www.circuitcellar.com

# CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#121 AUGUST 2000

## EMBEDDED PROGRAMMING

Software Testing Made Easy

Build a Webcam For $55

Developing Software Without A Target

A Free ANSI-C Compiler

$3.95 U.S. ($4.95 Canada)

# CIRCUIT CELLAR ONLINE

Double your technical pleasure each month. After you read *Circuit Cellar* magazine, get a second shot of engineering adrenaline with *Circuit Cellar Online*, hosted by ChipCenter.

## — FEATURES —

### Looking at the Specs
*Gerard Fonte*

With this article, Gerard helps us answer some of the questions about one of the defining characteristics of most engineering projects, showing us that, when engineers, marketeers, and customers all have a common goal, they can get the most out of specifications. But, is it really possible for all three to come to an agreement? See what Gerard has to say.

### Build a Virtual Wireless Automation System
*Michael Chan*

What is "real" and what is virtual reality? It's hard to tell nowadays. Similarly, Michael's project scheme is wireless, but is it really? It seems like it because there is little or no wiring or installation involved when you want to make a change to your system. Let Michael take you through the steps, then maybe you can get virtually wireless.

### Decisions, Decisions...
### Choosing the Right Technology
*George Novacek*

Are you creative? Well, if you are the engineer who has to make the decision concerning a certain design, then you had better be. George discusses some of the more important aspects of decision making and shows us that, by understanding the problems and options available, we can decrease the risk and choose the right alternative.

## — COLUMNS —

### Lessons from the Trenches
#### Number Crunching with Embedded Processors
*George Martin*

This month, George walks us through binary numbers, embedded processors, and how different numbering systems work together with CPU instructions and the C language. He takes us through the steps and then encourages us to walk on our own with different examples. And remember, practice makes perfect!

### Silicon Update Online
#### I'm a Traveling Man
*Tom Cantrell*

What season is it, anyway? I would say summer, but for Tom and many others it's conference and trade show season. So, pack your suitcase, kiss the spouse goodbye, and get ready for the ride because Tom is taking us on the road with him. Our first stop—FPGAs.

### Learning the Ropes
#### Virtex Proto Board
*Ingo Cyliax*

Although Ingo had originally planned to follow up his last article about multipliers, he ran into a little problem. OK, so it was a big problem. But, that's to our benefit.

WWW.CIRCUITCELLAR.COM/ONLINE

Table of Contents for July 2000

### Resource Links
- Obtaining CE Marking Certification for Products
- Industrial Laser Applications & Sources

*Rick Prescott*

### Test Your EQ
8 Additional Questions

## INTERNET PIC® 2000 CONTEST

WINNERS!

★ PIC Web CAM ★

★ PICX-10 Web Server
★ Web Chart Recorder
★ IDAPIC
★ UDP/IP Interface
★ SLIP Server
★ Environment Sensing

# INSIDE ISSUE 121


12 **Simplify Your Software Testing**
*Jonathan Valvano*

20 **Look Ma, No PC!**
A $55 Webcam
*Steve Freyder, David Helland, & Bruce Lightner*

30 **Anatomy of a Compiler**
A Retargetable ANSI-C compiler
*Sandeep Dutta*

36 **The Joys of Writing Software**
Part 1: Battle of the Bug
*George Novacek*

58 **Who Needs Hardware?**
Developing Without the Target
*Alan Harry*

68 **Count the Digits**
Designing a Frequency Meter
*Tom Napier*

74 **From the Bench**
Building on Familiar Ground
Part 1: Adding Analog to the 8051 Core
*Jeff Bachiochi*

78 **Silicon Update**
We Ride the Waves
*Tom Cantrell*

**Task Manager** 6
Rob Walker
Coming Events

**New Product News** 8
edited by Harv Weiner

**Reader I/O** 11

**Test Your EQ** 85

**Advertiser's Index** 95
**September Preview**

**Priority Interrupt** 96
Steve Ciarcia
First on the Block


## EMBEDDED PC


41 **Nouveau PC**
*edited by Harv Weiner*

43 RPC **Real-Time PCs**
Real-Time Executive for Multiprocessor Systems
Part 4: Debugging
*Ingo Cyliax*

49 APC **Applied PCs**
Embedded Kiosk or Mission Impossible?
*Fred Eady*

## Coming Events

**b**elieve it or not, the summer of 2000 is coming to a close. All in all, 2000 has turned out to be an interesting year thus far. Most of January was spent on sighs of relief and repeats of "I told you so." Then reality set in—what does one do with a six-month supply of canned meat and bottled water? Although the media never covered it, SPAM, SPAM, bottled water, SPAM became a frequent addition to the menu in many households. All of that processed meat got people thinking that maybe life in the year 2000 was going to be a lot more like life in the late 1900s after all.

So, we set about to make a year of it. Unfortunately, it was already late spring, so we had some catching up to do. In the summer of 2000, science and technology advanced with the mapping of the human DNA and the US government's release of the latest method of secure hard drive storage. (Using the BTC [behind the copier] hard drive storage method, you too can keep high-security files as safe and protected as they would be at Los Alamos.)

In the theater, we've weathered the perfect storm, accepted another mission impossible, and watched as one summer blockbuster after another was gone in sixty seconds. For some quality late summer entertainment though, pop up a bag of microwave popcorn, pour yourself a cup of ice cold soda, and sit down with this issue of *Circuit Cellar*. You'll get more than 146 minutes of entertainment for half the cost of a trip the theater—and you can smoke, talk, or leave your cell phone turned on if you so choose.

While Tom Cruise was hanging off cliffs, turn to page 49 and you'll find that Fred Eady was busy hanging two touchscreens off of one embedded computer to create an embedded kiosk application. Bruce Lightner, David Helland, and Steve Freyder directed an impressive sequel to their "A $25 Web Server" article (*Circuit Cellar Online*, July, 1999) by adding a basic digital camera to their web server to create a $55 webcam (p. 20). And, if all this talk of fame and fortune has you thinking that you wouldn't mind racing through the countryside in a sporty roadster and having your name up in lights, look across to page 7 where you'll find the details for the latest design contest from Circuit Cellar and Zilog.

In sports, the summer of 2000 has seen legends retire and new ones step into the spotlight. At Pebble Beach, Tiger Woods broke a 138-year old record and once again proved that analysts sometimes say the dumbest things. How could Tiger be bad for the game of golf?

If the title of this editorial had you looking for all kinds of what-the-future-holds information, obviously you didn't learn the Y2K lesson—hype kills. You'll get to 2001 before you know it, and chances are, you'll find out that it's quite similar to life in 2000. But, I must admit, there is one coming event that is generating enough interest to mention.

It's an age-old quest for glory. The dream of representing their country, standing on the platform, and waving to the masses has kept these contestants training for most of their lives. For some, it is the chance to step out of the shadows and become number one. For others, it is an opportunity to carry on the family name. These contestants have put away the controlled substances and now must begin the final push.

Yeah, nothing beats the pomp and circumstance of the American presidential race. Let the games begin.

rob.walker@circuitcellar.com


# CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS



**EDITORIAL DIRECTOR/PUBLISHER**
Steve Ciarcia

**MANAGING EDITOR**
Rob Walker

**TECHNICAL EDITORS**
Jennifer Belmonte
Rachel Hill
Jennifer Huber

**WEST COAST EDITOR**
Tom Cantrell

**CONTRIBUTING EDITORS**
Mike Baptiste
Ingo Cyliax
Fred Eady
George Martin
George Novacek

**NEW PRODUCTS EDITORS**
Harv Weiner
Rick Prescott

**PROJECT EDITORS**
Steve Bedford
James Soussounis
David Tweed

**ASSOCIATE PUBLISHER**
Joyce Keil

**CHIEF FINANCIAL OFFICER**
Jeannette Ciarcia

**CUSTOMER SERVICE**
Elaine Johnston

**ART DIRECTOR**
KC Zienka

**GRAPHIC DESIGNERS**
Melissa Clukey
Mary Turek

**STAFF ENGINEERS**
Jeff Bachiochi
Anthony Capasso

**QUIZ MASTER**
David Tweed

**EDITORIAL ADVISORY BOARD**
Ingo Cyliax
Norman Jackson
David Prutchi



Cover photograph Ron Meadows—Meadows Marketing
**PRINTED IN THE UNITED STATES**



### ADVERTISING

**ADVERTISING SALES REPRESENTATIVE**
Kevin Dows
(860) 872-3064
Fax: (860) 871-0411
E-mail: kevin.dows@circuitcellar.com

**ADVERTISING COORDINATOR**
Valerie Luster
(860) 875-2199
Fax: (860) 871-0411
E-mail: val.luster@circuitcellar.com

**ADVERTISING CLERK** Sally Collins



### CONTACTING CIRCUIT CELLAR

SUBSCRIPTIONS:
**INFORMATION:** www.circuitcellar.com or subscribe@circuitcellar.com
**TO SUBSCRIBE:** (800) 269-6301, www.circuitcellar.com/subscribe.htm, or subscribe@circuitcellar.com
**PROBLEMS:** subscribe@circuitcellar.com

GENERAL INFORMATION:
**TELEPHONE:** (860) 875-2199 **FAX:** (860) 871-0411
**INTERNET:** info@circuitcellar.com, editor@circuitcellar.com, or www.circuitcellar.com
**EDITORIAL OFFICES:** Editor, Circuit Cellar, 4 Park St., Vernon, CT 06066

AUTHOR CONTACT:
**E-MAIL:** Author addresses (when available) included at the end of each article.

For information on authorized reprints of articles,
contact Jeannette Ciarcia (860) 875-2199 or e-mail jciarcia@circuitcellar.com.

CIRCUIT CELLAR®, THE MAGAZINE FOR COMPUTER APPLICATIONS (ISSN 1528-0608) and Circuit Cellar Online are published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (860) 875-2751. Periodical rates paid at Vernon, CT and additional offices. **One-year (12 issues) subscription rate USA and possessions $21.95, Canada/Mexico $31.95, all other countries $49.95. Two-year (24 issues) subscription rate USA and possessions $39.95, Canada/Mexico $55, all other countries $85.** All subscription orders payable in U.S. funds only via VISA, MasterCard, international postal money order, or check drawn on U.S. bank.

**Direct subscription orders and subscription-related questions to Circuit Cellar Subscriptions, P.O. Box 5650, Hanover, NH 03755-5650 or call (800) 269-6301.**

**Postmaster:** Send address changes to Circuit Cellar, Circulation Dept., P.O. Box 5650, Hanover, NH 03755-5650.



Circuit Cellar® makes no warranties and assumes no responsibility or liability of any kind for errors in these programs or schematics or for the consequences of any such errors. Furthermore, because of possible variation in the quality and condition of materials and workmanship of reader-assembled projects, Circuit Cellar® disclaims any responsibility for the safe and proper function of reader-assembled projects based upon or from plans, descriptions, or information published by Circuit Cellar®.

The information provided by Circuit Cellar® is for educational purposes. Circuit Cellar® makes no claims or warrants that readers have a right to build things based upon these ideas under patent or other relevant intellectual property law in their jurisdiction, or that readers have a right to construct or operate any of the devices described herein under the relevant patent or other intellectual property law of the reader's jurisdiction. The reader assumes any risk of infringement liability for constructing or operating such devices.

Entire contents copyright © 2000 by Circuit Cellar Incorporated. All rights reserved. Circuit Cellar and Circuit Cellar INK are registered trademarks of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

# NEW PRODUCT NEWS

Edited by Harv Weiner

## DATA LOGGER

Designed for educational use, the **DrDAQ** data logger features built-in sensors for light and temperature and a microphone for sound. It plugs into a PC's parallel port for display or data gathering. Any pH probe can be used for acid or base measurements, and fast signals can be captured like sound waveforms. An output is available for control experiments.

In addition, DrDAQ has two sockets for external sensors. When a sensor is plugged in, the software detects it and automatically scales readings. DrDAQ uses power from the PC, so neither batteries nor external power are required.

PicoScope (oscilloscope) and PicoLog (data logging) software are supplied. PicoScope enables rapidly changing signals. PicoLog allows DrDAQ to be used as an advanced data logger over long periods of time. Both software products are compatible with any PC running Windows 3.1x 95/98/2000 or NT.

DrDAQ comes with practical experiments for educational purposes. The built-in sensors enable experiments concerning light, sound (dB level and waveforms), and temperature to be done almost immediately. DrDAQ also has useful outputs for controlling experiments.

DrDAQ is supplied with cables, software, documentation, and example science experiments. It sells for **$99**.

**Saelig Company**
**(716) 425-3753**
**Fax: (716) 425-3835**
**www.saelig.com**

## ANALOG EVALUATION SYSTEM

The **MXDEV** Analog Evaluation System helps embedded designers evaluate and develop products with Microchip analog components. It consists of a driver board and an evaluation daughter board. The driver board performs data acquisition and connects to a PC for analysis and display. The daughter board plugs into the driver board and contains the device to be evaluated.

The driver board contains three PIC microcontroller sockets, an LCD display, LED display socket, SRAM for data storage, and RS-232 interface. Software, complete documentation, user's guides, and a technical library CD-ROM are also included.

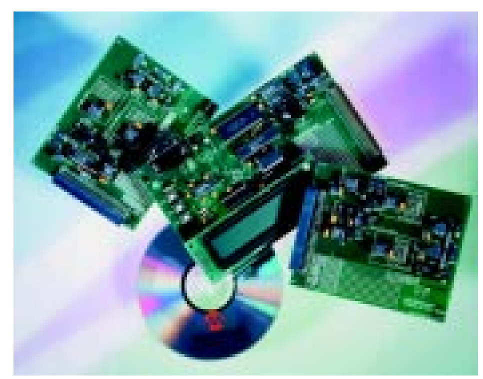
The MXDEV 1 system supports Microchip's 10- and 12-bit analog-to-digital converters with many development features. The tool allows single or continuous conversions for the analog-to-digital converter being evaluated.

Data can be acquired in real-time or acquisition mode. Data can be displayed in real-time numeric, stripchart, Fast Fourier Transform (FFT), Histogram, oscilloscope plot, and data list. The FFT display allows many window options, including Blackman, Blackman-Harris, Hamming, Hanning, and Rectangular.

Several jumper-selectable options mean flexibility. And, the FilterLab Active Filter Design Tool simplifies active filter design. Available for free at Microchip's web site, the tool automates the anti-aliasing filter design for an analog-to-digital, converter-based data acquisition system.

The MXDEV 1 Analog Evaluation System costs **$169** for the driver board and **$89** for the daughter board.

**Microchip Technology, Inc.**
**(480) 786-7668**
**Fax: (480) 899-9210**
**www.microchip.com**



# NEW PRODUCT NEWS

## GPS ANTENNA

The fixed-mount **GPS-FM antenna** enables the capture and deployment of synchronized time signals for timing applications in the cellular, paging, PCS operations, and other industries.

The antenna permits GPS signals to be used at fixed locations to establish precise time and space references. For example, local TV station affiliates requiring precision time signals to trigger switches to network operations would receive the signals from a satellite system using Hirschmann GPS antennae.

Also, the antenna can be used to locate signal sources. For example, if a caller dials 911, analysis and interpretation of the time measurement received at two cell sites may locate the caller.

The Hirschmann GPS-FM antenna withstands harsh conditions. Its UV inhibited radome is sealed to provide long service in extreme environments. The mounting bracket is allodized to prevent corrosion and pitting. All hardware is stainless steel.

The antenna has a 3.5-dB gain, 26-dB LNA gain, and operates at 1575 MHz ± 4.0. VSWR is <2.0:1 and impedance is 50 W.

Pricing for the GPS-FM antenna starts at **$275**.

**Hirschmann of America, Inc.**
**(973) 830-2000 • Fax: (973) 830-1470**
**www.hirschmann-usa.com**

# NEW PRODUCT NEWS

## RUGGED CAPACITIVE SENSORS

The **PT30** and **KT34 PVDF** capacitive sensors are barrel style sensors designed for harsh chemical environments. Capacitive sensors reliably detect all metallic and non-metallic materials, including water, metal, wood, glass, cardboard, plastic, concrete, glue, thin wire, silicon wafers, and others. Constructed of polyvinylidene fluoride (PVDF), the housing, cable, cord grip, and mounting brackets are immune to damage from contact with most industrial chemicals.

The devices feature a potentiometer for sensitivity adjustment. The 30-mm diameter, threaded, barrel PT30 model provides a 10-mm sensing range with the ability to repeat <2% of rated operating distance, even when embedded in steel. The 34-mm diameter, smooth barrel, non-embedded KT34 achieves a 20-mm sensing range, with the same repeat rating.

The sensors also feature a compensation electrode that minimizes false outputs due to contamination on the sensor face. The compensation circuitry ignores splashing water and dust buildup.

Both are four-wire DC, 10–65-VDC, capacitive devices that include LEDs that indicate "power on" and "output energized". Both feature outputs (usually one open and one closed) with either NPN (sinking) or PNP (sourcing) configurations. They are protected against short-circuit, overload, wire-break, reverse polarity, and transient voltages.

Available in flush and non-flush mounting styles, the sensors include a potted-in, 2-meter PVDF-jacketed, PVC insulated cable with an aluminum polyester shield and drain wire. The devices operate in temperatures from –25° to +70°C, and meet NEMA 1, 3, 4, 6, and 13 and IEC IP67 environmental standards.

**TURCK Inc.**
**(800) 544-7769**
**Fax: (612) 553-9575**
**www.turck.com**

# READER I/O

## THE OLD SCHOOL: CONTINUED

I have to agree with Clifford Stoll, as quoted by Ian Jefferson in June's Reader I/O section, but I'd extend that to the box of parts *not* containing a BIOS chip, but a simple machine code monitor only, with the only fancy code being something to service the keyboard and drive the basic SVGA monitor in a straight black and white mono mode. Having only the monitor, the next step after a powerup would be to start writing a BIOS for it.

Given that to do so would require a full, no warts covered, set of specs on the actual hardware that was used to build it.

Note that I'm not saying a bunch of school-kids are supposed to be able to to match what it took Award Software (now part of Phoenix Technologies) decades to do, but if they actually get the floppy drive working, I'd say the whole class gets an A.

That would accomplish two things. First, they would generally have a much better understanding of what it takes to actually program one of these things, bringing a lot of respect for the efforts of the folks who have made such things as the current windoze craze possible. The second benefit would be the serendipity effect of having exposed those children to the world of programming at the "get your hands dirty" level.

I could further expound, but I think this makes the point. Teaching a room full of kids how to run Windows 95 is a waste of both the teachers' time and the schools' resources. Schools should give children the basic tools to do the job, not assume those tools are nothing but a pull of the checkbook away.

**Gene Heskett**

---

*Editor's Note: In Brian Millier's article "Back to the BasicX: Part 1—NetMedia's Development System" (*Circuit Cellar*, 119) there was a mistake in the URL listed in the Software section. The correct URL for Brian's site is bmillier.chem.dal.ca.*

*Editor's Note: In Fred Eady's "Picking Some ExacTicks" article (*Circuit Cellar*, 119), it was not mentioned that ExacTicks is a commercial software library that is published by Ryle Design. For more information on ExacTicks, contact:*

*Ryle Design*
*(517) 773-0587*
*www.ryledesign.com*
*info@ryledesign.com*

FEATURE ARTICLE

Jonathan Valvano

# Simplify Your Software Testing

**The more complex your system is, the more complex your software will need to be. As Jonathan shows, using finite state machines can make the whole process of software design more efficient, more effective, and maybe even more fun.**

Successful engineers employ well-defined design processes when developing complex systems. When working within a structured framework, it's easier to prove the system works and then maintain it. As software systems become more complex, it's increasingly important to use well-defined software design processes.

In this article, I will present finite state machines implemented with linked data structures that you can use as a framework to solve embedded applications. Finite state machines provide an efficient, effective solution for embedded systems where the digital outputs depend on the current and previous digital inputs. To illustrate the design process, I'll present a stepper motor controller, traffic light controller, and a vending machine.

## FINITE STATE MACHINES

Software abstraction is the process of defining a complex problem in terms of a set of basic abstract principles. You have a better understanding of the problem and its solution if you can construct your software system using abstract building blocks.

Abstraction provides for a proof of correct function and simplifies both extensions and customization. The abstraction presented in this article is the finite state machine (FSM). The inputs, outputs, states, and state transitions are the abstract principles of FSM development. The FSM state graph defines the time-dependent relationship between its inputs and outputs. If you can map a complex problem into an FSM model, you can solve it with FSM software tools.

Other examples of software abstraction include Proportional/Integral/Derivative (PID) digital controllers, fuzzy logic digital controllers, neural networks, and linear systems of differential equations. In each case, the problem is mapped into a well-defined model with a set of abstract yet powerful rules. Developing the software solution is a matter of implementing the model's rules. After you prove your software correctly solves one FSM, you can make changes to this FSM with confidence that the solution correctly implements the new FSM.

The FSM controller uses a well-defined model or framework to solve problems. The state graph is specified using a linked data structure. There are three advantages of this abstraction. First, development is quick because many of the building blocks already exist. Second, it's easier to debug because it separates conceptual issues from implementation. Third, it's easier to change.

In this article, I'll demonstrate how to implement Moore FSMs, in which the output value depends only on the current state, and the inputs affect state transitions. The outputs of a Mealy FSM depend on the current state and inputs.

## LINKED DATA STRUCTURES

Linked lists are software data structures that consist of multiple identically-structured nodes. One or more of the entries in the node is a pointer, or link, to other nodes. In an

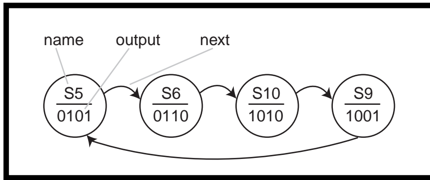



**Figure 1**—*This stepper motor FSM has four states. The 4-bit outputs are given in binary.*

embedded system, you usually use statically allocated, fixed-size linked lists. These lists are defined at compile time and exist throughout the software's life. The state graph is fixed in a simple embedded system, therefore you can store the linked data structure in nonvolatile memory. For complex systems in which the control functions change dynamically, you may implement dynamically-allocated linked lists. These linked lists are constructed at run time. And note that node numbers can grow and shrink in time.

I will use a linked structure to define the FSM. Note that there should be one node for each state.

## STEPPER MOTOR CONTROLLER

Figure 1 shows a circular linked graph containing the output commands to control a stepper motor. This simple FSM has no inputs, four output bits, and four states. There is one state for each output pattern in the usual stepper sequence—5, 6, 10, 9…. I use the circular FSM to spin the motor clockwise. Notice the one-to-one correspondence between the state graph in Figure 1 and the `fsm[4]` data structure in Listing 1.

I connected four high-current drivers to Motorola MC68HC11's Port B (see Figure 2). The low-current rating of the driver must be large enough to energize the stepper coils. The ULN2074 datasheet states that its maximum $I_{CE}$ is 1.25 A. But, because the ULN2074 is a high-current Darlington switch, its $I_{CE}$ will also be limited by the input base current, which comes from the 6811's $I_{OH}$. In this case, the $I_{OH}$ of the MC68HC11A8 is 0.8 mA, so this ULN2074 circuit can sink up to 500 mA.

