE

The hardware design is straightforward. Figure 2 shows a block diagram of the overall system. Once the GPS receiver is married to the WIZnet module (power, serial port, and PPS), the only external interfaces are the antenna connection to the receiver, the Ethernet connection, and the WIZnet module's power supply (a wall wart).

I just needed to add a 10-pin female header to the prototyping area to support the OnCore module. The only quirk stems from the fact that the OnCore serial interface uses TTL signal levels, while the WIZnet board only supports RS-232—there's no provision in the PCB artwork for disabling or bypassing the RS-232 level converter. As a result, I needed to add a small TTL-to-RS232 converter module in order to prototype this system.

The wall-wart power supply that comes with the WIZnet board provides regulated 5.0 VDC, and an on-board linear regulator drops this down to 3.3 V for the W7100. Both 5.0 V and 3.3 V are brought out to pads near the prototyping area, so I got the 5 V that the OnCore module requires there.

Photo 1 shows the entire system.

THE DESIGN—SOFTWARE

The software design is more involved, but we'll borrow heavily from the WIZnet sample code and Steven's original implementation.

First, let me say a few words about how the source code is structured. I'm a firm believer in top-down, modular design, abstraction and information hiding. Over the years, I've developed a scheme for structuring source code that helps reinforce those concepts.

Each software module implements a single logical piece of functionality, such as a low-level UART interface or a higher-level message protocol. To the greatest extent possible, each module presents an application programming interface (API) that is self-contained and hides all details about the underlying implementation.

I like to use short module names, and then prefix each of the global items belonging to that module (data types, shared data, and function names) with the name of the module. This makes it immediately obvious when reading some other module where to go to get more information about any item I see.

Take the UART interface as a specific

Listing 1—The header file for the sio module (sio.h) exposes only the interfaces that other modules need. All implementation details are hidden in the code file (sio.c). Yes, this module was indeed first developed in 1992, and I've been using it ever since!

```
/* sio.h */

/* Interrupt-based SIO driver for general breadboard use. */

/* History:
 *  2009/09/13 DT   add PARITY_NONE (8-bit data mode)
 *  2009/09/12 DT   tweak data types for W7100 project
 *                   add baud rates supported by W7100
 *  1992/11/24 DT   add 'sio_puthex', 'sio_put_ulong' and
 *                   'sio_status'
 *  1992/11/23 DT   started
 */

void sio_init (void);

#define B110     0
#define B300     1
#define B1200    2
#define B2400    3
#define B4800    4
#define B9600    5
#define B19200   6
#define B38400   7
#define B57600   8
#define B115200  9
#define B230400 10
#define B460800 11
void sio_set_baud (uint8 flag);

#define PARITY_SPACE 0
#define PARITY_MARK  1
#define PARITY_EVEN  2
#define PARITY_ODD   3
#define PARITY_NONE  4
void sio_set_parity (uint8 flag);

void sio_putc (char ch);
void sio_puts (char *s);
void sio_puthex (uint8 n);
void sio_put_ulong (uint32 n);

char sio_getc (void);
bool sio_status (void);
```

example. Typically, an application program is going to want to send bytes to the interface, see if bytes are available in the interface, and get those bytes if so. It also may need to configure the interface in terms of things like bit rate, parity, flow control, etc. However, the rest of the application code doesn't—and shouldn't—care whether the underlying implementation is polled or interrupt-driven, what kinds of hardware/software buffering might be going on, or what register bits to twiddle to configure the port.

Therefore, the .h (header) file for the sio module only exposes an abstract set of functions and constants that the application code can use to manipulate the interface in exactly those ways (see Listing 1). Note that unlike a lot of other coders (embedded and otherwise), I have *not* put details about hardware register addresses and bit field definitions into this file—those are implementation details that only need to be known by the corresponding .c (code) file. They either get defined directly in that file, or indirectly by virtue of including a different relevant header file.

Many embedded applications have multiple things going on in parallel, yet they don't really require the complex interactions among threads that the typical RTOS (real-time operating system) supports. Often, a simple "main loop" that calls the different tasks in round-robin sequence is more than sufficient, and avoids many of the pitfalls of interrupt-driven thread switching in the first place. I call this technique "pseudo-multithreading," and it has worked well for me for over 20 years.

With that in mind, take a look at the overall structure of the software for this project, as shown in Figure 3. The main module serves only to get the system initialized, and then it enters an infinite loop, in which it calls the "go" function for each module that has one. In this case, we have six such modules: the five socket servers—tp, dtp, sntp, loopback, and console—and the timebase module (time).