The main program (see Listing 1) begins by initializing the Port B output and the state pointer. The 6811 Port B has no direction register. Every 5 ms, the program outputs a new stepper command. The `Wait()` function uses the built-in 6811 timer to generate an appropriate delay between outputs to the stepper.

To illustrate how easy it is to modify this implementation, let's consider the two modifications. To make it spin in the other direction, I change pointers to sequence in the other direction. To implement an eight-step sequence (e.g., 5, 4, 6, 2, 10, 8, 9, 1…), I add the four new states and link all eight states in the desired sequence.

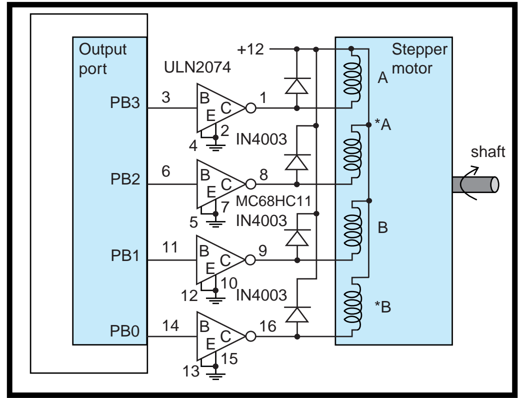



**Figure 2**—*When a unipolar stepper motor is interfaced to a Motorola MC68HC11, a high output value on Port B causes a low voltage on the ULN2074 output, causing current to flow through the coil. A low output value on Port B causes the ULN2074 output to float, and no current flows through the coil.*

**Listing 1**—*This 6811 software spins a stepper motor at 200 steps per second in the clockwise direction.*

```
/* Port B bits 3-0 are outputs to the stepper */
const struct State {
  unsigned char Out;          /* stepper command */
  const struct State *next;}; /* clockwise */
typedef const struct State StateType;
#define S5 &fsm[0]
#define S6 &fsm[1]
#define S10 &fsm[2]
#define S9 &fsm[3]
StateType fsm[4]={
  { 5, S6},                   /* Out=0101, Next=S6 */
  { 6,S10},                   /* Out=0110, Next=S10 */
  {10, S9},                   /* Out=1010, Next=S9 */
  { 9, S5}};                  /* Out=1001, Next=S5 */
/* delay time is given 500ns units */
void Wait(unsigned short delay){ short Endt;
  Endt=TCNT+delay;            /* TCNT at the end of delay */
  while((Endt-(short)TCNT)>0);}
void main(void){ StateType *Pt;
  PORTB=5;                    /* initial output */
  Pt=S5;                      /* initial state */
  while(1){                   /* embedded systems never quit */
    Wait(10000);              /* 5ms wait */
    Pt=Pt->next;              /* Clockwise step */
    PORTB=Pt->Out;}}          /* stepper drivers */
```

## TRAFFIC LIGHT CONTROLLER

Controlling traffic is another good example. I placed car sensors on the north and east roads, which I simulated with two switches connected to port C of my 6811 (see Figure 3 and Photo 1). If the digital input is a one (high), I assumed a car was on that road. To simulate the traffic light, I interfaced six color LEDs to Port B.

Figure 4 shows the simple Moore FSM that controls traffic at my intersection. The goal is to maximize traffic flow, minimize waiting time at a red light, and avoid accidents. For example, if I am in state `goN`, I set the port B output to $100001_2$ (green light on north and red light on east) and then wait 30 s.

Next, I read the sensor inputs. If the sensor value is $01_2$ (a car has entered the intersection on the east road, but no car exists on the north road), I go to state `waitN` (yellow light on north and red light on east). After showing the yellow light for 5 s on the north road, my controller switches to state `goE` (red light on north and green light on east), regardless of the input.

The main program (see Listing 2) begins by specifying the Port C bits 1 and 0 to be inputs. When working with CMOS microcomputers, define unused I/O pins either as outputs or specify them as inputs and tie the pin high or low in the hardware. In this example, I defined the unused pins

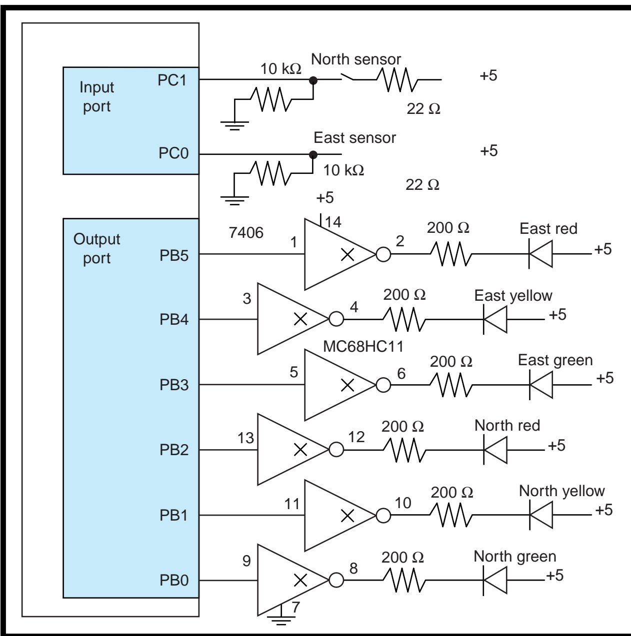



**Figure 3**—*This simulated traffic intersection is interfaced to a Motorola MC68HC11.*

PC7–PC2 as outputs. Because of the high impedance of CMOS inputs, an unconnected input pin may oscillate, dissipating power unnecessarily.

The initial state is defined as `goN`. My controller software first outputs the desired light pattern to the six LEDs, waits for the specified amount of time, reads the sensor inputs from Port C, then switches to the next state depending on the input data. The function `Wait1sec()` calls the `Wait()` function defined in Listing 1. To simplify, I made a one-to-one correspondence between the state graph in Figure 4 and the `fsm[4]` data structure in Listing 2.

Notice how this implementation separates the civil engineering policies (FSM specifications) from the computer engineering mechanisms (C program specifications). After I prove the C program is operational, I can modify it with confidence that the mechanisms will still work. When an accident occurs, I'll blame the civil engineer who designed the state graph.

Again, the FSM approach makes it easy to change. To change the wait time for a state, I simply change the value in the data structure. To add more complexity (e.g., put a red/red state after each yellow state, which will reduce accidents caused by bad drivers), I increase the size of the `fsm[]` structure and define the `Out`, `Time`, and `Next` pointers for these new states.

To add two more output signals (walk and left turn lights), I use all eight bits of the `Out` field. I increase the precision of the `Out` field to add more output bits. To add two more input lines (wait button, left turn car sensor), I increase the size of the pointer field to `Next[16]`.

Now, there are 16 possible combinations, because there are four input lines. Each input possibility requires a `Next` state pointer specifying where to go if this combination occurs. In this simple scheme, the size of the `Next[]` field will be two, raised to the power of the number of input signals.

## VENDING MACHINE

The vending machine example demonstrates additional flexibility that you can build into your implementations. Rather than simple digital outputs, I'll implement general functions for each state. I could solve this vending machine using the approach in the previous example, but I want to show you an alternative mechanism to use when the output operations become complex.

My simple vending machine has two coin sensors, one for dimes and one for nickels, which I will again simulate with two switches connected to 6811's Port C (see Figure 5). If the digital input is high, I consider there to be a coin in the slot. When a coin is inserted into the machine, the sensor goes high, then low. Unfortunately, when I activate or deactivate the switch, its contacts bounce, causing oscillations on the digital line. Even though a real coin sensor may not bounce, it's useful that the simulated machine sensors bounce, because now I can show you how to debounce a switch using FSM software.

**Listing 2**—*Here's the 6811 C software that controls traffic in this intersection.*

```
/* Port C bits 1,0 are sensor inputs,
   Port B bits 5-0 are LED outputs */
const struct State {
  unsigned char Out;                /* Output to Port B */
  unsigned short Time;              /* Time in sec to wait */
  const struct State *Next[4];};    /* Next if in-
                                       put=00,01,10,11*/
typedef const struct State StateType;
#define goN   &fsm[0]
#define waitN &fsm[1]
#define goE   &fsm[2]
#define waitE &fsm[3]
StateType fsm[4]={
  {0x21, 30,{goN,waitN,goN,waitN}},    /* goN   state */
  {0x22,  5,{goE,goE,goE,goE}},        /* waitN state */
  {0x0C, 30,{goE,goE,waitE,waitE}},    /* goE   state */
  {0x14,  5,{goN,goN,goN,goN}}};       /* waitE state */
void Wait1sec(unsigned short delay){ unsigned short i;
  for(i=0;i<delay;i++)
    Wait(2000);}                       /* 1 second wait */
void main(void){ StateType *Pt;        /* Current State */
  unsigned char Input;
  DDRC=0xFC;             /* Port C bits 1,0 are inputs from the
                            sensors */
  Pt=goN;                              /* Initial State */
  while(1){
    PORTB=Pt->Out;         /* Perform output for this state */
    Wait1sec(Pt->Time);   /* Time to wait in this state */
    Input=PORTC&0x03;     /* Input=00, 01, 10, or 11 */
    Pt=Pt->Next[Input];}}/* Move to next state */
```

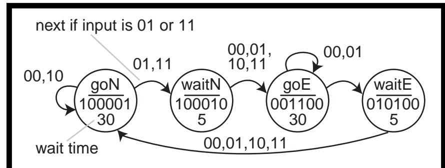



**Figure 4**—*This Moore FSM controls traffic in the intersection. The 2-bit inputs and the 6-bit outputs are given in binary. The wait times are given in seconds.*

I assume there can't be both a nickel and dime simultaneously. To simulate the soda and change dispensers, I interfaced two solenoids to Port B. The solenoids' coil current is less than 40 mA, so I can use the 7406 open collector driver. If you need more coil current (max $I_{OL}$ is 40 mA), you may upgrade to a ULN2074, similar to Figure 2.

Figure 6 shows the Moore FSM that implements my vending machine. A soda costs $0.15, and the machine accepts nickels and dimes. I have two input sensors that detect nickels (bit 0) and dimes (bit 1). The wait time in each state is 20 ms, which is greater than the switch bounce time but less than the time it takes the coin to pass by the sensor.

Waiting in each state will debounce the switch, preventing multiple counting of a single event. Notice that I wait in all states, because the switch bounces both on touch and release. Each state also has a function to execute. The function `soda` will trigger the Port B output so that a soda is dispensed. Similarly, the function `change` will trigger the Port B output so that a nickel is returned. The M states refer to the amount of collected money. When in a W state, I have collected that much money, but I'm still waiting for the last coin to pass the sensor.

For example, I start with no money in state M0. If I insert a dime, the input will reach $10_2$, and the state machine will jump to state W10. I'll stay in state W10 until the dime passes by the coin sensor. When the input reaches 00, I go to state M10. If I insert a second dime, the input will reach $10_2$, and the state machine will jump to state W20. Again, I stay in state W20 until this dime passes. When the input hits 00, I go to state M20. Then, I call the `change` function and jump to state M15. Lastly, I call the `soda` function and jump back to state M0.

The main program begins by specifying the Port C bits 1 and 0 to be inputs (Listing 3). The initial state is defined as M0. My controller software first calls the function for this state, waits for the specified amount of time, reads the sensor inputs from Port C,

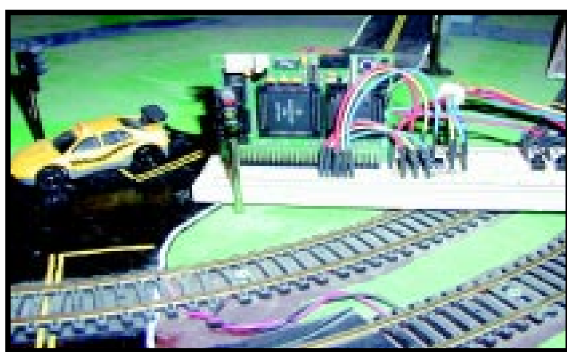


**Photo 1**—*A simulated traffic intersection is built with colored LEDs and is controlled by a Motorola MC68HC11.*

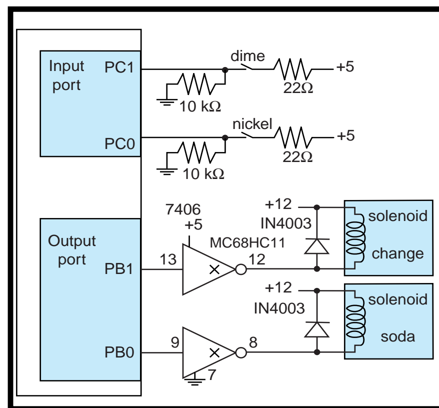



**Figure 5**—*A simulated vending machine interfaced to a Motorola MC68HC11.*

then switches to the next state depending on the input data. The `Wait()` function is defined in Listing 1. Again, note the one-to-one correspondence between the state graph in Figure 6 and the `fsm[9]` data structure in Listing 3.

## SIMPLIFY, SIMPLIFY

In this article, I presented a possible approach to solving embedded applications where digital outputs depend on the time history of its digital inputs. The advantage of FSM implementation is the separation of what the controller does (state graph) from how it is accomplished (C program).

You can improve these systems. If you move the state pointer (`Pt`) to a global variable, the controller software can be executed in a periodic interrupt handler. This way, you may run multiple machines on the same computer and still have the main program free to perform other tasks. As the number of input lines increase, substitute functions that test for certain conditions in place of the simple vector list used in these examples.

Early simulation will allow you to identify critical problems and rapidly evaluate alternative solutions. ◼

*Jonathan W. Valvano is a full professor of electrical and computer engineering at University of Texas at Austin, where he has taught and performed research since 1981 in the fields of medical instrumentation and embedded systems. He received his BS and MS degrees in Electrical Engineering and Computer Science at MIT in 1977. He did his PhD. research in biomedical instrumentation and received his doctorate in 1981 in Medical Engineering from the Harvard University/MIT Health Sciences and Technology Program. He has authored more than 60 journal articles, four book chapters, and one textbook. You may reach him at valvano@uts.cc.utexas.edu or visit*

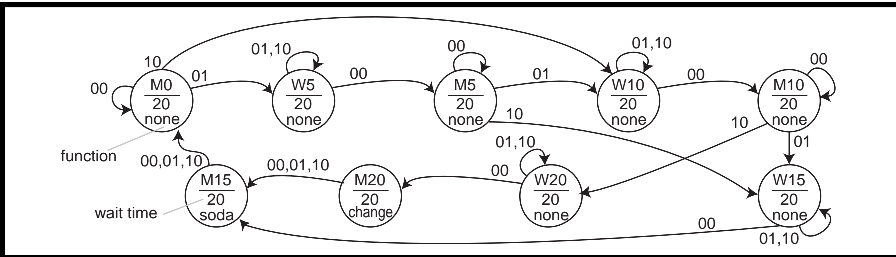



**Figure 6**—*This Moore FSM implements a vending machine.*

**Listing 3**—*Here's the 6811 C software that implements the vending machine application.*

```
/* Port C bits 1,0 are coin sensor inputs,
   Port B bits 1-0 are solenoid outputs */
void none(void){};
void soda(void){
  PORTB=1;                       /* activate soda solenoid */
  Wait(20000);                   /* 10 msec */
  PORTB=0;}                      /* deactivate solenoid */
void change(void){
  PORTB=2;                       /* activate change solenoid */
  Wait(20000);                   /* 10 msec */
  PORTB=0;}                      /* deactivate solenoid */
const struct State {
  void (*CmdPt)(void);           /* function to execute */
  unsigned short Time;           /* Time in msec to wait */
  const struct State *Next[3];}; /* Next if input=00,01,10*/
typedef const struct State StateType;
#define M0  &fsm[0]
#define W5  &fsm[1]
#define M5  &fsm[2]
#define W10 &fsm[3]
#define M10 &fsm[4]
#define W15 &fsm[5]
#define M15 &fsm[6]
#define W20 &fsm[7]
#define M20 &fsm[8]
StateType fsm[9]={
  {&none,  20,{M0,W5,W10}},       /* M0  state */
  {&none,  20,{M5,W5,W5}},        /* W5  state */
  {&none,  20,{M5,W10,W15}},      /* M5  state */
  {&none,  20,{M10,W10,W10}},     /* W10 state */
  {&none,  20,{M10,W15,W20}},     /* M10 state */
  {&none,  20,{M15,W15,W15}},     /* W15 state */
  {&soda,  20,{M0,M0,M0}},        /* M15 state */
  {&none,  20,{M20,W20,W20}},     /* W20 state */
  {&change,20,{M15,M15,M15}}};    /* M20 state */
void main(void){ StateType *Pt;   /* Current State */
  unsigned char Input;
  DDRC=0xFC;   /* Port C bits 1,0 are inputs from sensors */
  Pt=M0;                          /* Initial State */
  while(1){
    (*Pt->CmdPt)();               /* execute function */
    Wait(Pt->Time);               /* Time to wait in this state */
    Input=PORTC&0x03;             /* Input=00, 01, or 10 */
    Pt=Pt->Next[Input];}}         /* Move to next state */
```

*his web site at www.ece.utexas.edu/ ~valvano.*

## RESOURCE

For more information about FSMs, switch debouncing, interfacing, and software development, read *Embedded Microcomputer Systems: Real Time Interfacing*, Jonathan W. Valvano, Brooks-Cole Publishing, 2000. This book includes the TExaS microcomputer simulator.

## SOURCES

**Interface components**
BG Micro
(972) 271-9834
(800) 276-2206
Fax: (972) 205-9417
www.bgmicro.com

**LEDs and solenoids**
All Electronics Corp.
(888) 826-5432
Fax: (818) 826-5432
www.allelectronics.com

**6811 Adapt11 microcomputer board**
Technological Arts
(416) 963-8996
(416) 963-9179
www.interlog.com/~techart

**ICC11**
ImageCraft Creations Inc.
(650) 493-9326
Fax: (650) 493-9329
www.imagecraft.com

FEATURE ARTICLE

**Steve Freyder, David Helland, & Bruce Lightner**

# Look Ma, No PC!

## A $55 Webcam

If these guys caught your attention with their "$25 Web Server" article in *Circuit Cellar Online*, then you won't want to miss their latest project. Read the article, get the materials, build the project, smile for the picture. It's that easy.

If you think that it's impossible to build a full function web camera that includes the camera, web server, network interface, and software for under $55, then keep reading!

There has been a battle brewing at the low end of network interface products for embedded applications. It seems that everyone is interested in getting their equipment hooked up and online as a network appliance. Until recently this was an expensive proposition requiring PCs, network interface cards, HTTP server software, and TCP/IP protocol stacks.

If you have a simple application that would benefit from Internet accessibility, such as providing a temperature reading, buying a PC and the necessary network software for such an application is probably out of the question. However, cheaper alternatives are available.

### CHEAP WEB SERVERS

Early approaches to cutting the size and cost of embedded network controllers involved using single-board PCs (e.g., based on the 80188 or '386EX). These are still reasonable solutions if your task requires a fair amount of computational effort. However, the cost of these solutions is generally well over $150. There are now several special-purpose chips that supply the network interface protocols required to hook up your favorite micro to the Internet.

For example, Hewlett-Packard has the Bfoot-10501 chip. It has a serial port to attach to your external device, and a 10BaseT Ethernet interface to connect everything to the 'Net. HP's offering includes a web server, allowing a web browser to control and monitor your device. The HP device is relatively expensive ($240 in small quantities), but it's unlikely to be cost effective until your quantities are high.

Something like the EmWare system allows many small devices to be connected to a serial network for the purposes of web access. However EmWare's solution still requires a PC to provide a "gateway" to your local area network (LAN) and the Internet.

Another alternative is dial-up Internet access. ConnectOne, Scenix, and Epson have chips that can connect to the Internet via a modem (or terminal server). But, if you need a direct connection to your LAN, need high-speed access to your device, or can't afford modems (or terminal servers) at both ends of your connection, these solutions are probably unacceptable.

A more cost-effective way of providing a web-based network interface with a direct connection to your LAN is the PicoWeb server ($79). This is a complete solution that provides a TCP/IP stack, an HTTP web server, and a 10BaseT Ethernet connection for your device. The PicoWeb server can stand alone as a web server without the need to interface to another microprocessor, or in many cases, to even write software. Right out of the box you can load your HTML code and images, plug the device into the LAN, and use your web browser to display web pages from the device.

The PicoWeb project was started by a group of friends who wanted to settle an argument about whether or not a single chip microprocessor could really deliver web pages. The result was an article demonstrating how to build "A $25 Web Server" in *Circuit Cellar Online* 1 (July, 1999) using a $6 Atmel microcontroller and a $9 PC

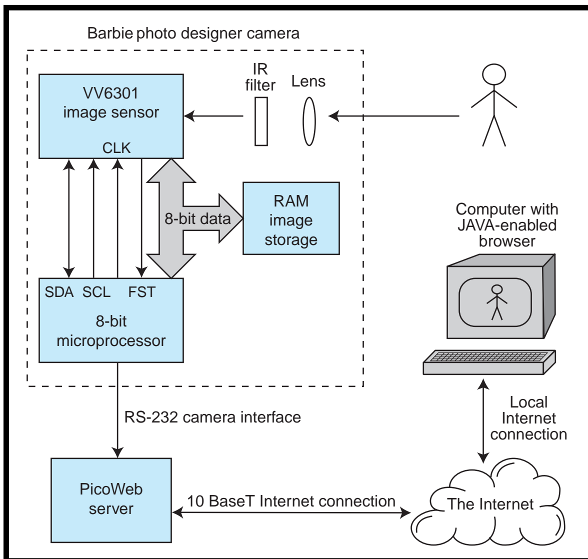



**Figure 1**—*A toy camera is combined with a tiny web server to give you an inexpensive webcam that can be accessed from anywhere on the Internet.*

Figure 1. No PC is needed to provide the web server functions and Internet interface. All functionality is provided by a tiny 8-bit Atmel microprocessor as part of the PicoWeb server.

ISA-bus network card, complete with all the necessary firmware and development system software.

Lightner Engineering's PicoWeb server is a commercial product spawned by that project. Even though the PicoWeb's microcontroller has only 8 KB of program memory and 512 bytes of RAM, it effectively delivers web pages and more. A 16-KB serial EEPROM chip adds storage for graphic images, HTML, and CGI programs. A built-in UART and about 16 unused general-purpose digital I/O lines provide the facilities to connect the PicoWeb server to a wide variety of user devices. Many project examples and the entire software development tool set can be found at the PicoWeb site (www.picoweb.net).

This article will show the power of the PicoWeb server by attaching it to a serial device (an inexpensive digital camera) and how to program the PicoWeb to acquire pictures from the camera so they can be transferred to a web browser for display. This project will work on both the $25 "homegrown" version of the PicoWeb server and the commercial version.

## HARDWARE DESIGN

The project demonstrates how to make an inexpensive, low power, low cost web camera with off-the-shelf parts as shown in

The camera used in this project is a toy camera sold by Mattel as the Barbie Photo Designer ($59) or the Nick Click ($29) digital camera. Which one you choose is up to you. If you like pink and have money to burn, go for the Barbie camera. If you are cheap (like us) and don't mind a purple camera, buy the Nick Click.

Both cameras are low-resolution (160 × 120) CMOS-based color digital cameras with RS-232 serial interfaces. Both come with noisy software that allows the pictures to be integrated with Barbie or Nickelodeon cartoon characters into a variety of output formats. (Turn off your PC speakers if you check out these at work!)

## PICKING A CAMERA

Choosing a digital camera was critical because of the limited code space in the PicoWeb's microcontroller. Remember, we are dealing with 8 KB of program memory in an Atmel AT90S8515 microprocessor and it is already providing support for ARP, BOOTP, PING, UDP, TCP/IP, and HTTP web server functions. (That's right, only 8 KB of flash memory and 512 bytes of RAM.) So, our requirements call for a simple camera interface compatible with the PicoWeb's available message formats.

The first cameras we examined were the many parallel port cameras that are widely available for adding video to your PC. At first glance, these looked perfect: small, cheap, simple interface, some with interface protocol information, including driver source code (typically Linux). However, a closer look at the available protocols revealed that they are not simple to handle in firmware. These devices look more like raw video cameras than true digital cameras (e.g., the Quickcam by US Robotics). The firmware must set all the camera chip registers and make real-time adjustments for light levels. Some cameras require you to detect start of frame and beginning of scan line in the raw video stream.

We found web sites that were useful in evaluating these cameras, including "QuickCam Third-Party Drivers" and the "CpiA Webcam driver for Linux" (see Sources).

Another approach would have been to utilize an NTSC video camera and then capture the video image with a video capture device such as Play Inc.'s Play Snappy Video Snapshot 4.0. This device has a parallel port interface but the source code for the interface drivers wasn't readily available. The cost of this route was going to be over $250 for the two devices, and multiple power supplies would be required.

Yet another alternative is one of the high-end digital cameras being sold as alternatives to film cameras. In fact, source code is available for controlling many high-end digital cameras, several of which deliver JPEG images via RS-232 serial ports.

Open-source "freeware" offered by Eugene Crosser (and Bruce Lightner) can be used to download JPEG images from the serial interfaces of many Agfa, Epson, Olympus, Sanyo, and Nikon camera models. (Full source code is available at www.average.org/digicam.) However, the cost of these cameras ($300 and up) and the complexity of their serial protocols eliminated them from the quick, simple, and inexpensive Webcam project we had in mind.

## CHEAP CMOS CAMERAS

Finally, there are several inexpensive digital cameras on

| Command | Char | Param | Description |
|---|---|---|---|
| Set image index | 'A' | index | Set current image index to one of 6 images |
| Take a picture | 'G' | delay | Take photo and store as current image |
| Upload a picture | 'U' | 0 | Send current image to RS-232 port |

**Table 1**—*These are the only commands we needed to turn our Mattel fun camera into a Webcam.*

the market now. These are low resolution color cameras (160 × 120 pixels) with serial port interfaces that are generally sold as fun cameras for children. The manufacturers include Oregon Scientific (DS3838), Polaroid (FUN 320), and Mattel (Barbie Photo Designer and Nick Click). The two cameras from Mattel appear to have identical electronics inside. These cameras all seem to be based on the VVL300 digital output sensor from STMicroelectronics (formerly VLSI Vision Ltd. of Scotland).

The camera chip used in the Mattel digital cameras seems to be the STMicroelectronics' VV6301, a highly integrated color camera sensor. A block diagram of this chip is shown in Figure 2. These chips use a CMOS imaging device rather than the typical charge-coupled device (CCD) sensor. The advantage of CMOS-based sensors is that a single silicon process can be used to manufacture the chip and all its ancillary logic. Therefore, most of the elements necessary to make a camera can be collocated on a single die, and manufactured inexpensively.

On the other hand, CCD-based cameras require multiple ICs and typically multiple voltages for the different IC technologies involved. The claim is that a single chip CMOS-based imager has lower noise as a result of internal parts that are in close proximity, plus onboard regulators that allow operation from a single 5-V supply. The VV6301 sensor also provides automatic black level calibration and includes a simple 2-wire I$^2$C interface for connection to a microprocessor.

All you need to make a complete camera is a lens, memory for image storage, and a microprocessor to provide the desired camera functionality. The Mattel camera uses an Intel MCS 51 family microprocessor and static RAM for image storage.

One difference between a fun camera and a high-end digital camera is that the former depends on a PC to convert the raw pixel data into something useful (e.g., a JPEG image), and the latter does this inside the camera. Typically, there is no image compression done in the fun cameras. You only get uncompressed, raw image sensor data out the serial port. As you will see, raw pixel data needs a bit of processing to yield an image that can be viewed on a web page.

Mattel's cameras send a total of 20 KB of raw pixel data per photo. When converted into a compressed JPEG image, this same photo is typically only one-tenth this size.

There was no question that we couldn't do any useful image processing in the PicoWeb's tiny microcontroller. However, we still had a trick up our sleeves!

## PICOWEB SERVER HARDWARE

The PicoWeb server uses the Atmel AT90S8515 microprocessor because the architecture is quite sophisticated for a processor of this size and cost. All of the registers are directly available (not mapped, as in the 8051) and the memory address space is linear (not segmented into pages, as in the PIC).

The AT90S8515 is a low-power RISC processor with 8 KB of flash program memory, 512 bytes of EEPROM, 512 bytes of RAM, 32 I/O lines, and a built-in UART. With an execution rate of one instruction per clock and a clock rate of 8 MHz, the AT90S8515 can drive the PicoWeb's 10BaseT Ethernet controller's I/O bus at 1 MBps. The PicoWeb server includes a 16-KB serial EEPROM chip to hold things like GIF and JPEG images as well as things like HTML, text files, and Java byte-codes. You can see a photo of the commercial version of the PicoWeb server in our "$25 Web Server" article. The schematic for this version of the PicoWeb server can be found in Figure 3.

The PicoWeb's Ethernet controller is a Realtek RTL8019AS, a single chip NE2000-compatible device with 16 KB of on-chip packet buffer RAM. This chip only needs a transformer, a single resistor and a few capacitors to implement a complete 10BaseT Ethernet network connection. The PicoWeb's DB-25 connector has up to 16 free general purpose digital I/O lines, an RS-232 serial port, and an in-circuit flash-memory programming port. An onboard voltage regulator accepts either AC or DC power in the range of 7 to 25 V . Typical current consumption is under 30 mA from the 5-V DC supply.

An NE2000 Ethernet chip is optimal because the Atmel processor memory is limited. The NE2000 controller has 16 KB of onboard SRAM that functions as a ring buffer to allow unattended reception of back-to-back Ethernet packets. (Because the commercial version of the PicoWeb server operates the Realtek chip in 8-bit mode, the available buffer RAM is reduced to 8 KB.) The same onboard Ethernet controller

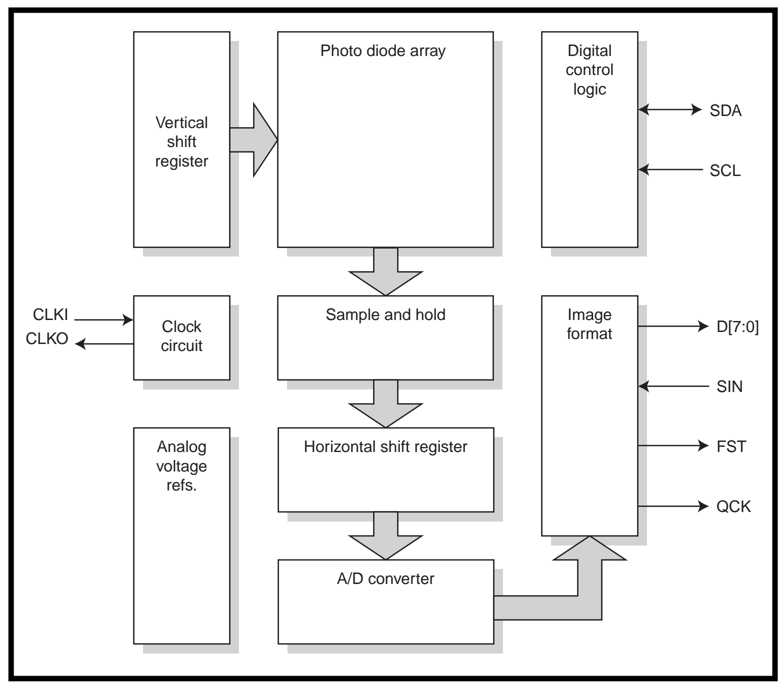



**Figure 2**—*The STM VV6301 gives you everything you need to make a digital camera on one chip, including a full-color CMOS sensor array.*

SRAM can be used to assemble transmitted Ethernet packets. The result is that the Atmel microcontroller's meager 512 bytes of on-chip SRAM is not needed to send or receive the maximum-sized 1500-byte Ethernet packets.

Connecting the camera to the PicoWeb server is simple, as Figure 4 illustrates. The data connection to the camera is a mini-stereo jack. The cable that comes with the camera (not used in our application) has this jack on one end with three wires (TX, RX, GND) that connect to a PC-compatible DE-9S serial connector on the other end.

A 9-V battery normally powers the camera, supplying a 5-VDC regulator chip inside the camera. We disassembled our camera and drilled a hole to add a power plug so we could power it off of the same unregulated 9-VDC supply as the PicoWeb. The PicoWeb's unregulated DC is available on a pin in its DB-25 connector. The camera only draws 70 mA from this connection. (We tried powering the camera's logic board directly from the PicoWeb's regulated 5-VDC power supply, but the camera kept warning us about its (missing) "low" 9-V battery!)

The images stored in the camera are located in its RAM, so removing the battery or disconnecting the DC cable from the PicoWeb will result in the loss of any stored images. To keep the camera from turning itself off to save its (now missing) 9-V battery, we programmed the PicoWeb to probe the camera over the serial port, about once per second. Modifying the standard PicoWeb clock frequency (from 8 MHz to 7.372 MHz) derived the 57.6 kbps rate needed by the camera.

## FIRMWARE FUNCTIONS

The basic function of the PicoWeb server is to allow embedded applications to display their data on the world wide web via its Ethernet connection. To accomplish this, the PicoWeb server's standard firmware supports a simple kernel, an optional tiny debug monitor, a "p-code" interpreter, a network adapter driver, a TCP/IP protocol stack, and an HTTP server (i.e., web server). The network protocols that the PicoWeb server's firmware supports include:

**Photo 1**—*The web page shows a photo that was captured by a Mattel Nick Click digital camera attached to a PicoWeb server's RS-232 serial port. Raw data from the CMOS-based 164 × 120 pixel sensor array are read into a Java applet (supplied by the PicoWeb server) and then converted into a viewable image. The raw pixels are in a 2 × 2 red-green-blue-green Bayer array. The Java applet uses nearest-neighbor bilinear interpolation to provide a full-color image from the raw pixel array. The resulting image is sharpened using a convolution function before display.*

- ARP—The PicoWeb server responds to network ARP requests to allow other computers to make an association between the PicoWeb server's assigned IP Address and its unique Ethernet address.
- BOOTP—The PicoWeb's IP address can be assigned statically, by storing the IP address in the microcontroller's flash EEPROM, or dynamically, by using the BOOTP protocol.
- Ping—The PicoWeb server also responds to ICMP Echo Requests ("ping") to allow users to quickly test network connectivity.
- UDP—The PicoWeb server can send and receive UDP packets.
- TCP/IP—At the TCP/IP level, the PicoWeb server responds to HTTP GET requests that are addressed to TCP port 80. The web server responds to these requests by sending

back HTML documents, text, and images. In addition, user-supplied firmware can make use of the PicoWeb's TCP/IP stack for other purposes.

The firmware kernel in the PicoWeb server provides all the code necessary to implement the needed parts of the Internet protocols listed above. In addition, the supplied software and firmware include tools to assist you in developing new web pages and adding program code to communicate via the external I/O devices.

The PicoWeb server allows both JavaScript (either embedded in HTML code or as separate files) and Java applets (i.e., Java byte-codes) to be stored in its serial EEPROM, along with HTML code and images. Java and JavaScript are potent tools that allow software routines that would otherwise be too large and or too complex to be run by the Atmel microcontroller to be executed by the user's web browser in a transparent way.

The PicoWeb's firmware suite contains an optional simple, extensible debugger that provides for things such as memory dumps, SRAM, and EEPROM memory alteration, "p-code" and network tracing control, and so on. Debugger commands can be entered via the serial port, or via the network using a web browser by accessing a special TCP port.

The format of a debugger command URL is http://IPaddress:911/command[[+parameter1]+parameter2].

Any results from executing a debug command will be returned as a web page. The supplied debugger commands are described in detail in documents located on the PicoWeb web site. New debugger commands can easily be added by the user (or deleted to save program code space).

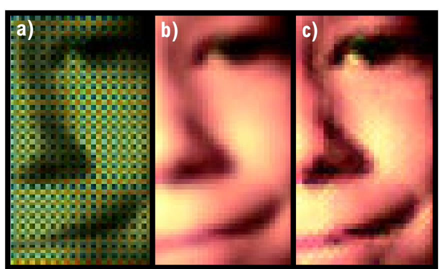



**Photo 3—***The first image* ***(a)*** *shows the raw pixel data from the camera (Bayer color pattern). This image is processed by the PicoWebCam-supplied Java applet to supply the "missing" pixels* ***(b)****, then it is sharpened* ***(c)****. Believe it or not, when not enlarged, the sharpened image looks better to most people.*

**Photo 2—***The Nick Click camera is powered by the same 9-V supply as the PicoWeb. Plug this into your 10BaseT LAN and you can snap and display the photos with your favorite web browser.*

The Atmel AVR AT90S8515 chip includes hardware to allow the flash memory and EEPROM to be programmed via a 3-wire SPI interface. This capability is used by the PicoWeb server to download the code into the flash memory and modify the on-chip EEPROM (e.g., for parameter storage). Downloading is accomplished by using a simple cable attached to the parallel port of a PC.

The PicoWeb server has firmware routines that allow the 16-KB serial EEPROM to be programmed remotely via the Ethernet interface. A utility program is provided that allows data, graphic images, HTML, Java applets, and p-code routines to be loaded into the serial EEPROM memory. The serial EEPROM segments are transferred over the network using a TCP-based loader program. This feature also allows the images and p-code routines to be updated while the PicoWeb server is active.

## SOFTWARE DESIGN

The first step in the software design was a little reverse-engineering. First, we analyzed the RS-232 traffic between our camera and the PC. The transfer rate was 57.6 kbps at 8 bits with no parity. A program was written for a PC in Borland C to further clarify the camera's communication protocol.

The command format for controlling the camera turned out to be simple. The commands are single upper case characters followed by an optional parameter. The command and parameter characters are preceded by an STX (0x02) and followed by an ETX (0x03). The camera responds to each command sequence sent to it with either an ACK (0x06) or a NAK (0x15). If a response to a command is warranted, the response data from the camera is preceded by an STX (0x02) and terminated by an ETX (0x03).

The camera typically responds by echoing the four-character command sequence, after changing the command character to lower case and replacing the parameter with a status/error code. Table 1 shows the camera commands utilized in this project.

Because we are interested in only the first picture, we must send a command to set the image index to zero before taking a picture and also before uploading a picture. The command sequence to take a picture and upload it to the serial port is shown in Table 2.

Note that the returned image data is sent in a continuous stream without any flow control. It takes about 4 s to transfer the full 20,680 bytes of image data. This means that the PicoWeb must receive, buffer, and transmit data to the open TCP/IP socket without dropping any of the incoming 57.6 kbps characters.

A prime design goal of the PicoWebCam was to make it work like other web cameras. That meant that accessing a web site (in this case the home page of the PicoWeb server) would simply cause a photo from the camera to be displayed. If the PicoWeb server is accessible from the Internet, then photos from the camera can be viewed from anywhere in the world. Nothing more should be needed to make this work than a Barbie camera, a PicoWeb server, and our simple cable. No gateway or helper PC should be required to produce images. Sounds like a problem for a $6 microcontroller with 512 bytes of RAM! But, with a little Java applet programming, we can cleverly push all of the "hard stuff" onto the web browser's host computer.

The first step in making all this happen is storing an HTML page in the PicoWeb server's serial EEPROM

memory. This HTML code is returned when the PicoWeb's home page URL is referenced (see Photo 1). This web page references a Java applet stored in the PicoWeb, which will give us a graphics window in the web page in which we later display the photos from the camera.

The web browser then asks the PicoWeb to deliver the referenced Java applet (stored as Java byte-codes). The applet is sent back to the browser which starts its Java interpreter and begins executing the Java program. The Java interpreter executing in the browser then displays a graphics window controlled by the Java applet.

The Java applet then makes a TCP/IP connection back to the PicoWeb to retrieve special HTML pages from the PicoWeb that contain embedded CGI p-code routine references. Retrieving these pages causes the associated p-code routines to be executed in the PicoWeb server. Initially, the PicoWeb server will be commanded to retrieve the latest photo from the camera. Note that this is not a Java security violation because a Java applet is allowed to make network connections back to the host computer that delivered the applet.

In response to the request from the Java applet, the PicoWeb server tells the camera to stream the latest photo over its serial port (i.e., upload command). The server sends this raw pixel data back to the Java applet over the open TCP/IP connection as a web page. This takes about 4 s and is paced by the speed of the 57.6 kbps serial connection with the camera. One byte of raw pixel data is sent for each pixel in the 164 × 120 sensor array.

The Java applet running in the web browser's computer receives the raw pixel data from the TCP/IP port and then processes the raw data to turn it into a displayable image. Next, the Java applet displays the received image.

The Java applet then begins "watching the mouse buttons." Using the mouse, users can call for a new photo to be taken by the camera and displayed, or they can call for the current image to be resized or alter the image processing options.

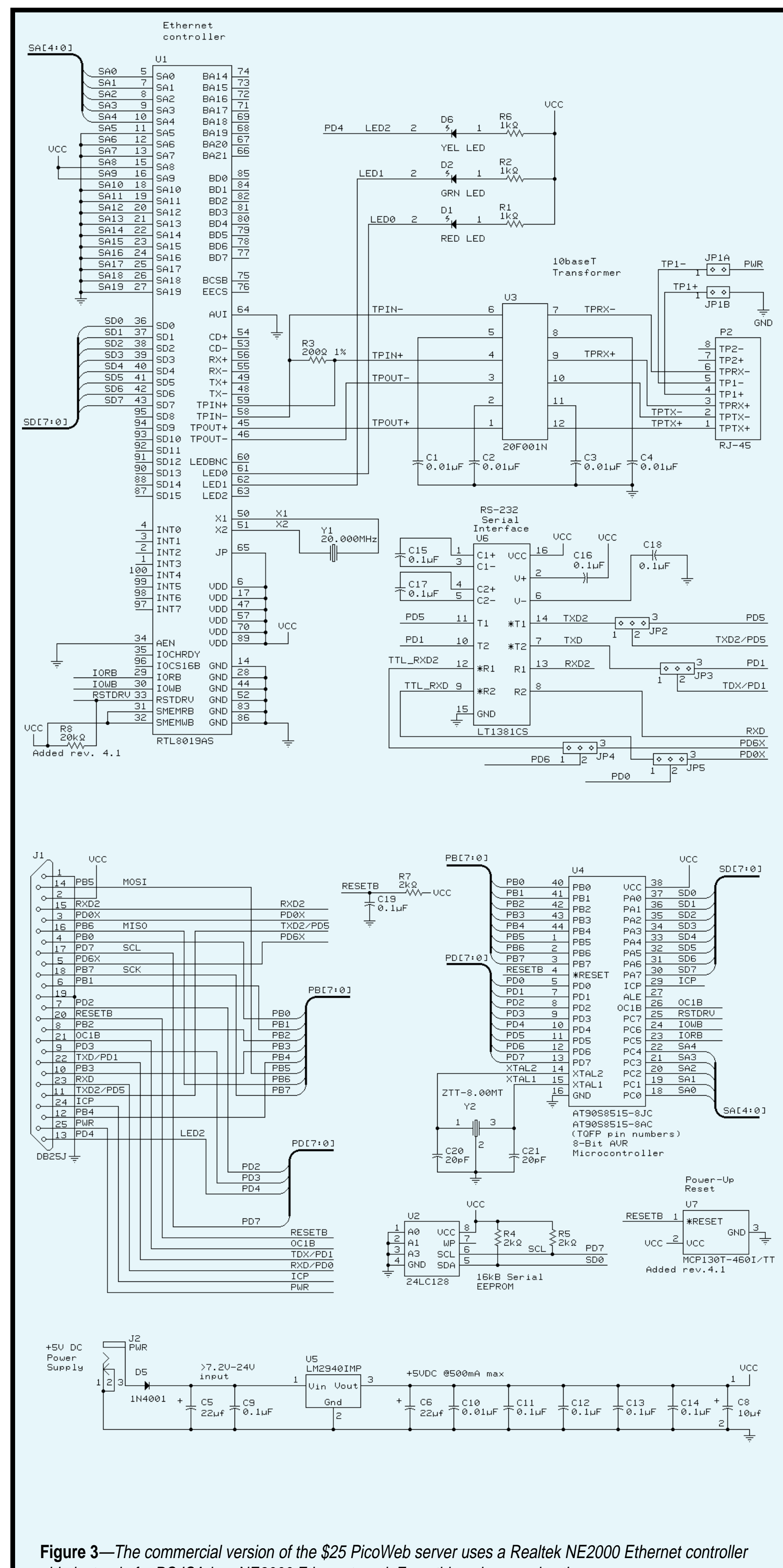



**Figure 3**—*The commercial version of the $25 PicoWeb server uses a Realtek NE2000 Ethernet controller chip instead of a PC ISA-bus NE2000 Ethernet card. Everything else remains the same.*

1. Set the image index to zero:
STX + 'A' + 0x00 + ETX then wait for ACK followed by STX + 'a' + 0x00 + ETX

2. Take a picture with no timer delay:
STX + 'G' + 0x00 + ETX then wait for ACK followed by STX + 'g' + 0x00 + ETX

3. Set the image index to zero again:
STX + 'A' + 0x00 + ETX then wait for ACK followed by STX + 'a' + 0x00 + ETX

4. Upload the picture:
STX + 'U' + 0x00 + ETX then wait for ACK followed by the image datastream: STX + 'u' + $N_1$ + $N_2$ + $N_3$ + $N_4$ + $D_1$ + $D_2$ + ··· $D_N$ + ETX

where $N_1$ = 164 (number of columns)
$N_2$ = 2 (number of black lines)
$N_3$ = 124 (number of visible lines)
$N_4$ = 16 (number of status bytes)
$D_1$ + $D_2$ + ··· $D_N$ are the data bytes of the image (20,680 bytes)

**Table 2**—*This is the sequence of camera commands and responses needed to take a photo and then upload the photo's image data to the PicoWeb server.*

As you see, we have the Java applet running on the user's web browser doing all of the things that are difficult or impossible for the PicoWeb server to do. As long as your web browser has Java enabled, you will be blissfully unaware of what is really going on behind the scenes.