The remaining modules perform support functions, called as needed by those six. The lcd module puts ASCII information on the LCD, and the sio module implements the UART driver. The socket module provides the abstract logical interface to the WIZnet TCP/IP core, while the wiz module hides the low-level details of talking to a particular implementation. The wizmemcpy module encapsulates the special high-speed memory-to-memory copy function used on the W7100 chip. The oncore and fifo modules support the console module by implementing the receiver-specific message processing and a generic FIFO function, respectively.

We can establish some specific lines of communication among the modules that are required for this project. For example, each of the time server modules needs to be able to get the current time from the time module, in addition to servicing its assigned socket via the socket module. The loopback module has no connections other than the one to the socket module.

The console module has several

connections. In addition to the aforementioned support modules, it has a socket interface running a Telnet server (on port 23) for general debugging, it can call into the time module in order to set or adjust the system clock, and it uses the sio module to communicate with the GPS receiver. The latter interface can also be used for debugging when the receiver is not connected, which is useful for debugging details of the TCP/IP interface.

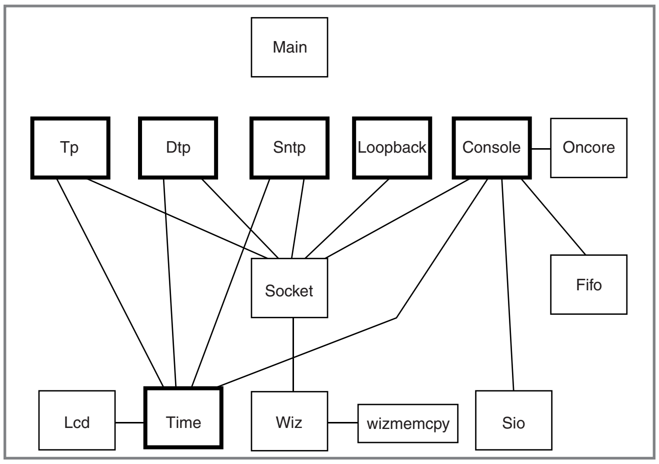

Figure 3—The software is broken up into modules. The ones with heavy borders represent the top-level "threads" that run concurrently, called in round-robin fashion by the main module. The others are support libraries and low-level drivers. The lines between them show how they communicate.

SOCKET INTERFACE

I started out by looking at the implementation of the TCP loopback server supplied by WIZnet, since three of the four servers I wanted to implement would involve TCP. The "TCPS" project as supplied by them is broken into three layers, with the loopback module at the top, a socket abstraction in the middle, and an iinchip module providing the low-level interface to the TCP/IP core.

I reviewed the source code and felt there was a lot of information shared among the three layers. For example, the iinchip module provided functions to read and write 8-bit registers in the interface, but no support for the several 16-, 32-, and 48-bit registers—the socket module had long strings of 8-bit reads and writes to deal with them instead.

So, partly for that reason, and partly to force myself to examine and understand all of the code, I started rewriting both modules in my own style and tweaking the interface between them. The first thing I did was to rename the iinchip module to wiz, and to start putting the wiz_ prefix on all the function names. This would allow the compiler to help me catch anything I might otherwise miss translating.

I created functions like wiz_read16() and wiz_write16() (along with 32- and 48-bit versions) and made the corresponding changes in socket, which made the overall logic of that module much clearer. Along the way, I discovered that some of the registers had dedicated access functions, and this led me to the fact that the driver can use an interrupt from the TCP/IP core to pick up certain status changes, but not all. It turns out that the driver must explicitly poll the hardware for each packet send or receive operation, *without* using the status-interrupt mechanism. This caused quite a bit of head-scratching until I discovered this detail.

I also made a pass through the loopback module itself, which implements the top-level state machine for any TCP server. You can use this module as a template for any TCP-based service, and I have in fact left it in place on the otherwise unused sockets in this design.

THE CONSOLE

The next thing I implemented was a generalized console (debug) interface. I knew that at first, I would be using the UART port for debugging some of the TCP/IP code, but then I would later need to devote this port to the GPS receiver, and so it seemed logical to provide a Telnet server that provided the same kind of access.

Doing this helped reinforce the knowledge I picked up while studying the loopback module. In addition, rather than using the extremely-simple polled UART driver code that WIZnet used, I pulled out my tried-and-true interrupt-based 8051 UART driver (called sio) that I developed back in the early 1990s while working on some commercial telecom-industry firmware. It is completely interrupt-driven, with large FIFOs in each direction, and supports all the baud rates and all the parity modes for 7-bit data. The only tweaks I needed for this project were to add some of the higher bit rates that the W7100 supports, and the PARITY_NONE mode to support the 8-bit binary data used in the OnCore interface.

The console module can accept data from either the UART or its Telnet socket, and it can send diagnostic output messages to either or both paths as well. Any of the other modules can send diagnostic messages by calling console_print(), and they don't need to know which path is actually in use at the time. An internal flag tells console whether the UART is being used for diagnostics, and this flag can be set/cleared on the fly by calling console_enable_sio().