## PICOWEB FIRMWARE

Very little new PicoWeb server code was required to implement this project. Existing example projects for controlling serial port devices were used as a starting point for this project. These are available at the PicoWeb web site (www.picoweb.net/downloads.html). All the code for the PicoWebCam project (p-code, HTML, Java) is also available and can be downloaded from the PicoWeb site.

The only new routines required for this project were those needed to reset the camera image counter, take a new picture, and upload the raw pixel data. These routines each consist of a few lines of PicoWeb p-code language, a kind of interpreted assembly language for a 16-bit virtual machine. The p-code interpreter was developed for the PicoWeb server to provide program code simplification and maximization of code re-use, allow the option to execute program code out of serial EEPROM, and reduce program code size as compared to native code.

More information about the PicoWeb p-code interpreter and how to write p-code for the PicoWeb server can be found at the PicoWeb web site in an article titled "PicoWeb P-code Description." The information necessary to allow developers to design their own PicoWeb projects also can be found at the same web site in an article titled "How to Build a PicoWeb Project."

## JAVA APPLET

A Java applet was necessary for this project because the image data returned from the Barbie Photo Designer camera needs to be processed before it can be displayed. The camera does not store images in a format that can be directly displayed by a web browser (e.g., JPEG or GIF images). Instead, the camera sends raw image data to the computer in the form of a Bayer color pattern. The Java code causes a TCP/IP socket to be opened by the web browser's computer to transfer the raw picture data from the PicoWeb server, and then to process the data as necessary into a viewable image.

The raw pixels from the camera come from a 2 × 2 red-green-blue-green Bayer array as shown in Figure 5. Each pixel in the image sensor chip is covered by a colored filter according to the Bayer pattern shown. There are two green pixels for every red and every blue pixel. We need to supply a red, green, and blue pixel for every possible pixel location in order to derive a real image. If we don't do this, we get a low-resolution greenish image, as shown in Photo 3a. We do this by looking at like-colored pixels in the neighborhood and making an intelligent guess about the probable color and intensity of the light that struck each pixel when the photo was snapped. (Next time you read about the latest full-color digital camera with 2.1 million pixels, remember that in some sense, two-thirds of the pixel data is made up!)

**Figure 4**—*A simple 5-wire cable connects the PicoWeb server to the Mattel digital camera. Adding a power connector to the camera's body means the camera can be powered from the same 9-VDC supply as the PicoWeb server.*

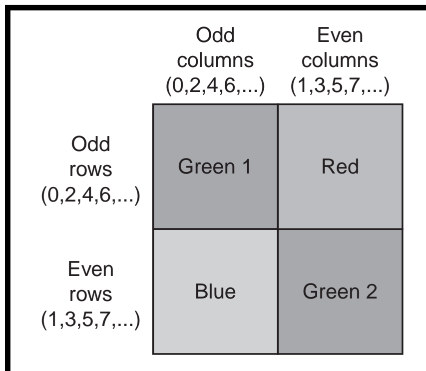



**Figure 5**—*This is the pattern of colored filters that covers the image sensor chip's 160 × 120 array of light sensors (pixels). The "missing" red, green, and blue pixel values must be interpolated from neighboring pixels of like color.*

We have lots of pixel interpolation algorithms to choose from, of varying complexity, and with a wide range of computational requirements. Our Java applet executes a simple, fast nearest-neighbor bilinear interpolation algorithm to quickly provide a full-color image from the raw pixel array. The resulting image is then sharpened using a convolution function before display. This is something that Mattel's PC software does in order to make their camera's otherwise tiny fuzzy photos look better. Photo 3b and 3c show an enlargement of a sample camera image after processing by the PicoWebCam's Java applet.

An excellent discussion of Bayer color pattern processing algorithms is titled "A Study of Spatial Color Interpolation Algorithms for Single-Detector Digital Cameras", by Ting Chen. [1] The basic algorithm for the image sharpening was inspired by an article titled "Image Processing with Java 2D", by Bill Day and Jonathan Knudsen. [2] The Java compiler used for this project was provided by Sun Microsystems. A complete, free Java program development kit (JAVATM 2 SDK, Standard Edition Version 1.3) is available for downloading from Sun.

## SMILE

We have established that our PicoWebCam can be constructed for as little as $55 by first building our $25 web server and then connecting it to a Nick Click digital camera. Not surprisingly, we think that for a few dollars more, the commercial version of the PicoWeb server is a better way to go. In either case, you get a complete, inexpensive, standalone web server with attached web camera, all in a tiny package. And the best part is no PC is required!

Clearly, the resolution of the toy cameras we used in the project is less than optimal for many applications. However, there are cost-sensitive commercial applications that could benefit from this project (e.g., a keypad entry system that records photos of all entry attempts).

The fact that the camera takes more than 4 s to send its image data out its serial port means that the frame rate of our PicoWebCam is horrible. However, it doesn't take a propeller head to note that the serial port bottleneck can be removed from the picture (no pun intended). In fact, just like Mattel, you too can buy CMOS imaging chips from STMicroelectronics (STM), and for a whole lot less than $29 each. All of STM's imaging chips that we looked at have a high-speed parallel interface, and evaluation boards sporting even higher resolution imaging chips are available from STM.

How about a PicoWebCam that delivers photos at the speed of the Ethernet! It's all possible, and now you've got all the information you need to roll your own. ■

*Steve Freyder telecommutes from his home in La Jolla, CA for Transcore, working on automated toll collection systems. He lost his "real" office many years ago by never visiting it. Steve has been programming since he first discovered computers in high school in 1970. You can reach him at steve@freyder.net.*

*David Helland works for Science Applications International Corporation (SAIC) in San Diego, CA, most recently working on portable electronics for military training range systems. Dave has been building hardware and software systems for several decades. In his spare time Dave restores vintage fiberglass dune buggies. You can reach him at dhelland@worldnet.att.net.*

*Bruce Lightner also works from home for Lightner Engineering in La Jolla, CA. He too discovered computers several decades ago and has been building hardware and software for them ever since. In his spare time he likes to abuse Dave's dune buggies. You can reach him at lightner@lightner.net.*

## SOURCES

**PicoWeb server**
Lightner Engineering
www.picoweb.net

**Digital camera software**
QuickCam third-party drivers
www.crynwr.com/qcpc

**CpiA webcam driver for Linux**
http://webcam.sourceforge.net

**PhotoPC digital camera software**
(open-source freeware)
www.average.org/digicam

**VVL300 digital output sensor**
STMicroelectronics
www.vvl.co.uk/products/image_sensors

**Java compiler**
Java 2 SDK, Standard Edition Version 1
Sun Microsystems
http://java.sun.com/products/jdk/1.2/

## RESOURCE

S. Freyder, D. Helland, and B. Lightner, "A $25 Web Server", *Circuit Cellar Online*, July 1999, www.chipcenter.com/circuitcellar/july99/c79bl1.htm.

## REFERENCES

[1] T. Chen, "A Study of Spatial Color Interpolation Algorithms for Single-Detector Digital Cameras", www-ise.stanford.edu/~tingchen/main.htm.

[2] B. Day and J. Knudsen, "Image processing with Java 2D", *JavaWorld*, www.javaworld .com/javaworld/jw-09-1998/jw-09-media.html, September 1998.

**You can link to the PicoWeb software and parts list under the "Sources and PDF" section of this article in August's issue of *Circuit Cellar Online*.**

FEATURE ARTICLE

Sandeep Dutta

# Anatomy of a Compiler

## A Retargetable ANSI-C Compiler

Wouldn't it be great if there was an affordable C compiler that was specifically designed to the needs of 8-bit micros? Not to worry, Sandeep explains the inner workings of just such a compiler, which happens to be quite affordable—it's free!

Small device C compiler (SDCC) is an open-source optimizing C compiler developed primarily for 8-bit MCUs. Although there are several free C compilers available to address the general-purpose processors (GCC, LCC), there are no free C compilers available except SDCC that address specific needs of 8-bit MCUs. In this article, I'll explore SDCC and some of the special considerations when designing a compiler for 8-bit MCUs.

I'll discuss the Intel 8051 because it's a widely used MCU. The concepts have been implemented in SDCC and the source code is available under GPL in the hope that other people will find it useful and contribute (either by providing feedback or making enhancements). Several commercial compilers implement the concepts demonstrated here.

### ADDRESS SPACES

Unlike their 32-bit brethren, most of the 8-bits use Harvard architecture, which means the code and data reside in different address spaces and are usually accessed using different instructions. For example, the 8051 family of controllers has three address spaces (four if you consider the upper 128 bytes of internal RAM as a different address space). C language allows for storage classes, however they are restricted to const, volatile, static, auto, and register. Although these are adequate for Von Neuman architecture, they're not sufficient for architectures with many address spaces.

The problem is more complex when you consider pointers. Where does the pointer reside? Which address space is it pointing to? Also consider library routines, which take pointers as parameters. Do you need to write library routines for all the combinations of address spaces?

SDCC handles this problem by adding keywords for new storage classes. Using the 8051 as an example, SDCC has storage class specifiers for each MCU address space. Listing 1 shows examples of declaring variables in different address spaces.

Frequently, a variable must be allocated at a specific/absolute address (i.e., a memory-mapped I/O device). Again, standard C doesn't provide for this, you would have to provide a special assembler routine (or inline assembly code). SDCC, however, provides a special keyword "at" to specify an absolute address for a variable. The memory-mapped I/O device can then be accessed in an expression using standard C syntax.

### POINTERS

The same concept of storage class extension can be used to solve the pointer problem. Listing 2 shows different ways to specify pointers. This leaves the problem of library routines, not knowing which storage class the pointer points to at compile time. The SDCC solution is generic 3-byte pointers; the third (highest order) byte contains information about the pointed at object's storage class.

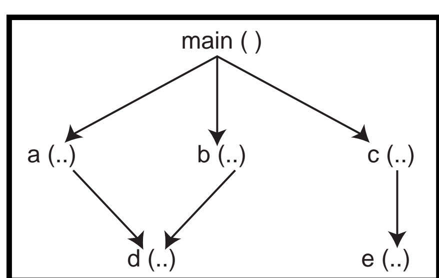



**Figure 1**—*The parameter locals of functions a, b, and c can be overlaid.*

At runtime, the compiler calls a routine that determines the storage class from the third byte and uses the appropriate instruction to fetch or store data. This technique increases code and data size, but is a compromise to allow coding general-purpose library routines, like `strcmp`.

## STACK

The most difficult obstacle in programming small devices (such as the 8051) in high-level language like C is the limited stack space available for local variables and parameter passing. Using registers for parameter passing lessens the problem, but you still must allocate local variables. SDCC solves this problem by treating parameters and local variables as static (at the expense of re-entrancy). It goes a step further by overlaying parameters and local variables of leaf functions (i.e., functions that call no other functions) to the same memory region.

What if you need the re-entrancy? You could either compile the entire source file with the stack-auto compiler option (all functions in the source file will be treated as re-entrant), or you can choose only specific functions to be reentrant by using the reentrant keyword in the function declaration. SDCC allocates parameters and local variables of a reentrant function on the stack.

The current version of the compiler only overlays local variables and parameters of a leaf function, but this isn't enough in some cases. Development is underway to do function call tree analysis, which would allow the compiler to overlay parameters and local variables of functions that don't belong to the same call sub-tree.

Consider the call tree illustrated in Figure 1. In this case, local variables and parameters (auto variables) of functions *d()* and *e()* can be overlaid (and are by the compiler). In addition, auto variables of functions *a()*, *b()*, and *c()* can be overlaid with each other, because they don't call each other and are not present in the call trees of any of their children. This kind of overlaying needs to be done by the linker because the compiler has only a partial view of the call tree.

**Listing 1**—*Here's the pointer declarations with extended storage classes.*

```
/* the following array will be declared in program memory
    MOVC instruction will be used to access this array */
 code  short array_in_code[3] = {0x01,0x02,0x03};

/* The integer will be allocated in internal ram space
MOV instruction will be used access this data item */
data unsigned char in_internal_ram ;

/* this will allocated in the external RAM and MOVX will
    be used to access this data item */
xdata char array_in_external_ram[9];

/* This variable will be allocated at address 0x8000 of
the external RAM */
xdata at 0x8000 ADC_PORTA;

/* pointer in data space points to object in xdata */
xdata char * p;

/* pointer in xdata space points to object in code space */
code char * xdata p;

/* pointer in code space points to object in data space */
data char * code p;
```

**Listing 2**—*This sample code illustrates iCode generation and optimizations.*

```
xdata int * p;
int gint;
                     /* This function does nothing useful. It is
                       used for the purpose of explaining iCode */
short function (data int *x)
{
short i=10;          /* dead initialization eliminated */
short sum=10;        /* dead initialization eliminated */
short mul;
int j ;
while (*x) *x++ = *p++;
sum =0 ;
mul =0;
                     /* compiler detects i,j to be induction vari-
                       ables */
for (i = 0, j = 10 ; i < 10 ; i++, j—) {
    sum += i;
    mul += i * 3;    /* this multiplication remains */
    gint += j * 3;   /* this multiplication changed to addition */
}
return sum+mul;
}
```

## INTERNAL DETAILS

The current version of SDCC can generate code for Intel 8051 and Z80 MCUs. It's easy to retarget for other 8-bit MCUs. Let's take a look at some of the internals of the compiler.

Parsing involves reading the input source file and creating an Annotated Syntax Tree (AST). This phase also involves propagating types (annotating each node of the parse tree with type information) and semantic analysis. There are some MCU-specific parsing rules. For example, the extended storage classes are MCU specific: while there may be an xdata storage class for 8051, there's no such storage class for the Z80 or Atmel AVR. SDCC allows MCU-specific storage class extensions to be treated as a storage class specifier when parsing 8051 C code, but to be treated as a C identifier when parsing Z80 or Atmel AVR C code.

In the intermediate code generation phase, the AST is broken into three operand forms (iCode). These forms are represented as doubly linked lists. iCode

is the term given to the intermediate form generated by the compiler. Listing 3 shows examples of iCode generated for simple C source functions.

The bulk of target-independent optimizations is performed during optimization. Optimizations include constant propagation, common subexpression elimination, loop-invariant code movement, strength reduction of loop induction variables, and dead-code elimination.

During the intermediate code generation phase, the compiler assumes the target machine has an infinite number of registers and generates many temporary variables. The live range computation determines the lifetime of each of these compiler-generated temporaries. iCode example sections in Listing 3 show the live range annotations for each operand. Note that each iCode is assigned a number in the order of its execution, which compute the live ranges. The `from` is the iCode number that first defines the operand and `to` signifies the iCode that uses this operand last.

**Listing 3**—*This is the iCode generated for the sample code in Listing 2.*

```
Sample.c (5:1:0:0)   _entry($9) :
Sample.c(5:2:1:0)   proc _function [lr0:0]{function short}
Sample.c(11:3:2:0)     iTemp0 [lr3:5]{_near * int}[r2] = recv
Sample.c(11:4:53:0) preHeaderLbl0($11) :
Sample.c(11:5:55:0)    iTemp6 [lr5:16]{_near * int}[r0] := iTemp0
   [lr3:5]{_near * int}[r2]
Sample.c(11:6:5:1)  _whilecontinue_0($1) :
Sample.c(11:7:7:1)     iTemp4 [lr7:8]{int}[r2 r3] = @[iTemp6
   [lr5:16]{_near * int}[r0]]
Sample.c(11:8:8:1)     if iTemp4 [lr7:8]{int}[r2 r3] == 0 goto
   _whilebreak_0($3)
Sample.c(11:9:14:1)    iTemp7 [lr9:13]{_far * int}[DPTR] := _p
   [lr0:0]{_far * int}
Sample.c(11:10:15:1)   _p [lr0:0]{_far * int} = _p [lr0:0]{_far *
   int} + 0x2 {short}
Sample.c(11:13:18:1)   iTemp10 [lr13:14]{int}[r2 r3] = @[iTemp7
   [lr9:13]{_far * int}[DPTR]]
Sample.c(11:14:19:1)   *(iTemp6 [lr5:16]{_near * int}[r0]) :=
   iTemp10 [lr13:14]{int}[r2 r3]
Sample.c(11:15:12:1)   iTemp6 [lr5:16]{_near * int}[r0] = iTemp6
   [lr5:16]{_near * int}[r0] +
                                                         0x2 {short}
Sample.c(11:16:20:1)   goto _whilecontinue_0($1)
Sample.c(11:17:21:0)_whilebreak_0($3) :
Sample.c(12:18:22:0)   iTemp2 [lr18:40]{short}[r2] := 0x0 {short}
Sample.c(13:19:23:0)   iTemp11 [lr19:40]{short}[r3] := 0x0 {short}
Sample.c(15:20:54:0)preHeaderLbl1($13) :
Sample.c(15:21:56:0)   iTemp21 [lr21:38]{short}[r4] := 0x0 {short}
Sample.c(15:22:57:0)   iTemp23 [lr22:38]{int}[r5 r6] := 0xa {int}
Sample.c(15:23:58:0)   iTemp17 [lr23:38]{int}[r7 r0] := 0x1e {int}
Sample.c(15:24:26:1)_forcond_0($4) :
```

*(continued)*

The register allocation determines the type and number of registers needed by each operand. In most MCUs, only a few registers can be used for indirect addressing. The compiler tries to allocate the appropriate register to pointer variables.

Listing 3 shows the operands annotated with the registers assigned to them. The compiler tries to keep operands in registers. The compiler uses several schemes to achieve this. When the compiler runs out of registers, it checks if there are any live operands that are not used or defined in the current basic block being processed. If found, it will push that operand and use the registers in this block. Then, the operand will be popped at the end of the basic block.

There are other MCU-specific considerations in this phase. Some MCUs have an accumulator so short-lived operands may be assigned to the accumulator instead of a general-purpose register.

A complete table that defines the iCode operations supported by the compiler is available on the *Circuit*

**Listing 3**—*continued*

```
Sample.c(15:25:27:1)   iTemp13 [lr25:26]{char}[CC] = iTemp21
   [lr21:38]{short}[r4] < 0xa {short}
Sample.c(15:26:28:1)   if iTemp13 [lr25:26]{char}[CC] == 0 goto
   _forbreak_0($7)
Sample.c(16:27:31:1)   iTemp2 [lr18:40]{short}[r2] = iTemp2
   [lr18:40]{short}[r2] +
                                                    ITemp21
   [lr21:38]{short}[r4]
Sample.c(17:29:33:1)   iTemp15 [lr29:30]{short}[r1] = iTemp21
   [lr21:38]{short}[r4] * 0x3 {short}
Sample.c(17:30:34:1)   iTemp11 [lr19:40]{short}[r3] = iTemp11
   [lr19:40]{short}[r3] +
                                                     iTemp15
   [lr29:30]{short}[r1]
Sample.c(18:32:36:1:1) iTemp17 [lr23:38]{int}[r7 r0]= iTemp17
   [lr23:38]{int}[r7 r0]- 0x3 {short}
Sample.c(18:33:37:1)   _gint [lr0:0]{int} = _gint [lr0:0]{int} +
   iTemp17 [lr23:38]{int}[r7 r0]
Sample.c(15:36:42:1)   iTemp21 [lr21:38]{short}[r4] = iTemp21
   [lr21:38]{short}[r4] + 0x1 {short}
Sample.c(15:37:45:1)   iTemp23 [lr22:38]{int}[r5 r6]= iTemp23
   [lr22:38]{int}[r5 r6]- 0x1 {short}
Sample.c(19:38:47:1)   goto _forcond_0($4)
Sample.c(19:39:48:0)_forbreak_0($7) :
Sample.c(20:40:49:0)   iTemp24 [lr40:41]{short}[DPTR] = iTemp2
   [lr18:40]{short}[r2] +
                                                      ITemp11
   [lr19:40]{short}[r3]
sample.c(20:41:50:0)   ret iTemp24 [lr40:41]{short}
sample.c(20:42:51:0)_return($8) :
sample.c(20:43:52:0)   eproc _function [lr0:0]{ ia0 re0
   rm0}{function short}
```

*Cellar* web site. Code generation involves translating these operations into corresponding assembly code for the processor. This seems simple, but that's the essence of code generation. Some operations are generated in an MCU-specific manner. For example, the Z80 port doesn't use registers to pass parameters, so the Send and Recv operations won't be generated, and it doesn't support jumptables.

## ICODE EXAMPLE

This section shows some details of iCode. The example C code isn't useful, but it illustrates the intermediate code generated by the compiler. Sample.c generates the iCode sequence in Listing 3.

In addition to the operands, each iCode contains information about the file name and line it corresponds to in the source file. The first field in the listing should be interpreted as follows:

File name (line number: iCode Execution sequence number: ICode hash table key: loop depth of the iCode).

The readable form of the iCode operation is found next. Each operand of this triplet form can be of three basic types—compiler generated temporary, user-defined variable, or a constant value. Note that local variables and parameters are replaced by compiler-generated temporaries. Live ranges are computed only for temporaries. Registers are allocated for temporaries only. Operands are formatted in the following manner:

Operand name [lr live-from: live-to] {
type information} [registers allocated]

As mentioned, live ranges are computed in terms of the execution sequence of the iCodes. For example, the iTemp0 is a live `from` (i.e., first defined with execution sequence number 3) and is used last with number 5. For induction variables such as iTemp21, the live range computation extends the life from loopstart to end.

The register allocator used the live range information to allocate registers, the same registers may be used for different temporaries if their live ranges don't overlap. In addition, the allocator takes into consideration the type and usage of a temporary.

Some short-lived temporaries are allocated to special registers that have meaning to the code generator. The code generation makes use of this information to optimize a compare-and-jump iCode.

Several loop optimizations are performed by the compiler. It detects induction variables iTemp21(i) and iTemp23(j). And, the compiler does selective strength reduction (i.e., the multiplication of an induction variable in line 18 [gint = j × 3] is changed to addition, temporary iTemp17 is allocated and assigned an initial value, constant 3 is added for each loop iteration). The compiler does not change the multiplication in line 17, however, because the processor supports an 8 × 8 bit multiplication.

Note the dead code elimination optimization eliminated the dead assignments in line 7 and 8 to I and sum respectively.

## READY, SET, COMPILE

You can download the compiler at sdcc.sourceforge.net. SDCC is an active project and, as with all GPL software, many people contributed. Recently, it was retargeted for Nintendo Gameboy.

The compiler is distributed as GPL software with the hope that you'll find it useful. The SDCC team believes in continuous improvement of the software so if you have any suggestions, feel free to send me an e-mail. ■

*Sandeep Dutta is a compiler engineer working for WindRiver Systems Inc. She works on the DIAB optimizing C, C++, and Java compiler for 32-bit processors. You can reach her at sandeep.dutta@windriver.com.*

## SOFTWARE

A complete table of the iCode operations that are supported by the compiler as well as an additional sample code listing are available on the *Circuit Cellar* web site.

FEATURE ARTICLE

George Novacek

# The Joys of Writing Software

## Part 1: Battle of the Bug

No bugs. No excuses. The way George sees it, the three-finger salute should not be considered an acceptable fix for any software problem. If it means taking more time in the test stages, so be it. That's why test standards exist.

Recently, while waiting for a boarding call, I observed a pair of high-tech pacing workaholics. One had a cell phone attached to his ear, and the other wore a headset. I thought how nice it would be if airports installed cell phone jamming devices that are popular in finer restaurants. To focus on something other than their conversations, I bought a computer magazine.

But this wasn't my lucky day. I opened to an article about Windows 2000 that stated, "The new operating system is living up to its billing. But even a stable system must be expected to have many bugs...." I get emotional when it comes to shoddy workmanship, bad engineering, and defects, among which software bugs undoubtedly belong.

"Bug" is not as sinister sounding as "defect." The software industry seems to have convinced us that not only are bugs necessary, if not loveable, but that we should happily pay for their fixes! Regardless of industry experts' claims, a bug is a defect.

If we accept a bug as no big deal because it can be fixed with the infamous three-finger salute and the customer's aggravation is nothing to worry about, the next time we may feel free to leave a bug in an embedded controller where it could do serious damage. Remember the automobile cruise controllers that caused uncontrolled acceleration, or the radiation treatment machine that calculated wrong dosages?

For more than a decade, responsible engineers have used proven software development methods to guide development, integration, and testing processes to minimize defects. Today, there is no excuse not to use these methods. In this article, I will take you through the widely used software standard RTCA DO-178B (see Photo 1). Although this standard is primarily for aircraft software, the principles are the same for developing any software.

### RTCA DO-178B

As the title "Software Considerations in Airborne Systems and Equipment Certification" implies, the document addresses software certification first and development second. So, I'll work backwards to identify the development processes from the certification requirements.

Makers of safety-critical equipment, medical, nuclear energy, transportation, and weapons systems quickly realized that, although software provided unprecedented intelligence, there should be a way to control the software development process. At that time, development was a domain of a few avant-garde engineers with artistic flair. This mixture of art, black magic, late hours, Coke, pizza, and Twinkies was unpredictable and spelled trouble to the conservative industrialists.

Not surprisingly, the military was in the forefront of trying to quantify and tame this unruly, yet useful, bunch of creative people. The result was MIL-STD-2167 software development standard and several related standards dealing with software quality assurance (SQA) and the like. Engineers were assured that following the book step by step would result in certifiable software.

However, with the rapid development of software processes and engineering tools, it was impossible to keep the standard current. Then, RTCA DO-178 emerged. Instead of

saying, "Follow these rules and you'll be OK." it said, "Here are the parameters you have to satisfy. Show us how you did it and we'll tell you if we like it or not." So, the creativity, responsibility, and risk were returned to the engineers.

Knowing the parameters, it isn't difficult to determine what you need to do to satisfy them. And, you have the freedom to set up your own processes. Here's the twist. DO-178B divides software into five categories, A through E, based on criticality. A matrix in DO-178B shows which documentation is required for each category. Category E, which is the lowest criticality, has essentially none. Remember that DO-178B addresses software certification! The fact that your software does not need to be certified nor requires formal documentation doesn't mean that the development process can be shortchanged.

Whether you belong to a large engineering organization, are an independent contractor, or a hobbyist, you need a robust development process. So, delivering certifiable software is a matter of formalizing your documentation to accompany the product. Moreover, a well-defined development process guarantees a flawless product on time and within budget.

The benefit is bug-free, predictable software. No more embarrassing upgrade patches and lame explanations. To be sure, there is a degree of uncertainty in every development.

## START WITH A PLAN

When engineers succeed, it's a result of planning. Planning makes you understand the task, prepare, break it into manageable pieces, then tackle it.

DO-178B identifies five plans. If you're developing certifiable software, present these plans as formal documents. The same may be required in a large organization to make sure everybody on the team understands the job.

I advise you to put it on paper. You won't forget anything, and you'll impress customers with your professionalism. You can modify the plan quickly to fit the next projects, to include the lessons learned, and to use them during the projects as checklists.

## CERTIFICATION

Plan for Software Aspects of Certification (PSAC) is the first, top-level plan that is presented to the customer and the certification authorities before other work has started. Some of the data it contains is addressed in more detail in the other plans I'll discuss.

The PSAC should contain several parts. First, you need a system overview, including a short description of what the project will do, a block diagram, and how the functions are allocated between hardware and software.

Also, include a software overview, how it will be partitioned, which resources will be shared, and what is its safety effect. The safety effect is the most important part, it defines what may happen if the software fails.

Certification consideration provides the basis for the certification level. The software criticality levels are defined as follows:

- A—software could cause or contribute to a catastrophic failure
- B—software failure could result in a hazardous condition
- C—software failure can result in a major failure of the system
- D—software failure may cause a minor failure of the system
- E—no operational safety effect

An important part of this consideration is fault trapping and exception handling. Even if your software is bug free, you must assume that it can derail at any time in response to external influences, for example, an ESD (electrostatic discharge) to the cabinet, or in response to alpha particles.

The software life cycle defines development methods, programming languages, tools, hardware, and processes to be used during development.

A schedule is useful for the customer and you. And, leave room for special considerations like buying software modules off the shelf.

PSAC is an executive summary to communicate to the customer and, if needed, the certification authorities. It ensures that everybody understands the job before it starts. It maintains the project memory even when the original players are no longer around. If the customer and the authorities agree that the criticality level is E, no other formal documentation needs to be produced. But, even if the customer says there's no certification requirement, it's good practice to develop this plan for your own records.

Ten years later you may need to remember the basic features of the system and other special facts. No less important in our litigious society is your ability to show that you took all reasonable care while designing.

## DEVELOPMENT

Now that you have a PSAC, the Software Development Plan (SDP) gives you a roadmap to successful conclusion. Next, identify standards. Are there internal or external standards you must satisfy? Do you have copies? Do you understand them? What are the implications?

Software lifecycle description adds detail to the same section in the PSAC above so it can be implemented properly. In short, it's your plan of attack.

In the software development environment, identify the development hardware, software, development platform, tools, and so on.

## VERIFICATION

Before designing, think about how you will test the software and verify that it does its intended job. If you don't, you may find that some functions cannot be properly tested.

So, organize first. Who will do the testing? In a large organization, testing is separate from designing. In a small organization, especially for critical software, you need to show independence. That is, you can't have the designer test his own code.

You want to have regular peer and customer reviews to make sure you're on track. Traceability matrices and verification checklists are indispensable. Testing may have to be done on the target hardware only or modules are run on a separate platform.

Next, determine the availability of emulators and simulators. Do you need to verify and validate them?

Development and testing is an iterative process. If there are different people involved, at what point do you

tell the designer to stop debugging and give the module to the test people? When does it become a formal verification for credit?

If you used partitioning, how do you verify and prove its integrity? What about the compiler? Don't assume compilers come without defects. More often than not, the compiler supplier won't give you data nor source code to alleviate your concerns. You may have to spend time testing the libraries, for example.

You also should consider future recertification and retest. How are you going to handle software modifications or future updates? Having a robust development process is crucial. Identify each module affected by even the smallest change, then retest them.

Here are some lessons I learned. Always investigate an unexpected event. Second, don't change the compiler version unless you want to completely retest the software. After use, identify its version in the life cycle documentation, store it in a vault, and never use a different one on that code.

To maintain old software, use the same compiler version, although you may want to transfer it to a modern medium. Third, in a critical application where a dual redundant design is used, use two different compilers.

I lived through a C compiler horror story. During verification testing, an engineer on my team observed a unexplained event, but because he couldn't repeat it, he wrote it off as an isolated, inexplicable accident.

When we delivered the product, the fluke occurred again. With the customer questioning our credibility, we needed to explain how this could happen to certified, safety-critical software. Eventually, after going through the hex dump with a fine-tooth comb, we found the culprit. Under certain conditions, the well-known and popular C compiler had a flaw that caused it to drop DI (disable interrupt) assembler instruction if it happened to be the first line of code in the module.

The consequences of such a flaw are obvious to anyone who has written software. Upon discovery, the fix was simple. We modified our software design standards to require an NOP instruction as the first line in every module, starting with DI, to be compiled by that brand name. Now, we never let an unexplained event go. And of course, we test new compilers.

How do you protect yourself against such disasters? One way is to use Ada language, if you have $100,000 for the development system and don't mind using a 32-bit processor and lots of memory for tasks that an 8-bit processor could easily handle. Ada is the only language for which you can buy a validated compiler. You're well advised to buy compilers from a reputable, well-established company, and then test them (which also means taking apart the libraries and examining the compiled hex dump line by line). Then lock the original compilers in a vault and make them a standard.

## CONFIGURATION MANAGEMENT

Configuration or version control is an important task that cannot be left to chance. You want to impress your

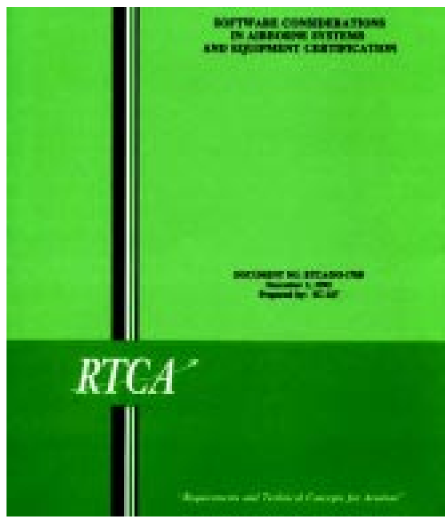



**Photo 1—***DO-178B is the worldwide standard for avionic software development and certification.*

customer by demonstrating how seriously you treat the subject. Your plan should address several issues.

How are you going to control the design activities? Version control? At this point, you should design a source code file header you may use with every module to identify the creation date, author, change history, and so on.

Determine how the software modules and revisions are identified. Let's say you're developing a program and you assign it part number 24000. It has 15 modules and you call them 24001 through 24015. You need to identify the versions with dash numbers, such as 24011-05. Every time the module is modified, the dash number is bumped up. In the end, the production will have a configuration index to compile the final software version, let's say 24000-3. The configuration index will identify that you have to link the following modules: 24001-1, 24002-17, 24003-13…24015-9.

I used only odd numbers for revision dash numbers because there are avionic systems specified for left and right hand (starboard and portboard) application. Odd numbers traditionally are used for left or both hand systems, even numbers for right hand systems only. However, you can choose either.

Problem reporting and change control are other areas that require good organization. Remember that problems may arise during testing, they also may arise in the field. When a new version is released for testing, ensure that it has a version number that no one can modify without formal approval and documentation.

If a problem is found during testing, a report is created; this can be on paper or a database entry. Record the problem, symptoms, cause, and solution for future reference. Determine if other modules could be affected and test them again.

Archive the life cycle environment and data control. You want to be able to repeat a test 10 years from now under identical conditions. Every piece of even a homemade test fixture should have a number and a sketch so that they can be maintained or reproduced. Before CD-ROMs, engineers archived data on floppies. Each floppy was produced twice, one stored on the premises, the other off. And every two years, the floppies were written again to refresh the magnetic record.

No less important is application programs storage. Many data formats changed during the past decade, now some of them are incompatible with their older versions. Despite promises of a paperless office, sometimes having a hard copy is the only safe way. This is the case with many EDA (engineering design automation) tools.

## QUALITY ASSURANCE

Often, the configuration control and software quality assurance (SQA) plans can be combined, because the former identifies necessary activities, and the latter shows how compliance will be monitored.

QA is responsible for neither software testing nor execution of the identified development tasks. QA is the industrial cop whose purpose is auditing what's done and verifying it's accomplished by the book. If you're working alone, it's easiest to show your employer's QA that you'll be accountable with checklists and traceability matrices. If you keep them updated, the audits won't be painful.

## CHECKS AND BALANCES

I mentioned checklists and traceability matrices several times. What are they? The checklists help you remember the numerous tasks to watch during development, while

providing a written record for the customer's auditor. Although simplistic, checklists are invaluable.

For example, let's assume you plan to develop a home control software. When you start, regularly check a few things on your checklist. Have you backed up the data? Do the module headers indicate the necessary information? The checklist should cover completion of every task and subsequent start of the next phase.

Some of these things may appear superfluous, especially if you are not depending on the success of the project for your next paycheck. But, by remaining disciplined, you're saving time for future debugging while learning to be a better programmer.

The traceability matrix enables you to trace the specification requirements through the entire process, to the source code and back. I'll discuss this in Part 3, when I look closer at the design process. It's sufficient to understand that the traceability matrix is essentially a database of requirements and their implementation.

In a well-maintained matrix, you must be able to pick a customer requirement, follow it through the design process to understand its implementation, and end up with the source code lines that perform the function. Similarly, picking up any line of source code, you should be able to trace it to the customer's spec and understand why it's there. The last step between source and executable code traceability is not needed as long as you pre-test your compiler.

Tracing between source and executable code isn't easy today when high-level languages are prevalent. Most embedded software is written in C. Assembler source code is difficult to certify because of its low readability. Authorities frown at it, allowing it only in the absence of other choices, mainly for execution speed.

Traceability matrix is a perfect tool for proving to the customer full compliance with the specification and for verifying tests. In case of changes, it identifies all code that will be affected. Tools create and maintain the matrix automatically. Tools also exist for automatic version control and test coverage determination, making the software design a more predictable process, without human error.

## WHAT'S NEXT?

In this article, I covered the front-end work for developing a stable, predictable software design process. The plans can be customized to accommodate new specifications and lessons learned. In Part 3, I'll discuss the design process itself.

Critics argue that this process produces bloated, slow code, and still is plagued with bugs. I disagree. Likely, the code will be larger, but it won't be bloated. The greater memory requirement is a small price to pay for the structure, which will give you the correct environment for generating code without defects.

If your ambition doesn't reach beyond writing small, single purpose, non-critical programs, and you don't mind an occasional frustrating debugging, you'll be OK forgetting everything you read here and sticking to your haphazard code. But if your ambition aims higher, stay tuned! ■

*George Novacek has 30 years of experience in circuit design and embedded controllers. He currently is the general manager of Messier-Dowty Electronics, a division of Messier-Dowty International, the world's largest manufacturer of landing-gear systems. You may reach him at gnovacek@nexicom.net.*

## REFERENCES

[1] "Software Considerations In Airborne Systems and Equipment Certification", RTCA/DO-178B, RTCA Inc., 1140 Connecticut Ave., Washington D.C., 1992.

[2] M. R. Lyu, *Handbook of Software Reliability Engineering*, IEEE Computer Society Press, Los Alamitos, CA, 1995.

[3] B. Beizer, *Software Testing Techniques*, Van Nostrand-Reinhold, New York, 1990.

[4] A. Ralston, ed., *Encyclopedia of Computer Science and Engineering, Second Edition*, Van Nostrand Reinhold Co., New York, 1983.

# NOUVEAU*PC*

Edited by Harv Weiner

## 486DX PC/104 PLUS MODULE

The **CPU-1410** is an embedded 486 AT computer in a PC/104-plus form factor. The board is PC/104-plus compliant and can be expanded with other PC/104 or PC/104-plus modules. High-integration enables the board to be used as an SBC in embedded applications such as industrial terminals and automobile navigation devices.

The module integrates a 486DX-75/100-MHz CPU with 32 MB of DRAM. Interfaces include two serial ports, parallel port (with an optional floppy disk controller), IDE, SVGA, PAL/NTSC TV-OUT, three timers, and a keyboard port. Other onboard functions include an SSD socket with up to 144 MB of solid state disk space, watchdog timer, and real-time clock.

The BIOS is in a flash EPROM and is onboard programmable. Setup parameters also are saved in flash memory, allowing the module to operate without a battery. The flash BIOS can store 1 MB, 128 Kb of which is used for the BIOS and its extensions. The remainder can be used as a read-only disk and to store the operating system, user programs, and data.

The CPU-1410 supports all operating systems available for standard PC platforms, including: DOS, ROMDOS, Windows 3.11, 95, 98, 2000 and NT, Linux, as well as real-time operating systems like QNX, pSOS, PharLap, VxWork, WinCE, and RTLinux.

**Eurotech S.p.a.**
**+39-0433-486258**
**Fax : +39-0433-486263**
**www.eurotech.it**

## CELERON ATX MOTHERBOARD

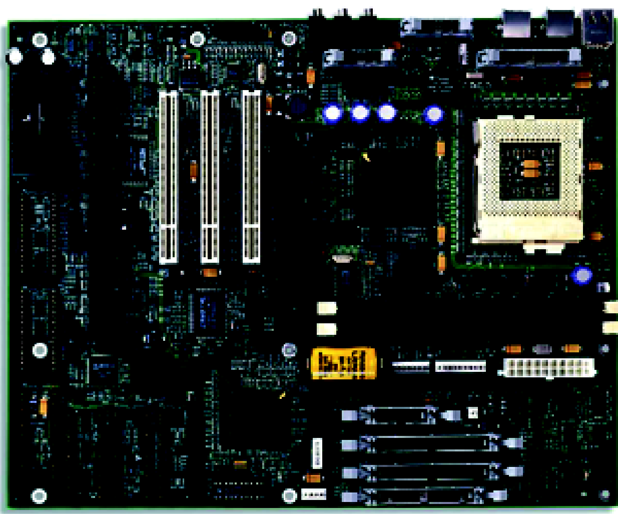
The **ATX-C440** is an industrial-grade motherboard that supports an Intel Celeron microprocessor. It's a standard ATX motherboard designed for OEMs in the industrial and embedded markets. The ATX-C440 will be available for up to five years with consistent form and features.

Based on the Intel 440BX chipset, ATX-C440 supports socketed Celeron processors to 500 MHz, with a 66-MHz bus, and up to 256 MB of SDRAM. The motherboard conforms to the ATX form factor and has one 16-bit ISA slot, three 32-bit PCI slots, and one 2X AGP slot. It has a SoundBlaster Pro-compatible 15-W-per-channel PCI sound system, two USB ports with keyboard support, two serial ports (optional RS-232 with RS-485), one enhanced parallel port, a dual floppy port, and two independent Ultra DMA-33 EIDE ports.

Features include a watchdog timer, serial EEPROM for storing configuration data, battery monitor circuit, industrial grade latching connectors, AC-power failure detection via NMI, flash memory disk support, static RAM support, and a BIOS you can customize.

The ATX-C440 with 400-MHz Celeron CPU is available for **$420** in quantities of 100.

**Adastra Systems**
**(510) 732-6900**
**Fax: (510) 732-7655**
**www.adastra.com**



# NOUVEAU *PC*

## PC/104 HIGH-DENSITY DIGITAL I/O MODULE

The **DIO96-104** provides 96 TTL/CMOS compatible digital I/O channels arranged as four 24-bit groups. Each group is divided into three 8-bit ports and controlled by a separate 82C55A peripheral interface chip. This chip offers flexible configurations including software programmable port directions and strobed I/O handshaking. Pull-up and pull-down resistors are absent, so the user's circuitry dictates how each channel will be handled during reset and input modes.

External devices connect to the DIO96-104 through four identical, 26-pin IDC, flat-ribbon headers that include access to the host's 5 V and GND for powering external circuitry. All channels default to high-impedance inputs during system reset. The DIO96-104 occupies 16 consecutive locations within the host computer's I/O map, and the starting address is jumper selectable for any value between 0X000 and 0x3f0. The module conforms to the PC/104 (IEEE-996) standard and operates on a single 5-V power supply.

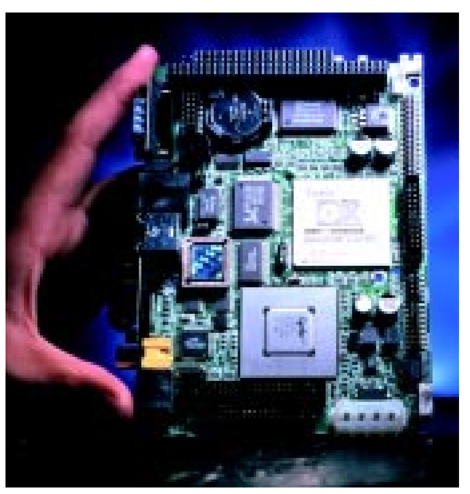
A standard J1/P1 stack-through connector allows the DIO96-104 to reside anywhere within an 8-bit PC/104 stack. Adding an optional J2/P2 connector provides 16-bit stack-through compatibility.

The DIO96-104 costs **$119** in quantities of 100.

**Scidyne**
**(781) 293-3059**
**Fax: (781) 293-4034**
**www.scidyne.com**

## SBC WITH TV OUTPUT

The **PCM-5822** is a half-size single board computer that features a TV-out function, new switching power regulator, and low-power NS GXMLV-200/ 2.2-V processor. Other onboard features include audio interface and controller, CompactFlash card socket, watchdog timer, 10/100 Base-T Ethernet, and support for VGA/LCD and LVDS (low-voltage differential signal).



Because the CPU is mounted directly on the board, there is no need to set jumpers for speed or voltage differences. The CPU works in environments up to 60°C without a fan. For convenience, all cables connect to the front panel. AV and S-Video connectors are also provided on the front panel.

The compact unit fits in the space of a 3.5″ HDD and accommodates ISA-bus expansion with an onboard PC/104 connector. The unit provides a TV-out function in NTSC and PAL formats. The AWARD BIOS has 256 KB of flash memory, and there is one 144-pin SO-DIMM socket that accepts up to 128 MB of SDRAM. An Enhanced IDE HDD interface supports up to two enhanced IDE devices. Two floppy disk drives are also supported.

I/O consists of one parallel port, RS-232, RS-232/422/485 serial port, infrared port, two USB connectors, and connectors for keyboard and PS/2 mouse. The unit's power management is APM 1.1 compliant, and a 104-pin, 16-bit PC/104 module connector is included.

Pricing for the PCM-5822 starts at **$392**.

**Advantech Technologies, Inc.**
**(949) 789-7178**
**Fax: (949) 789-7179**
**www.advantech.com/epc**

Ingo Cyliax

# Real-Time Executive for Multiprocessor Systems

## Part 4: Debugging

It's time to wrap up this series on RTEMS, so Ingo shows us what it takes to debug an RTEMS application. Working with the run-time debugging environment and the GNU debugger makes the process even easier.

For the last few months, I have been exploring RTEMS, an open-source licensed real-time environment from OAR Corporation. RTEMS runs on a multitude of architectures and platforms and implements a TCP/IP protocol stack and an embedded web server.

RTEMS also includes a run-time debugging environment that can be used with the GNU Debugger (GDB). This setup lets you debug multitasking applications under RTEMS from a remote host.

### GNU DEBUGGER

Let's start with GDB. GDB is a source-level debugger that is available under the GNU license. It can be downloaded from the 'Net and is bundled with most Linux distributions. With GDB, you can either start applications or debug already running applications by attaching to them. You can also do post-mortem analysis of crashed programs or systems.