At the moment, the console module is probably the messiest one in terms of its internal logic, and it also is the one that will change the most as the project evolves. In its present state, console_print() only goes to the Telnet connection, any data received via Telnet is translated into binary form and forwarded to the OnCore module via the UART, and any data coming from the OnCore module is converted to readable ASCII form and forwarded to the Telnet connection. In addition, if the message from the OnCore module is recognized as a status message (starting with "@@Ea"), it is parsed into a data structure, and then the time and date fields from this structure are used to set the timebase.

I also retained the LCD interface from the original TCPS project. It shows some start-up information, but then the time module takes it over and displays the current date and time, updated every second.

THE TIMEBASE

The software I've described up to this

USING TELNET

Using the Telnet protocol (RFC854) to connect to your project is very straightforward. Pretty much every operating system has a command-line Telnet client—usually called "telnet"—and most GUI-based terminal emulators support Telnet as well.

To get started, just get to a command prompt on your desktop system and type "telnet <host>," where <host> is either an IP address or a host name that is known to your system. For example:

```
# telnet 192.168.1.20
Trying 192.168.1.20...
Connected to 192.168.1.20.
Escape character is '^]'.
```

From then on, everything you type will be sent to the remote system on a line-by-line basis each time you hit <CR>, and anything the remote system sends back will be displayed.

Make note of the escape character; that's how you'll get out when you're done. It isn't the same thing as the Escape key—that would be '^['—you really have to hit Ctrl-]. At that point, you'll get a prompt from the client program on the local system, and you can type "quit" to terminate the session or "help" for additional commands.

point can be characterized as generic infrastructure code that would be applicable to pretty much any application. Here's where we start to get into the details of the time server application in particular. There are two parts to this: setting up a timebase based on the CPU clock (accessed by means of the hardware timer modules) and setting/calibrating that timebase using data found in the OnCore GPS messages.

Ultimately, the CPU's crystal is the timing reference for the timebase. On the W7100, the 11.0592-MHz crystal frequency is multiplied by eight to get a raw CPU clock of 88.4736 MHz. (You might recall that 11.0592 MHz is a convenient value for generating standard UART bit rates.) The raw CPU clock gets divided by 12 (7.3728 MHz) to create the clock that drives the hardware timers.

I reserved Timer 1 to generate the UART bit rate clock, so that left Timers 0 and 2 for use in the application timebase. I eventually want to use Timer 2 to accurately capture the PPS signal from the GPS receiver, which leaves Timer 0 for generating a fundamental "tick" interrupt that can be used to measure the passage of time. It turns out that the most convenient tick rate (i.e., one that's an integer multiple of 1 Hz) that I can get using this combination of clock frequency and the divider ratios available in Timer 0 is 900 Hz.

One thing we're going to have to keep in mind is that the 11.0592-MHz crystal is just a generic unit, with probably on the order of ±100 ppm accuracy. Since I eventually want to be able to establish a "virtual" timebase that's a couple of orders of magnitude better than this (on the order of 1 ppm or better), I need a mechanism that will allow the passage of time per software tick to be adjusted by small amounts. I borrowed the technique used in direct digital synthesis (DDS) frequency generators. It works as follows.

I maintain three variables to record the passage of time: a 32-bit picosecond counter, a 16-bit millisecond counter, and a 32-bit seconds counter. I also have a variable called `ps_per_tick`, which is initialized to a particular value, but can be adjusted on the fly. With a nominal tick rate of 900 Hz, there should be 1,111,111,111 ps per tick. This is a number that just fits into a 32-bit variable. For each tick interrupt that occurs, the `ps_per_tick` value gets added to the picosecond accumulator. Then, as long as the picosecond accumulator is greater than 1,000,000,000, that value is subtracted from the accumulator and the millisecond accumulator is incremented. This will happen once or twice per tick, depending on the starting value of the picosecond accumulator. Finally, each time the millisecond counter reaches 1,000, it gets cleared and the seconds counter gets incremented. The seconds counter simply counts seconds from the start of January 1, 1900—it will overflow sometime in the year 2036.

You can see that this setup allows 1-LSB adjustments of the `ps_per_tick` value to vary the perceived rate of time by about 1 ppb, which is more than enough resolution (about 32 ms per year) to reach my goals. After experimenting with this for a while, I discovered that the crystal on my particular board runs about 80 ppm fast, (gaining almost 7 seconds per day); so for now, I initialize `ps_per_tick` to 1,111,022,229 and leave it there. It currently keeps time on its own to better than 0.5 s per day.