GDB can be built for various 32-bit processor architectures. For this kind of project, however, you typically need to get the source code from one of the GNU code repositories (e.g., gnu.prep.ai.mit.edu) and compile it for your specific cross target. GDB will run on most Unix and Unix-like operating systems. It even runs under Windows, using the Cygwin environment from Cygnus, which is now part of RedHat. Cygwin is a run-time environment that allows you to port and run Unix- and Linux-type applications under 32-bit Windows.

GDB also supports various loader and symbol table formats such as `Coff`, `a.out`, and `ELF`. It uses the GNU `binutil` package, the same library that the GNU C compiler and linker (as well as various other GNU tools) use to access and manipulate object files and libraries. By using the `binutils` library, it's fairly easy to add a new object file format if you have all of the information.

One of the best features of GDB is its remote debugger interface. If you are debugging native applications in a Unix or Linux environment, GDB uses the `ptrace()` system call. With `ptrace()`, a process can attach to another process and perform functions including reading and writing memory registers, starting and stopping the process, setting breakpoints, and tracing and receiving signals.

**Photo 1**—*GDB is a command line-based source-level debugger. Command line-based editors work well in situations where you may only have a simple terminal interface.*

GDB's remote debugging environment provides an application programming interface (API) for targets that mimic the functionality of the `ptrace` calls. For example, you can implement routines that will read registers from a debug target. Listing 1 shows all of the functions that the API supports.

A traditional way to use this API is to write a protocol module that understands how to talk to a ROM-level debugger via a serial link. In this case, you implement smaller functions, like read and write memory and registers by talking to the ROM debugger over the serial link. Several ROM debugger protocols have already been implemented in the GDB tool set. These are automatically included, if appropriate, for a particular target architecture.

GDB also provides a generic remote protocol module. This module implements a generic debugging protocol and is included in all implementation of GDB. It can be run through serial links or network connections. GBD also includes a sample remote client module that can be used to remotely debug a Unix application program. This module can be adapted to run in an embedded client as well.

The remote debugger interface is useful for interfacing with applications on a variety of targets. However, it is also easy to implement simulators to architectures. In this case, the target protocol module is an interface to an instruction-based simulator for the target architecture. Several simulators exist in the default GDB tool sources.

Add to this the fact that GDB will run on many popular host environments like Solaris, Linux, or even Windows (using Cygwin), and the source code is available for the debugger and existing protocol modules, and you have the recipe for a popular debugging environment. Photo 1 shows GDB running in a text window, and Photo 2 shows GDB running in a simple X Window interface.

As it stands, GDB can be used to debug embedded systems using the remote debugger interface and a ROM-based debugger, such as those available in an evaluation module. On

**Listing 1**—*Here is the list of functions to implement a remote debugger in GDB. There are basic debugging functions like register, memory, and symbol table interfaces, but GDB can support multiple tasks in a target.*

```
to_open()
to_close()
to_attach()
to_detach()
to_resume()
to_wait()
to_fetch_registers()
to_store_registers()
to_prepare_to_store()
to_xfer_memory()
to_insert_breakpoint()
to_remove_breakpoint()
to_terminal_init()
to_terminal_inferior()
to_terminal_ours_for_output()
to_terminal_ours()
to_terminal_info()
to_kill()
to_load()
to_lookup_symbol()
to_create_inferior()
to_mourn_inferior()
to_can_run()
to_notice_signals()
to_thread_alive()
to_stop()
```

**Listing 2**—*Here is a short section of the RPC descriptions file for* `rdbg`*, a remote network debugger implementation for RTEMS. It's meant to give you the flavor of the C-like syntax used.*

```
/* Data for GET_TEXT_DATA */
struct get_text_data_in {
  int           pid;            /*process/actor id if non-zero */
  string        actorName<16>; /*actor name for system mode */
};

struct get_text_data_out {
    int         result;
    int         errNo;
    u_long      textStart;
    u_long      textSize;
    u_long      dataStart;
    u_long      dataSize;
};

                                /* Data for GET_SIGNAL_NAMES */
struct one_signal {
    u_int       number;
    string      name<>;
};
```

**(continued)**

**Listing 2**—*continued*

```
typedef one_signal all_signals<>;

struct get_signal_names_out {
    all_signals signals;
};

%  /* now define the actual calls we support */

program REMOTEDEB {
        version REMOTEVERS {

                /* open a connection to server or router */
                open_out
                OPEN_CONNEX(open_in)            = 1;

                /* send a signal to a process */
                signal_out
                SEND_SIGNAL(signal_in)          = 2;

        /* all routines below require a connection first */

                /* close the connection to the server */
                void
                CLOSE_CONNEX(close_in)          = 10;

                /* process ptrace request */
                ptrace_out
                PTRACE(ptrace_in)               = 11;

                /* poll for status of process */
                wait_out
                WAIT_INFO(wait_in)              = 13;

                get_signal_names_out
                GET_SIGNAL_NAMES(void)          = 17;

        } = 2;                  /* now version 2 */
} = 0x20000fff;
```

some platforms, you can even use on-chip debuggers like BDM for Motorola processors.

Background Debug Mode (BDM) is a hardware debugger that is included on many Motorola 32-bit processors, such as MC683*xx* and ColdFire (see articles by Craig Haller and myself about BDM in *Circuit Cellar* 89). A bit serial interface on the chip can access registers and memory and various break- and tracepoints. With GDB and a BDM-to-serial interface, GDB can be used to download and debug programs on these processors.

However, I have been looking at running RTEMS on the standard Intel PC/AT platform, which doesn't come with a standard ROM-based debugger. Here, I can use the RTEMS debugger server (`rdbg`) module that comes with RTEMS. Let's look at this environment in more detail.

## RTEMS DEBUGGER SERVER

The clever folks at Cannon Research Center France have implemented a Sun RPC-based debugger module for RTEMS. Sun RPC is a remote procedure call protocol developed by Sun Microsystems. It can be run on top of UDP, which is nice if you want to build small protocol stacks. Sun RPC is freely available and has been implemented on many operating systems.

Sun RPC applications are built by specifying data types and procedure interfaces that will be used by a client program to communicate with a server program. In this case, the GDB is the client and the server is the module that lives in RTEMS.

A program (`rpcgen`) is then used to compile the description into two C modules. One is for the client program and the other is for the server application. The client application simply calls the routines, which are then encoded to be sent over the network and executed in the corresponding function in the server.

So why use this? The designers figure that because Sun RPC is standard and public, it is a good way to specify a network protocol. The designers also have had experience with VRTX and Chorus, which are commercial RTOSs. These also use a Sun RPC-based debugger interface with GDB.

By implementing a network-based remote debugging environment, it is possible to get good throughput when transferring memory and register contents between the application and debugger. Another advantage to network-based debugging comes when you're developing multiprocessor systems. It's fairly simple to switch the context of GDB from debugging one application on one processor to another (no serial cables to switch from one board to another). You can even do this on shared memory multiprocessors by implementing Sun RPC through shared memory message passing.

Let's take a look at the `rdbg` module. `rdbg` is implemented as a library that is linked to your application. In this case, all you do is call the function `rtems_rdbg_initialize (void)` from your application program and it installs itself. Also, you need to initialize the network module, which I ran through in last month's installment.

When `rtems_rdbg_initialize()` is called, `rdbg` creates a couple of sockets and starts a RPC UDP server. This server simply waits for RPCs from the client using the server module created by `rpcgen` and processes them. The rest of the module deals with implementing the RPC calls. This involves translating exceptions, such as breakpoint, and dividing by zero to signals, which is what GDB likes to deal with.

Clearly, dealing with processor exception involves some architecture-dependent code in the `rdbg` module. So far, these routines have been implemented for PowerPC and i386 targets.

## NO WORKIE...

I was eager to give this a try. Moving on, I built the `rdbg` target libraries by specifying `enable-rdbg` in the configuration command line. Referring to the previous articles, remember that RTEMs uses the GNU auto configuration scheme. You call a configuration script that dynamically configures software for your specific environment.

`Configure` uses command line options to override default choices. This might include the location where you want the application installed and the type of board support package (BSP) to build the system (pc386). By default, `rdbg` support is off so I had to reconfigure the system with `rdbg` support turned on.

I then recompiled one of the demo programs from last month by adding the `rtems_rdbg_init()` call right after the `init` call to the network protocol stack. The programs compiled and linked fine, so I copied it to the boot disk to run. The debugger did not interfere with the normal operation of the application, as you would expect.

However, there was one gotcha—I didn't have a GDB with the Sun RPC calls enabled and I couldn't figure out how to rebuild GDB to include the calls. It seemed there was some unexplained magic in the document that comes with the system. I did spend some time figuring out that I didn't have the support in the GDB I downloaded as a pre-built application from OAR's web site. Then I tried to figure out how to get it. I finally ran out of time. At this point, I had to punt and come up with a different way of doing my debugging.

In a way, writing for *Circuit Cellar* is much like doing consulting work. There is an exact deadline (i.e., the article is due!) and all of the work needs to be completed and written on time, or else it won't make it into the next issue.

When I was scouring through the sources trying to figure out how to make the RPC-based debugger work, I did find that the pc386 BSP supports

| Command | Description |
|---|---|
| break [file:]function | Set a breakpoint at function (in file) |
| run [arglist] | Start your program (with arglist, if specified) |
| bt | Backtrace: display the program stack |
| print expr | Display the value of an expression |
| c | Continue running your program (after stopping at a breakpoint, etc.) |
| next | Execute next program line (after stopping); step over any function calls in the line |
| step | Execute next program line (after stopping); step into any function calls in the line |
| help [name] | Show information about GDB command name, or general information about using GDB |
| quit | Exit from GDB |

**Table 1**—*These are the basic GDB commands. GDB also has an interactive help facility and a powerful macro language to define your own commands.*

**Listing 3**—*Here's the code for initializing serial port-based GDB support. It senses which serial port is used for the console and uses the first available serial port for the debugger port. You then simply connect your target and host via a null-modem cable, and voilà!*

```
/* Init GDB glue  */

  if(BSPConsolePort != BSP_UART_COM2)
    {
      /*
       * If com2 is not used as console use it for
       * debugging
       */
      i386_stub_glue_init(BSP_UART_COM2);
    }
  else
    {
      /* Otherwise use com1 */
      i386_stub_glue_init(BSP_UART_COM1);
    }

  /* Init GDB stub itself */
  set_debug_traps();

  /*
   * Init GDB break in capability,
   * has to be called after
   * set_debug_traps
   */
  i386_stub_glue_init_breakin();

  /* Put breakpoint in */
  breakpoint();
```

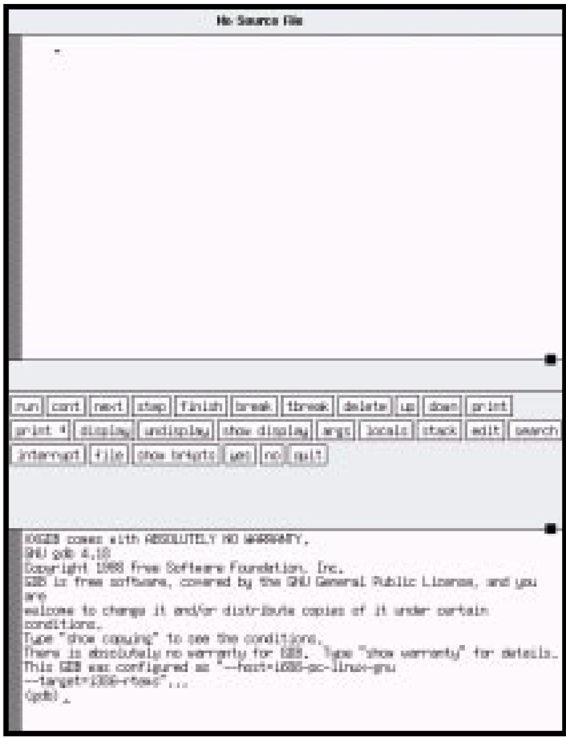



**Photo 2**—*Several GUIs exist for GDB to make it easier to use than the command line version. The sample that you see here is a simple X Window interface (xxgdb).*

the standard GDB serial-based debugger protocol. It's a bit more involved because you have to add the code in Listing 3 to the program.

The code in Listing 3 figures out which serial port to use, employing the first one available—COM2 if the console is on VGA/KBD, and COM1 if the system console is already on COM2. With a null-modem cable, the target then can be debugged by a host running an i386-aware GDB debugger using the standard GDN remote debugger interface.

On the host, you start up GDB with the executable image to read the symbol table and then point the debugger to the serial port where it will communicate with the remote debugger stub running on the target:

```
% i396-rtems-gdb —nx o-opti-
  mize/netdemo.exe  ...   set
  remotebaud 38400  target
  remote /dev/ttyS1 ...
```

After everything is up and running, you can perform a variety of tasks. Table 1 gives you a brief summary of the commands that are available in GDB.

Although using the serial-based debugger approach isn't as neat as the network-based debugger, at least now I have it working. Also, going through the exercise of trying to get the network-based debugger to work brings a few points to light. For instance, beta software usually lacks some of the finer points in the documentation. You shouldn't wait until the last minute to muck with new software. Also, having source code is nice because you can figure out how things really work, rather then relying on documentation. Finally, I don't need to tell you, but it's always good to have a backup plan.

Thanks to Emmanuel Raguet and Eric Valette of Canon Research Center in France S.A. I do plan to revisit the network-based remote debugger, perhaps when RTEMS version 4.5.0 is released.

*Ingo Cyliax is the Sr. Hardware Engineer at Derivation Systems Inc. (DSI) where he designs and builds embedded systems and hardware components. DSI is the leader in formally synthesized FPGA cores and specializes in embedded Java technology. Ingo has been writing on various topics ranging from real-time operating systems to nuts-and-bolts hardware issues for several years.*

## REFERENCES

M.K. Johnson and E.W. Troan, *Linux Application Development*, Addison-Wesley, Reading, MA, 1998.

R. Stallman, *Debugging with GDB Version 4.17*.

## SOURCE

OAR Corporation
(256) 722-9985
Fax: (256) 722-0985
www.oarcorp.com

EPC
Applied PCs

Fred Eady

# Embedded Kiosk or Mission Impossible?

This project began as a challenge that even Fred couldn't resist (er, avoid). Hidden keyboards, limited access, multiple screens—the mission had the trimmings of a certified goose chase. The Florida-room clock was ticking….

If you've ever worked in the commercial sector, you know that the marketing folks dictate the direction of the product line. If you are a design engineer in the commercial world, you also know that sometimes the engineering requests made by the marketeers (not to be confused with mouseketeers) can be a little out there. In this article, I'm going to show you one time when the marketing department was absolutely right.

Have you ever been walking through a mall or airport and reached into your pocket, only to pull out a couple of pennies and a quarter? It's rather difficult to buy a soda or go to a movie for $0.27 these days. And, if you've just gotten off a flight and your car is parked in the airport pay-to-park lot that doesn't take credit cards, you've just extended your trip and may be walking a bit further than you planned. Hopefully, in either situation, you have your trusty ATM card in the other pocket.

A few keystrokes later, you are solvent and can afford to get your car out of hock or see "Godzilla vs. Mothra" with a soda *and* popcorn.

What if you get off that plane in a foreign city and rent a car to travel to your final destination? It would be nice to know how to get there, wouldn't it? Again, a few keystrokes at the rental counter and, lo and behold, a map instantly appears.

I see your lips moving. You're muttering, "Okay, Fred, what does using your ATM card have to do with generating a map at the rental car counter?" The answer is that in both scenarios, the user was bailed out of a bad situation by a kiosk-like device.

Although you may not equate kiosks with ATMs or map generators, in a basic sense, all of these devices are kiosks in one form or another. The typical kiosk is simply a window into a database. For instance, today's ATMs allow you to get just about any information you desire about your account status. Where is this account status? It's in the bank's database. The same goes for the generated map at the rental counter. Database here, database there.

Not all kiosks are created equal. Some kiosks have mutated to accommodate web browsing. And, instead of

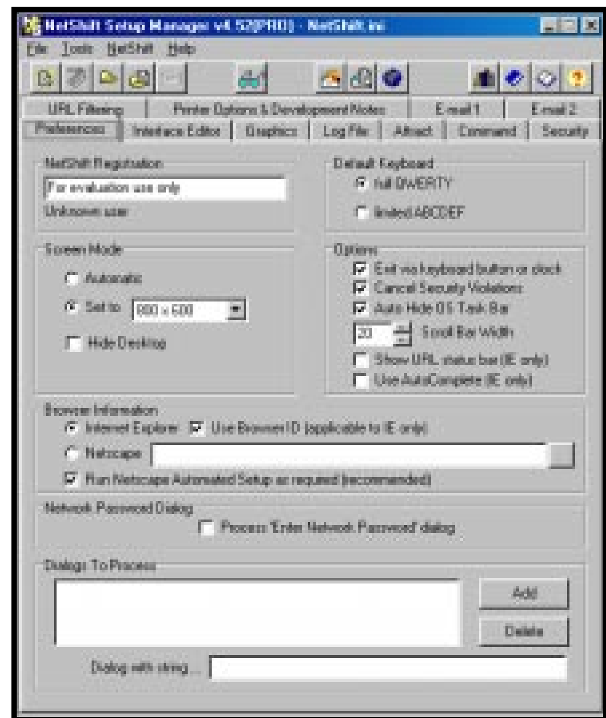


Photo 1—*This is a busy window. Note the lean towards Bill's Internet Explorer.*

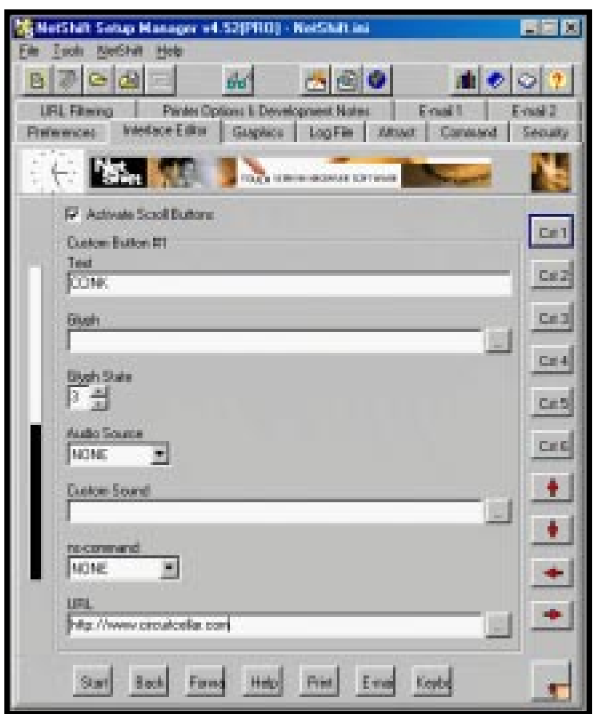



**Photo 2—***A Glyph is actually a file name and path pointing to a button graphic.*

getting data from a remote database, some kiosks are now collecting data to be added to a database. In fact, it has been determined that a public kiosk that is located on business premises can be used later to sample and rate the success or failure of that business's products and services. This is similar to the suggestion cards you see in restaurants.

People love to push buttons on machines. The kiosk feeds on this. While you're having fun punching buttons, your answers or suggestions are being logged to spot trends that could lead to increased sales volumes or higher customer loyalty.

## MR. PHELPS

I think you know where I'm headed on this. This time around, I'll be embedding a kiosk solution. However, the marketing department has made a "request." I've got to run not one, but two screens. Unfortunately, it's not as easy as it sounds. I've got to do this with not two, but one embedded computer. And, by the way, I'm using some software from an outfit called NetShift. "If you can't get that to work, write your own code."

I could hear that music from "Mission Impossible" as I accepted the kiosk mission from hell. The problem is, if I fail, I don't think anybody's going to disavow any of my actions. Let's just hope that this project doesn't self-destruct in five seconds.

The idea of two kiosk screens is a clever idea. The customer wants to have a public kiosk in the main area of their business and another in the employee break room. Not only does having two kiosks on a single embedded computer save money on hardware, the second employee-only kiosk saves administrative time by allowing employees to query human resource databases from the kiosk station's employee-only screen. Instead of going to the main office to get a copy of a W-2, the employee could call it up at the kiosk site and print it there. Need a dental form? Get it at work from the employee kiosk. Want to change your 401K options? Use the company kiosk and intranet, and do it yourself. Sometimes those guys and gals in the marketing department use their heads for something other than the perfect hairstyle.

**Photo 3—***This screen is a product of URL filtering. If the URL is not on the list, you can't get there from here.*

**Listing 1—***No kidding. This is all there is to it!*

```
     ' MODULE .BAS CODE BEGINS HERE

   Public Type STARTUPINFO
           cb As Long
           lpReserved As String
           lpDesktop As String
           lpTitle As String
           dwX As Long
           dwY As Long
           dwXSize As Long
           dwYSize As Long
           dwXCountChars As Long
           dwYCountChars As Long
           dwFillAttribute As Long
           dwFlags As Long
           wShowWindow As Integer
           cbReserved2 As Integer
           lpReserved2 As Long
           hStdInput As Long
           hStdOutput As Long
           hStdError As Long
   End Type
   Public Type PROCESS_INFORMATION
           hProcess As Long
           hThread As Long
           dwProcessId As Long
           dwThreadId As Long
   End Type
   Public Declare Function CreateProcessA Lib "kernel32" (ByVal
lpApplicationName As Long, ByVal lpCommandLine As String, ByVal
lpProcessAttributes As Long, ByVal lpThreadAttributes As Long,
ByVal bInheritHandles As Long, ByVal dwCreationFlags As Long,
ByVal lpEnvironment As Long, ByVal lpCurrentDirectory As Long,
lpStartupInfo As STARTUPINFO, lpProcessInformation As
PROCESS_INFORMATION) As Long
   Public Declare Function PostThreadMessage Lib "user32" Alias
"PostThreadMessageA" (ByVal idThread As Long, ByVal msg As Long,
ByVal wParam As Long, ByVal lParam As Long) As Long
   Public Declare Function RegisterWindowMessage Lib "user32"
Alias "RegisterWindowMessageA" (ByVal lpString As String) As Long
   Public Declare Function WaitForInputIdle Lib "user32" (ByVal
hProcess As Long, ByVal dwMilliseconds As Long) As Long

   Public NameOfProc As PROCESS_INFORMATION
   Public NameStart As STARTUPINFO

     ' Dimension variables for Keyon message registration

   Public WM_KEYON_CLOSE As Long
   Public WM_KEYON_HIDE As Long
   Public WM_KEYON_LEFT As Long
   Public WM_KEYON_MAX As Long
   Public WM_KEYON_MIN As Long
   Public WM_KEYON_NOTOP As Long
   Public WM_KEYON_ONTOP As Long
   Public WM_KEYON_SHOW As Long
   Public WM_KEYON_TOP As Long

     ' Const for priority of Keyon Engine process

   Public Const NORMAL_PRIORITY_CLASS = &H20

     ' Define constants for registering Keyon messages. These
     ' could be literals in the code as well.

   Public Const KEYON_HIDE As String = "KEYON HIDE"
   Public Const KEYON_SHOW As String = "KEYON SHOW"
   Public Const KEYON_MIN As String = "KEYON MIN"
   Public Const KEYON_MAX As String = "KEYON MAX"
   Public Const KEYON_ONTOP As String = "KEYON ONTOP"
```

**(continued)**

## NETSHIFT

By the way, because this project is experimental, it may not work. Marketing has mandated that I use the eval copies of NetShift and its utilities. I'm sinking in quicksand already.

The first task is to find an evaluation copy of NetShift. But, at this point, I don't even know what NetShift is. Off to the 'Net.

It seems logical enough to try www.netshift.com in my search process, right? Bingo...the NetShift homepage. The first buttons I see are Introduction and Downloads. Not only can I get "edumacated" (that's Southern for "familiar with"), I can get my evaluation copy of the software, too.

As it turns out, NetShift is a dedicated kiosk program that runs under Bill's Win98 and WinNT. For security reasons, the NetShift software literally takes over the computer. This means that the average user can't hack his or her way into the kiosk and damage any underlying computing infrastructure back at the home office. NetShift allows the kiosk user to e-mail and even print when the kiosk application requires it. Of course, this is all done via touchscreen.

Putting together a good-looking kiosk with NetShift is easy. The NetShift evaluation package comes with a setup manager program (see Photo 1). As you see, there's a tab for every function that NetShift can perform. Keep in mind that this is the development stage and some of the options you see in the Preferences window will not be employed in the final kiosk design. For instance, "Exit via keyboard button or clock" will not be allowed in the final instance of NetShift.

Under the Interface Editor tab is where the kiosk rubber meets the road. This window allows the kiosk designer to lay out the basic framework of how the kiosk will look and act as far as the user is concerned. As you see in Photo 2, the NetShift kiosk screen is composed of four outer frames that contain the buttons and banners. Clicking on the "Cst 1" button reveals the properties that can be modified to make the button unique. I named it CCINK and blatantly set the URL to the *Circuit Cellar* web site. Notice at the top of Photo 2 that the information I am entering is kept in a `NetShift.ini` file.

After saving the information into the startup configuration `NetShift.ini` file, I clicked on the new CCINK button, only to be denied! Imagine the dismay of the *Circuit Cellar* webmaster, having to explain the contents of Photo 3 to staffers.

The good news is that *Circuit Cellar*'s web site is operating fine. By design, it's NetShift that is throwing a wrench into the works. Do you see an entry for *Circuit Cellar*'s web site in Photo 4? I didn't think so. After adding the site to the allowed viewing list and attempting to click on CCINK once more, Steve and company finally showed up (see Photo 5).

At this point, note that Photo 5 gives you a good look at NetShift. To affect a final kiosk design, the designer needs to put together some

Listing 1—*continued*

```
Public Const KEYON_NOTOP As String = “KEYON NOTOP”
Public Const KEYON_CLOSE As String = “KEYON CLOSE”
Public Const KEYON_LEFT As String = “KEYON LEFT”
Public Const KEYON_TOP As String = “KEYON TOP”

   ‘ The API’s are functions, so we need a variable to check
   ‘ the return code

Public RC As Long

   ‘ EXECUTABLE CODE BEGINS HERE
Dim kbflag As Boolean

Function RegisterKeyonMessages()
WM_KEYON_HIDE = RegisterWindowMessage(KEYON_HIDE)
WM_KEYON_SHOW = RegisterWindowMessage(KEYON_SHOW)
WM_KEYON_MIN = RegisterWindowMessage(KEYON_MIN)
WM_KEYON_MAX = RegisterWindowMessage(KEYON_MAX)
WM_KEYON_ONTOP = RegisterWindowMessage(KEYON_ONTOP)
WM_KEYON_NOTOP = RegisterWindowMessage(KEYON_NOTOP)
WM_KEYON_CLOSE = RegisterWindowMessage(KEYON_CLOSE)
WM_KEYON_LEFT = RegisterWindowMessage(KEYON_LEFT)
WM_KEYON_TOP = RegisterWindowMessage(KEYON_TOP)
End Function

Private Sub btnback_Click()
On Error Resume Next
WebBrowser1.GoBack
End Sub

Private Sub Form_Load()
```

**(continued)**

fancy bit-mapped buttons and graphics to place into the four areas of the finished NetShift kiosk display.

## LOOKS ARE DECEIVING

No one’s denying that NetShift is a neat site, but I’ve got to run two instances of it to satisfy the marketing reps. To prove the concept, I got on the web and ordered a Matrox G200 Multi-Monitor card. Then, from the Florida-room lab, I pulled out a Pentium-based desktop and a couple of MicroTouch-enabled touchscreen monitors.

As you know, Win98 supports two monitors. Although WinNT does not do this natively, the Matrox driver coaxes NT into that position. WinNT is a consideration as far as security and OS flexibility are concerned, but I won’t use it in the design phase. There are a couple of ways to control the dual-monitor configuration under Win98. Of course, the Matrox way is to use their dual-headed card, the G200. Another route would be to use the host computer’s onboard video hardware and an additional SVGA

card or I could try two SVGA cards if the host system board does not contain its own SVGA hardware.

I tried all three methods. The cleanest method is to use the Matrox card. In fact, stacking dual-headed Matrox cards allows twice as many monitors as the number of cards you can stuff into your computer. We only need two, thank you.

Win98 was loaded with the Matrox drivers and card. I then attached the touchscreens, fired up the whole thing, and kicked off NetShift. The NetShift program opened and quickly placed itself in the first touchscreen. Touchscreen 2 did nothing, so it was time to do some digging.

After careful investigation and lots of web browsing, I decided that I needed to expand the screen size. NetShift recommends 800 × 600. Experimenting with WinNT and the Matrox card, I remembered seeing a good graphic depiction of how the screens logically looked to NT. This graphic is not present in Win98 and is part of the Matrox WinNT utility set I loaded with the drivers.

To make this work, think of the two touchscreens set up side by side as one large touchscreen of 1600 × 600. Touchscreen 1 is 800 × 600. Touchscreen 2 starts at 800 through 1600 horizontally. With that knowledge, I put some quick and dirty VB code together to see if I could place an application in the Touchscreen 2 area. Attempted and achieved! Now the only thing left to do is put NetShift in both touchscreen areas. But alas, NetShift only runs in Touchscreen 1 space. To add insult to injury, only one NetShift instance can be initiated. To the phone….

There I met with another gotcha. NetShift is headquartered in the UK. So, instead of calling, I proceeded to the support e-mail. I must admit I did get a fast response from their technical desk, yet with a very short answer—no. To be exact, "No, Fred, you can't run two instances of NetShift. That defeats the purpose of a kiosk and has potential security problems as well." You all know me, so in characteristic form, I pleaded my case over a total of four e-mails. The answer to the third e-mail gave me the impression that this was of interest to the NetShift troop and they may indeed help me along. Wrong impression. In the fourth e-mail they said no again, this time a little more emphatically. Most of the great inventors of our time were doled out their share of negativity, but went on to eventual fame and international fortune. I too must persevere.

**Listing 1—***Continued*

```
    On Error Resume Next
    btnshkybd.Caption = "SHOW KEYBOARD"
    kbflag = False

    RegisterKeyonMessages

    WebBrowser1.Navigate2 "http://www.circuitcellar.com"
    DoEvents
    End Sub
    Private Sub btnshkybd_Click()
    Select Case btnshkybd.Caption
        Case "SHOW KEYBOARD"
            NameStart.cb = Len(NameStart)
            RC = CreateProcessA(0&, "C:\Keyon\Keyon.exe", 0&, 0&,
 1&, NORMAL_PRIORITY_CLASS, 0&, 0&, NameStart, NameOfProc)
            RC = WaitForInputIdle(NameOfProc.hProcess, 5000&)
            RC = PostThreadMessage(NameOfProc.dwThreadId,
 WM_KEYON_TOP, 0, 320)
            RC = PostThreadMessage(NameOfProc.dwThreadId,
 WM_KEYON_LEFT, 0, 825)
            kbflag = True
        Case "HIDE KEYBOARD"
            RC = PostThreadMessage(NameOfProc.dwThreadId,
 WM_KEYON_CLOSE, 0, 0)
            RC = WaitForInputIdle(NameOfProc.hProcess, 5000&)
            kbflag = False

    End Select

        If kbflag = True Then
            btnshkybd.Caption = "HIDE KEYBOARD"
        Else
            btnshkybd.Caption = "SHOW KEYBOARD"
        End If
    End Sub
```

## A TWO-HEADED MONSTER

Part of the mission was, "If you can't get it to work, write your own code." Well, it's crunch time. I need a web browser solution that I can run multiple instances of in the Touchscreen 1 and 2 space. I need a flexible multi-instance solution, because at this point I don't know if NetShift will coexist with other applications.

Fortunately, Internet Explorer is a component of Visual Basic 6.0. Putting together a web browser is as easy as the code you see in `webbrowser1.Navigate "http://www.circuitcellar.com`. That's all there is to it. In the VB6 IDE, you simply expand the form, load the Microsoft Internet Controls component, place the browser component on the form, and

**Listing 2—***If you have VB6, break out the API Text Viewer and note the differences in the `CreateProcessA` function declarations.*

```
KEYON HIDE
KEYON SHOW
KEYON MIN
KEYON MAX
KEYON ONTOP
KEYON NOTOP
KEYON CLOSE
KEYON LEFT
KEYON TOP
```

size it to run the one line of code. And so, a web browser is born. All that's left is to put some buttons on the form to allow the user to go back, forward, up, and down.

Visual Basic's IDE also includes a virtual display to allow you to adjust your form position visually on the screen. There is a problem in that VB doesn't know about two monitors side by side. I took a chance and placed my Touchscreen 2 form to the outside and right of the virtual monitor. Sometimes even a blind hog finds an acorn.

It worked perfectly. I then fired up a NetShift session on Touchscreen 1 and my homebrew web browser on Touchscreen 2. Then I successfully browsed two different web sites on the two touchscreens.

So far, I can put the second homemade browser in Touchscreen 2 space and run the NetShift product in Touchscreen 1 space. The kiosk can talk to us, but you also need to talk back. You can't use traditional keyboards and mice in this case because you would have to place a guard and a technician at the kiosk to keep the equipment onsite and operational. You must find and employ a data entry solution that takes advantage of the touchscreen.

Fortunately, the NetShift kiosk product includes a built-in touchscreen keyboard that can be hidden when not in use. That's great for the NetShift touchscreen window, but how do I put a touch keyboard together for the homebrew web browser side of the equation?

My first thought was to roll my own in VB. After the first few key definitions, I realized I would collect retirement before I could get this to not only work, but look good.

**Photo 4**—*From the looks of this list, there's not much outside NetShift stuff that this kiosk is going to show you.*

I noticed a separate keyboard offering from NetShift in my travels on their web site. KEYON is the

touchscreen keyboard engine that is in place on the Net-Shift kiosk product. I downloaded the evaluation copy and discovered that I could customize the touchscreen keyboard using their keyboard manager. I could also control the visibility and location of the keyboard from a VB program. Yes! That "Mission Impossible" music started again, but this time Mr. Phelps was in control. The solution is to combine the VB web browser and the KEYON control code for the kiosk solution on Touchscreen 2. Listing 1 shows the code that put the kiosk solution into operation. Mission continues with impossibility diminished.

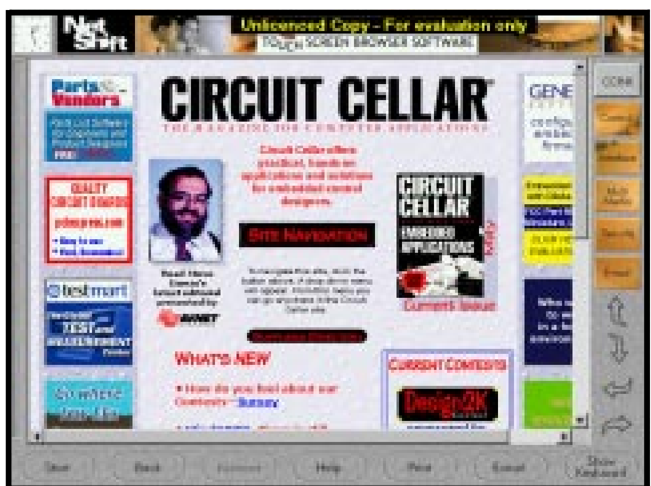



**Photo 5—***This kiosk is now aimed at one of the best sites on the web.*

With all of the software and video driver and hardware technical details worked out, you'd think that things would go smoothly. Well, not quite. I worked with the least complicated KEYON solution most of the day. I didn't want to write any complex code and assumed that the KEYON manager program would do what I needed. As it turns out, I couldn't hide the keyboard or consistently place the keyboard in Touchscreen 2 space, so I thought I'd better refer to the KEYON document.

In reading the KEYON technical documentation, I came across some Windows code specifically tailored to control the KEYON keyboard engine using VB. I would like to thank and spotlight Palmer King of Piquing Futures, Inc. in Safety Harbor, Florida. He is responsible for the KEYON VB code in the KEYON doc. Basically, Palmer has the KEYON developer copy some types, functions, and declarations from the API Text Viewer Database that comes with VB6 into a `VB.bas` module. Then the KEYON messages are defined and registered. The messages are commands for the KEYON keyboard engine. They include those shown in Listing 2.

These commands are just what we need to make our homebrew side of the kiosk look slick. All of this API activity is documented in Listing 2. I called Palmer to talk to him about this and his words were, "I had this thing working in about 15 minutes." After speaking to him and going over his code description, I had this thing working in about 30 minutes.

Why did it take me so long? Well, Palmer's declaration of the function `CreateProcessA` is correct. The same function copied from the API Text Viewer database didn't work. Looking closely at both, the differences were in the variable-type definitions. To make a long story short, VB picked the API Text Viewer function declaration code out as a compiler error and Palmer's documented code works. The fruits of my labor, KEYON and a homebrew browser, are shown in Photo 6.

## JUST ONE MORE THING

Life is good on the dual-headed kiosk front. But, this is all running on a desktop! Problem number one is that the Matrox video card is PCI-based. In fact, the two-SVGA card solution is PCI-based also. I need a PCI-based embedded hardware solution to finalize the design.

I was a little concerned until I contacted Advantech, where I found a PCI backplane, the PCA-6104P4. This

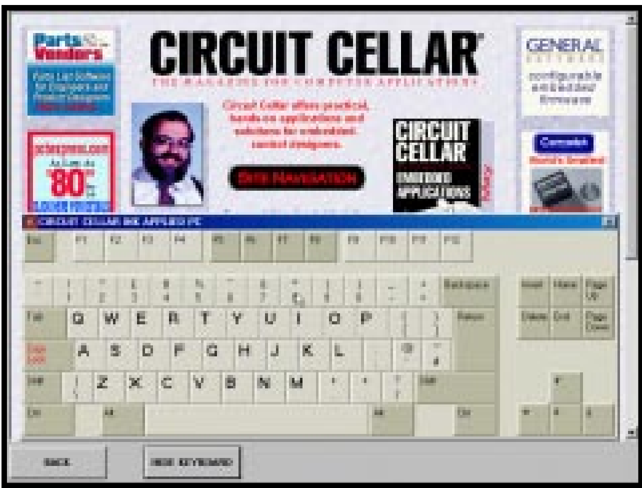



**Photo 6—***All that's left to do is add some fancy navigation buttons. Ignore the cursor you see on the "8" key. It won't be there in the final production version.*

backplane allows me to use either an ATX or AT power supply and provides me with four PCI slots. Where there's PCI backplanes, there must be PCI embedded computers. Sure enough I plugged Advantech's PCI-6771 into the Matrox G200 video card next.

Other than the fact that I need PCI capability for the video solution, the advantage of using the PCI bus standard is the higher performance of 32 bits at 33 MHz compared to the ISA buses' standard 8/16 bits at 8 MHz. And, if I need to expand beyond the resources offered by the PCI-6771, I could do so easily with off-the-shelf PCI expansion cards.

The PCI-6771 supports Socket 370 for Intel Celeron Pentium III processors, topping out with the 500-MHz part. Although it won't be used in this solution, the PCI-6771 employs a Trident Cyber 9525DVD controller that supports up to 1024 × 768 resolution. The MicroTouch-enabled touchscreens require a serial port to enable the input function. The PCI-6771 comes standard with two serial ports—one RS-232 and one RS-232/422/485. That covers the touchscreens.

Right now, the customer is in the midst of installing a nationwide intranet. So, the first kiosks may have to use DUN (Dial-up Networking). Because there are only two native serial ports, I'll temporarily plug a PCI serial card into the mix. When the intranet is installed, the PCI-6771 sports a 10/100 Mbps Ethernet interface that will be used to tie back to the home office and the serial card can be removed.

So, the next time you're in the airport or the mall and see a screen beckoning to be touched, indulge it because now you know that kiosks aren't complicated. They're embedded. ◼

*Fred Eady has more than 20 years of experience as a systems engineer. He has worked with computers and communication systems large and small, simple and complex. His forte is embedded-systems design and communications. Fred may be reached at fred@edtp.com.*

## SOURCES

**Setup Manager V4.52**
NetShift Software, Ltd.
+44 (0) 1672 511 094
+44 (0) 1672 511 078
www.netshift.com

**G200 Multi-monitor card**
Matrox Electronic Systems, Ltd.
(514) 685-7230
Fax: (514) 685-2853
www.matrox.com

**Touchscreen modules**
MicroTouch Systems, Inc.
(978) 659-9000
Fax: (978) 659-9100
www.microtouch.com

**PCA-6104P4, PCI-6771**
Advantech Co., Ltd.
(949) 789-7178
Fax: (949) 789-7179
www.advantech.com

FEATURE ARTICLE

Alan Harry

# Who Needs Hardware?

## Developing Without the Target

If you're a software designer, you know what it's like to have your hands tied while you wait for hardware to arrive. Alan steps into the Virtual Workshop to demonstrate how you can continue development and testing even without the target hardware.

When developing a product, the ability to design software without waiting for the hardware can significantly reduce the time to market. In "Developing a Virtual Hardware Device" (*Circuit Cellar* 64), Michael Smith outlines how time is spent developing software. [1] He allots 215% for "Waiting for Hardware." Depending on the project, there are times when even 215% would be an underestimation.

This article discusses how it's possible to develop and debug a complete embedded program for the 8051 microcontroller, or a variant, without target hardware. The benefits extend beyond time to market to encompass the complete software life cycle.

For example, Crossware developed a medical product that uses high-power lasers to treat skin, so FDA guidelines were followed. Development was completed in four months. The software was developed and verified almost entirely using simulation.

The software was ready before the hardware and prototypes were given to the client while testing on enhancements continued.

Consider another example. A few years ago, Crossware developed an instrument that measures the slope and height of a flowing liquid (see Photo 1). It uses three lasers and three position-sensitive detectors, which measure the displacement of the reflected beams.

The instrument is positioned above the liquid. Mounted on a frame that is not necessarily horizontal, the instrument must know its exact orientation so it can compensate for its tilt. Two ceramic tilt sensors were embedded inside the instrument, solving the problem.

The liquid's height is displayed on a graphic LCD and an image of a bubble is used to indicate the two-axis slope. The operator adjusts the screws supporting the tank that contains the liquid until the bubble is in the center of a circle printed on the display and the height offset is zero.

The software development wasn't easy. In order to move the bubble around without delay, six images were created, and the correct one was placed at the correct location by manipulating the graphics origin. The only guide was what could or could not be seen on the display.

Before testing other features, the final board was needed. And, prior to full testing, the complete mechanical unit had to be finished. By then, clients were waiting.

To provide an example for development tool users, the company recently revisited this project and determined how it would develop the software today.

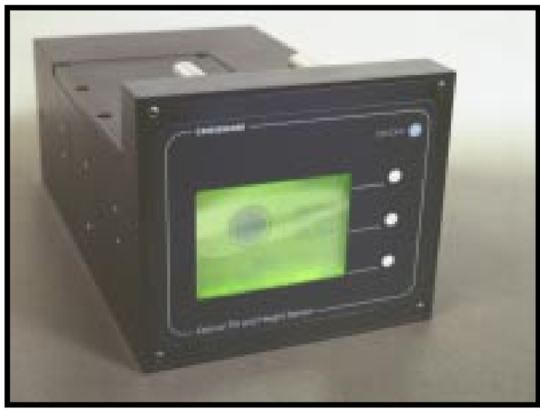


**Photo 1—***The Optical Tilt Sensor measures the slope and height of a flowing liquid's surface. Embedded tilt sensors allow the instrument to monitor its own orientation in space.*

## COMPLETE SYSTEM SIMULATION

The Crossware 8051 Virtual Workshop simulates the 8051 instruction set, timer/counters, UART, interrupts, and it can be extended. The extension interface originally was implemented to allow rapid additional support for 8051 variants.

Extensions are DLLs, which support some or all of the interface calls. An extension has a special file name so Virtual Workshop will recognize it. The interface is a set of C function calls, thus, any DLL written in C can be an extension. However, for its own extensions, Crossware creates a CExtensionState C++ object and immediately converts the C call into a `CExtensionState` function call.

By writing the DLL in a particular way and using Microsoft Visual C++ to build it, you can create an extension that integrates seamlessly with Virtual Workshop. You can add dialog boxes, windows, and menu items using the Microsoft graphical editor and Class Wizard. Virtual Workshop's `Capture State` command can be supported, allowing the target system to be captured and restored later.

Any number of extensions can be added. Virtual Workshop looks for `esim0.dll`, `esim1.dll`, `esim2.dll`, and so on, and loads when it finds them. This means an extension can be developed for a particular peripheral. The extension may be used again if the same peripheral is being used in a different target system.

Virtual Workshop sends calls to each extension, which handles the ones it chooses. And, extensions can communicate with each other (named pipes make it easy to demonstrate the communication between extensions).