The next part of the problem is to get the counters set to the correct value, based on the information coming from the GPS receiver. The `oncore` module (software) takes care of the details of communicating with the OnCore module (hardware) using its binary protocol. There are several useful functions here: `oncore_create()` takes a "generic ASCII" representation of an OnCore message (one that can be typed by a user) and turns it into the "pure binary" form that the OnCore expects, while `oncore_process()` does the opposite. These are useful for testing the interface. The specific message we're interested in is the "`@@Ea`" status message, so there are two functions specific to that: `oncore_parse_Ea()` reads the contents of that message and puts the information into a C structure for use by the other modules, and `oncore_show_Ea()` prints the contents of that structure to the console for monitoring what's going on. It's actually the `console` module that pulls the date and time information out of that structure and then calls `time_set()` to synchronize the software timebase with the real world.

For now, that's all I'm doing—forcing the seconds counter to the value that represents the same time that's in the GPS message. I'm not (yet) making any attempt to synchronize the picosecond and millisecond counters to the 1-s boundaries, which means that there's still up to 1 s of difference between internal time and external time. The next step will be to use the rising edge of the PPS signal coming from the GPS module to take care of that detail. Eventually, I'll be setting up a software phase-locked loop (PLL) that drives the software timebase into exact alignment with the PPS signal by dynamically adjusting the `ps_per_tick` value. This will also give me a more precise measurement of the CPU crystal's frequency error.

THE TIME SERVERS

With the software timebase set up, it's actually quite straightforward to implement the time server modules themselves. Both TIME protocol and DAYTIME protocol are TCP services, so I took the generic TCP state machine from the TCPS `loopback` module, and then dropped Steven's data-handling code into them, creating the `tp` and `dtp` modules, respectively. SNTP protocol is UDP-based, so I went to the WIZnet UDP loopback example to get the template for the `sntp` module, and put Steven's packet-building code into it, making suitable adjustments.

Steven had some Java client code for all three protocols that runs on a PC that he used to test his server, and I figured that a fair test of my implementation would be to see whether it works with those clients. After getting the latest versions of Java and Java Beans from the Sun website, I was able to adjust the hard-coded IP addresses and compile the clients. Everything worked just fine!

I figured the real acid test would be to see whether a Windows machine would actually be willing to synchronize with my server (all versions from Windows 2000 on have SNTP built in). It turned out that Steven had some of the timestamps in the wrong places in his SNTP packet, but after a simple adjustment, my Win2K machines were happy with the setup. Also, I took advantage of my millisecond counter to add some fractional-second information to the timestamps, which makes it easier to see how well things are tracking.

FUTURE DIRECTIONS

I hope that you will find some of the modules in the code accompanying this article a useful base for your own W7100 projects. In terms of this particular project, I'm not sure if the Motorola OnCore series of GPS receivers is still available on the surplus market, but it should be straightforward to replace the `oncore` module with an NMEA sentence parser to allow the use of most other GPS receiver modules.

As I said before, I plan to continue development of this project to support precision timing and frequency, and if I come up with something interesting, I'll write a follow-up article. I'd also like to add additional TCP/IP features to the project, such as a DHCP client and a simple HTTP server. I've seen some interesting work regarding the use of client-side Javascript to create relatively rich web interfaces for embedded systems that I'd like to explore. ■

David Tweed (dtweed@acm.org) is a hardware and real-time firmware engineering consultant who has been working with embedded processors starting in 1976 with the Intel 8008. His system design experience includes computer design from supercomputers to workstations, digital telecommunications systems, and the application of embedded microcomputers and DSPs. He is also a Circuit Cellar *project editor and quiz master. When not playing with electronics and software, he pursues his hobby as an amateur musician, playing keyboards and low brass instruments in several community groups.*

PROJECT FILES

To download the code and additional content, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2009/233.

RESOURCES

D. Mills, "RFC2030: Simple Network Time Protocol," Network Working Group, 1996.

Motorola, OnCore Manual, www.wa5rrn.com/oncore.htm.

S. Nickels, "Time Server Design: Synchronize with the WWVB Time Code Signal," *Circuit Cellar* 220, 2008.

———, Time Server Project, www.circuitcellar.com/Wiznet/winners/001066.html.

J. Postel, "RFC867: Daytime Protocol," Network Working Group, 1983.

J. Postel and K. Harrenstien, "RFC868: Time Protocol," Network Working Group, 1983.

WIZnet, "Internet Embedded MCU W7100 Datasheet," Ver. 0.9 Beta, 2009.

WIZnet Wizwiki, http://wizwiki.net/forum/.

SOURCES

GNU Tools on Windows
Cygwin | www.cygwin.com

RSLink Module
Embed, Inc. | www.embedinc.com/products/ser/

8051 Compiler tool
IAR Systems | www.iar.com
Keil | www.keil.com

Java Beans
Sun Microsystems | www.java.sun.com

W7100 Evaluation module/kit
WIZnet | www.wiznet.co.kr