**Listing 1—***Virtual Workshop detects a falling edge on a port pin and generates an interrupt if appropriate.*

```
#define P2 0XA0
#define P3 0XB0

#define KEY1PIN 0X02 // P2.1
#define KEY2PIN 0X04 // P2.2
#define KEY3PIN 0X08 // P2.3

void CExtensionState::GetPortPins(BYTE nPortAddress, BYTE* pnPins, BYTE*
pnHandledPins, BOOL bSimulating)
{
  switch (nPortAddress)
  {
    case P2:
      if (m_pKeys->IsKey1Pressed()) // interrogate the dialog box
      {
        *pnPins &= ~KEY1PIN;            // clear pin
        *pnHandledPins |= KEY1PIN;      // pin handled
      }
      else
      {
        *pnPins |= KEY1PIN;             // set pin
        *pnHandledPins |= KEY1PIN;      // pin handled
      }
      if (m_pKeys->IsKey2Pressed())     // interrogate the dialog box
      {
       // clear appropriate pin and trigger EX0 with a falling edge
        *pnPins &= ~KEY2PIN;            // clear pin
        *pnHandledPins |= KEY2PIN;      // pin handled
      }
      else
      {
        *pnPins |= KEY2PIN;             // set pin
        *pnHandledPins |= KEY2PIN;      // pin handled
      }
      if (m_pKeys->IsKey3Pressed())     // interrogate the dialog box
      {
        *pnPins &= ~KEY3PIN;            // clear pin
        *pnHandledPins |= KEY3PIN;      // pin handled
      }
      else
      {
        *pnPins |= KEY3PIN;             // set pin
        *pnHandledPins |= KEY3PIN;      // pin handled
      }
      break;
    case P3:
       // trigger an interrupt if any key is pressed
       if (m_pKeys->IsKey1Pressed() || m_pKeys->IsKey2Pressed() || m_pKeys-
>IsKey3Pressed())
     {
        // P3.2 goes low for external interrupt 0
        *pnPins &= ~0X04;               // clear P3.2
        *pnHandledPins |= 0X04;         // P3.2 handled
      }
      else
      {
        // P3.2 high
        *pnPins |= 0X04;                // set P3.2
        *pnHandledPins |= 0X04;         // P3.2 handled
      }
      break;
  }
}
```

## VIRTUAL SPIRIT LEVEL

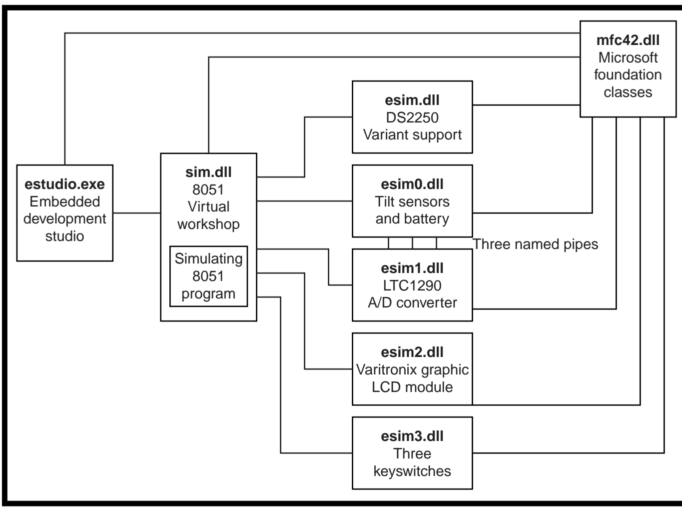
Let's discuss the simulation of a complete sub-set of Crossware's measurement instrument. Lasers and the position-sensitive detectors are ignored in this example. The display, A/D converter, tilt sensors, battery, and keyswitches are simulated to create a virtual electronic spirit level.

When finished, the component programs work together as a set, as shown in Figure 1. I'll briefly describe the components before covering the target board-specific extensions.

The Embedded Development Studio (`estudio.exe`) is the environment that provides project management and editing facilities, and allows you to compile and assemble your source code. When you select 8051 as the target microcontroller, it loads and initializes the 8051 Virtual Workshop (`sim.dll`).

A lot of interaction occurs between the Embedded Development Studio and Virtual Workshop. The former informs the latter about the target program and source level breakpoints. Virtual Work-

**Figure 1**—*When the virtual electronic spirit level is running, the four custom DLLs (esim0.dll, esim1.dll, esim2.dll, and esim3.dll) will integrate with the rest of the Crossware development environment.*




shop places additional windows, menus, and toolbars in the Embedded Development Studio environment.

When the 8051 program is ready to run, the user selects an appropriate command, such as `Go` or `StepInto`, and Virtual Workshop starts its functions. First, Virtual Workshop asks the Embedded Development Studio where it should look for extensions and then loads and initializes them. After that, Virtual Workshop loads the 8051 program and begins simulating it instruction by instruction. At this point, all DLLs are running and receiving calls from Virtual Workshop.

Extension `esim.dll` provides support for extra features presented by Dallas Semiconductor's DS2250. Features include a watchdog timer, additional interrupts, banked RAM, and such. The extras are part of the Virtual Workshop package and are auto-

**Listing 2—***Here's the 8051 program interrupt function that reads the keyswitches. The developer can set breakpoints and single step through interrupt routines.*

```
void _interrupt IVN_INTERRUPT0 _using 1 KeyPress()
{
   unsigned char i;
   unsigned char nKeyPressed;
   unsigned char nKeyMask;
   unsigned char nKeyMaskCheck;
   _ie0 = 0;              // clear the interrupt flag
   nKeyMask = _p2;        // read port 2
   ResetWatchDog();
   for (i = 0; i < 10; i++);  // delay
                          // read port 2 a second time to debounce
                            the keys
    nKeyMaskCheck = _p2;
   if (nKeyMask == nKeyMaskCheck)
   {
                          // The same data was read both times,
                          // so assume valid keypress
      switch (nKeyMask)
      {
         case 253:        // 11111101
            KeyOneResponse();
            break;
         case 251:        // 11111011
            KeyTwoResponse();
            break;
         case 247:        // 11110111
            KeyThreeResponse();
            break;
      }
   }
}
```

**Listing 3—***With the graphic LCD extension, you simply write to an external address at or greater than 8000 hex to access the display data bus. The HandleCommand routine does the work.*

```
BOOL CExtensionState::SetXDataMemory(int nAddress, BYTE nValue, BOOL
                         bSimulating)
{
   if (m_bChipEnabled && nAddress >= 0X8000)
   {
      if (m_bCommandMode)
      {
                       // program is writing a command byte
         m_nCommand = nValue;
         HandleCommand();
      }
      else
      {
                       // program is writing a data byte
         if (m_bDataAutoWrite)
         {
                       // place the byte in memory and increment the memory
                         pointer
            m_Memory[m_nAddressPointer++] = nValue;
            if (g_pDisplayDlg)
            {
                       // show the updated memory pointer in the dialog box
               g_pDisplayDlg->SetAddressPointer(m_nAddressPointer);
            }
         }
         else
         {
                       // keep track of the 1st two data bytes for use
                       // by the next command
            m_nData[1] = m_nData[0];
            m_nData[0] = nValue;
         }
      }
      m_nBusy = 4;  // time 4 micro-second busy period
                       // tell the Virtual Workshop that this extension has
                       // handled SetXDataMemory by returning TRUE
      return TRUE;
   }
   return FALSE;
}
```

**Listing 4**—*The data bytes have already been received when the command byte is written. They are stored and can be used if the command needs them. This is a partial listing.*

```
void CExtensionState::HandleCommand()
{
   CString strCommand;
   switch (m_nCommand & 0XF0)
   {
      case CONTROL_WORD_SET:
         switch (m_nCommand & 0X0F)
         {
            case TEXT_HOME_ADDRESS_SET:
               strCommand = "Text home address set";
               m_nTextHomeAddress = m_nData[0] << 8 | m_nData[1];
               if (g_pDisplayDlg)
                  g_pDisplayDlg->SetTextHomeAddress(m_nTextHomeAddress);
               break;
            case TEXT_AREA_SET:
               strCommand = "Text area set";
               m_nTextArea = m_nData[0] << 8 | m_nData[1];
               if (g_pDisplayDlg)
                  g_pDisplayDlg->SetTextArea(m_nTextArea);
               break;
            case GRAPHICS_HOME_ADDRESS_SET:

      .....................
      .....................
      .....................

      default:
```

**(continued)**

matically selected when you choose the DS2250 variant in the Embedded Development Studio.

The four custom extensions, `esim0.dll`, `esim1.dll`, `esim2.dll`, and `esim3.dll`, will be developed specifically for this target system.

Finally, `mfc42.dll` contains the Microsoft Foundation Classes—a comprehensive C++ interface to Microsoft Windows. Note that this DLL is used by all components. Without this link, the components couldn't integrate seamlessly.

When developing a Virtual Workshop extension, you may run it in the Microsoft debugging environment. You'll be asked what `.exe` program your DLL is associated with when you first run from the Start Debug menu. Specify `estudio.exe` and the Embedded Development Studio environment will fire up. You can set breakpoints in your DLL, so you can single step through and observe its behavior.

## FOUR CUSTOM EXTENSIONS

Next, let's delve into the custom extensions, starting with the simplest, `esim3.dll`, and finishing with `esim0.dll`.

Virtual Workshop comes with an AppWizard. This program interacts with the Microsoft environment, so when you create a new Microsoft C++ project, you may create a Crossware 8051 Virtual Workshop extension. When you do, skeletal source code for a complete extension will be created, ready for you to customize and build.

To customize `esim3.dll`, create a modeless dialog box containing three buttons (see Photo 2). This is done using the Microsoft graphical tools, with the outline code and variables generated by Microsoft Class Wizard. You can make it a modeless dialog box by adding the `Create` call to the class constructor (just make sure it's visible). The process takes about 10 minutes.

Then, modify the CExtensionState class by adding code to create the dialog box object and to interrogate its buttons whenever `CExtensionState::GetPortPins` is called.

It takes about 30 minutes to create a set of virtual keyswitches. You may want to add extra cosmetic features.

**Listing 4**—*continued*

```
            strCommand = "Invalid command";
            break;
        }
        if (g_pDisplayDlg)
            g_pDisplayDlg->ShowCommand(strCommand);
    }
```

**Listing 5**—*The data bytes have already been received when the command byte is written. They are stored and can be used if the command needs them.*

```
    // the routine in esim0.dll that is running three times
    // in three separate threads

    CCriticalSection g_CriticalSection;

    UINT PipeRoutine(void* pParam)
    {
        const char* pszName = (const char*)pParam;
        CString strPipeName;
        strPipeName.Format("\\\\.\\pipe\\%s", pszName);
        HANDLE hPipe = CreateNamedPipe(strPipeName,
            PIPE_ACCESS_DUPLEX,                    // dwOpenMode
            PIPE_TYPE_MESSAGE |
            PIPE_READMODE_MESSAGE |
            PIPE_WAIT,
            PIPE_UNLIMITED_INSTANCES,
            BUFSIZE,
            BUFSIZE,
            PIPE_TIMEOUT,
            NULL);
        if (hPipe == INVALID_HANDLE_VALUE)
        {
            CString strMessage;
            strMessage.Format("Could not create pipe %s", strPipeName);
            AfxMessageBox(strMessage);
            return 0;
        }
        BOOL bConnected = ConnectNamedPipe(hPipe, NULL) ? TRUE :
(GetLastError() == ERROR_PIPE_CONNECTED);
        if (!bConnected)
        {
                                                   // exit thread
            CString strMessage;
            strMessage.Format("Could not connect to pipe %s", strPipeName);
            AfxMessageBox(strMessage);
            return 0;
        }
        CSingleLock AccessToExtensionState(&g_CriticalSection);
        while (nExtensionCount > 0)
        {
            char chRequest[BUFSIZE];
            char chReply[BUFSIZE];
            DWORD nBytesRead, nReplyBytes, nWritten;
            BOOL bSuccess = ReadFile(hPipe, chRequest, BUFSIZE, &nBytesRead,
NULL);
            if (!bSuccess || nBytesRead == 0)
                break;
            AccessToExtensionState.Lock();
            g_pExtensionState->GetAnswerToRequest(chRequest, chReply,
                &nReplyBytes, pszName);
            AccessToExtensionState.Unlock();

            bSuccess = WriteFile(hPipe, chReply, nReplyBytes, &nWritten, NULL);
            if (!bSuccess || nReplyBytes != nWritten)
                break;

        }
        FlushFileBuffers(hPipe);
        DisconnectNamedPipe(hPipe);
        CloseHandle(hPipe);
        return 0;
    }
```

The code that you need to add GetPortPins depends on the electronic circuit exhibited in Figure 2. Figure 2 shows each keyswitch connected to its own microcontroller port pin, and to INT0. That means that pressing a keyswitch causes an interrupt, and the 8051 program can interrogate Port 2 to determine which keyswitch caused it.

**Photo 3—***The LCD and display driver attributes allow the developer to easily see if the embedded program is functioning as expected.*

GetPortPins is called repeatedly after each instruction is simulated. This allows Virtual Workshop to be sensitive to falling-edge, rising-edge, and level-sensitive external interrupts. The extension only needs to apply the correct levels to the pins it controls. It does this by setting or clearing appropriate *pnPins bits and informing Virtual Workshop that it has set or cleared a particular bit by setting the corresponding bit, *pnHandledPins.

Listing 1 shows the complete GetPortPins function and Listing 2 shows the 8051 interrupt function code that reads the keys.

## GRAPHIC LCD MODULE

The circuit diagram shows that the data bus is connected to the microcontroller's Port 0, and the display's and microcontroller's /WR and /RD pins are connected. The microcontroller writes to or reads from the display when it writes to or reads from off-chip external data memory.

For the DS2250, this occurs when the xdata address is greater than 8000 hex. So, use CExtensionState::SetXDataMemory to determine if the display is being written to, and CExtensionState::GetXDataMemory to determine when it's being read.

The display's chip enable is connected to P2.5. You need to use CExtensionState::SetPortPins to keep track of this pin. Then, ignore all reads and writes unless the display is enabled. Similarly, you need to monitor P2.6 to determine whether the display is in command or data mode. Listing 3 shows code for SetXDataMemory. HandleCommand interprets the display driver commands and is shown in Listing 4.

Variable g_pDisplayDlg points to a dialog box (see Photo 3) that displays the LCD and its attributes. As with the keyswitches, this dialog box and associated code can be created quickly using Microsoft's graphics editor and Class Wizard.

However, because you're displaying graphics in the dialog box, you have to create a CWnd object and subclass it to a placeholder in the dialog box. The CWnd object then receives the Windows messages that the placeholder would have received, and it can draw an image of the display in the placeholder's window area. In order to accomplish sub-classing, you may use a simple, single call to the MFC function SubclassDlgItem.

**Photo 2—***The three keyswitches for esim3.dll will be positioned immediately to the right of the display. Legends on the display change as the functions of the keyswitches change.*

Keyswitches
Key 1
Key 2
Key 3

Photo 3 also shows the attributes and the display driver chip's state. This makes development of the 8051 program easier because you can see if it works as expected.

Also, you must account for timing. The display driver requires 4 µs to process a byte. Crossware's 8051 program polls the display's status byte to determine when the display is ready to receive another byte, so the simulation needs to incorporate a busy flag.

CExtensionState::IncMachineCycles times the busy flag. After each in-

**Listing 5—***continued*

```
    // CExtensionState constructor in esim0.dll
    // tilt sensors and battery extension

    CExtensionState::CExtensionState()
    {
       ....
       ....
       ....
       g_pExtensionState = this;                    // let threads access
class functions
       AfxBeginThread(PipeRoutine, (void*)"TiltSensor1");
       AfxBeginThread(PipeRoutine, (void*)"TiltSensor2");
       AfxBeginThread(PipeRoutine, (void*)"Battery");
       ....
       ....
    }

    // CExtensionState constructor in esim1.dll
    // A/D Converter extension

    CExtensionState::CExtensionState()
    {
       ....
       ....
       ....
       for (int i = 0; i < NO_OF_CHANNELS; i++)
       {
          m_hPipe[i] = INVALID_HANDLE_VALUE;
       }
       m_hPipe[5] = OpenNamedPipe("Battery");
       m_hPipe[6] = OpenNamedPipe("TiltSensor1");
       m_hPipe[7] = OpenNamedPipe("TiltSensor2");
    }
```

**Photo 4—***The simulating A/D converter can fetch its inputs from the dialog box edit fields or from named pipes running anywhere else on the system.*

struction is simulated, `IncMachineCycles` is called with an argument that contains the number of machine cycles elapsed since the last call. Thus, cycle-accurate features can be implemented in the extension.

## THE A/D CONVERTER

The `esim1.dll` extension simulates the A/D converter, which is driven by the microcontroller's Port 1. This device's extension uses `CExtensionState::SetPortPins` to monitor the port's output and `CExtensionState::GetPortPins` to send data back.

To help the simulation, Crossware constructed a UML-style state chart that depicts the operation. The chart is based on the description in the manufacturer's datasheet (see Figure 3). The code in SetPortPins and GetPortPins is based on the state chart. If Crossware had constructed this chart a few years ago when developing the original instrument, it would have helped the original 8051 program development, too.

Photo 4 shows the dialog box for the A/D converter. This displays the device's attributes and allows developers to directly enter data representing the voltage level on its inputs.

## NAMED PIPES

However, Crossware wanted the A/D converter simulation inputs to come from the tilt sensors and battery. In order to facilitate reuse, these are split into a separate extension. To communicate between the two extensions, named pipes are used. Extension `esim0.dll` creates three named pipes, TiltSensor1, TiltSensor2, and Battery. `esim1.dll` connects to these and requests data from the appropriate one when it needs an input level.

Named pipes work system-wide, and the operating system handles the details. To the program, they behave like files. To service a pipe, `esim0.dll` must create a separate thread. It then loops continuously and waits for a return from the `ReadFile` function. This function returns in response to a data request, so the thread gathers the data and sends it to the requestor using `WriteFile`.

Extension `esim0.dll` creates three threads, and there are three separate channels of communication between `esim0.dll` and `esim1.dll` (see Listing 5). Then, the tilt sensors and battery charge can be controlled separately using the dialog box in Photo 5.

Battery drain is simulated using the `IncMachineCycles` function, which decrements a variable representing the charge state at a rate that corresponds with the simulation rate. Similarly, charging is simulated by incrementing the same variable whenever the charge box is checked.

**Figure 2—***The electronic spirit level is a sub-set of the Optical Tilt Sensor. It uses two ceramic tilt sensors to detect orientation in two axes.*

**Photo 5—***Three separate threads, one for each sensor and one for the battery, service named pipes so that the A/D converter can retrieve suitably scaled values from this dialog box.*

The MFC function `AfxBeginThread` starts new threads. This is easy, but be careful. Any action on a window handle, other than using it to post a message, is likely to fail. So, the dialog box is programmed to independently keep variables `m_nTiltSensor1`, `m_nTiltSensor2`, and `m_nBattery` up-to-date and uses a `CCriticalSection` to ensure that the variables are not accessed simultaneously from separate threads.

## DEBUGGING WITHOUT HARDWARE

With the extensions in place, you now can run a complete simulation of the target system and develop the 8051 program. Features may be added to the extensions to trap error conditions or to automate testing. Try it yourself by downloading the package from the *Circuit Cellar* web site.

You'll notice that not everything was simulated. In particular, there isn't support for many display driver features. The objectives are to speed development and support the verification and life cycle processes. Spending time providing features that won't be used doesn't support the objectives.

It took three days to develop the extensions, with the display extension requiring the most time. The benefits of being able to see the internal attributes of the display driver and A/D converter, to test a wide range of tilt sensor inputs, and to automate the testing process at any time and independently makes it worthwhile—even when the hardware is available.

How about e-mailing your specification and Virtual Workshop extensions to an inexpensive resource on the other side of the world, while you're at the beach? Developing your embedded program is easier now, so you'll see enough of the beach while waiting for the hardware to catch up. ■

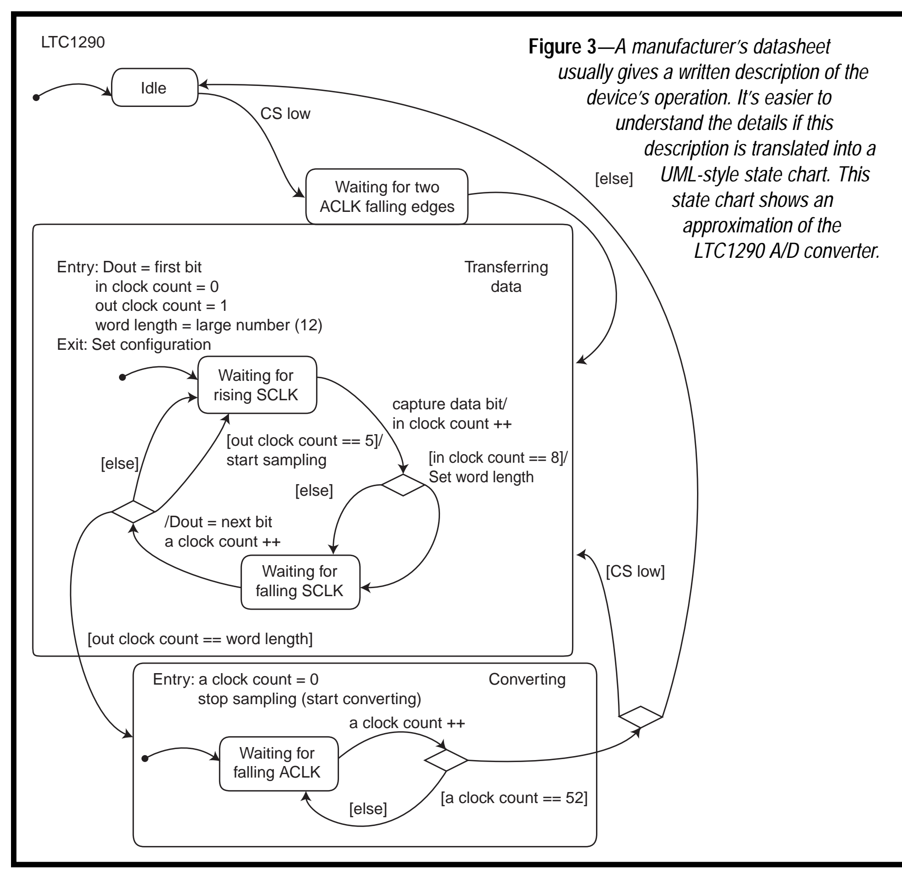



**Figure 3—***A manufacturer's datasheet usually gives a written description of the device's operation. It's easier to understand the details if this description is translated into a UML-style state chart. This state chart shows an approximation of the LTC1290 A/D converter.*


*Alan Harry is the founder and managing director of Crossware, a developer of C cross compilers and other development tools for embedded systems based on the 8051, ColdFire, 68000, CPU32, and other chip families. He also heads a multi-disciplinary product consultancy that works on developments for international clients. You may reach Alan at alan_harry@crossware.com.*


## REFERENCES


[1] M. Smith, "Developing a Virtual Hardware Device", *Circuit Cellar*, 64, November, 1995.

[2] B.P. Douglass, *Real-Time UML: Developing Efficient Objects for Embedded Systems*, Addison-Wesley, 1998.


## SOURCES

**8051 Virtual Workshop**
Crossware Products
011 44 1763 853500
Fax: 011 44 1763 853330
www.crossware.com

**MGLS12864T-LV2**
VL Electronics Inc.
(213) 738-8700
Fax: (213) 738-5340
www.vle.com

**T6963C**
Toshiba America Electronic Components, Inc.
(770) 931-3363
Fax: (770) 931-7602
www.toshiba.com

**LTC1290**
Linear Technology Corporation
(408) 432-1900
Fax: (408) 434-0507
www.linear-tech.com

**SH50055**
Spectron Glass and Electronics Inc.
(516) 582-5600
Fax: (516) 582-5671
www.spectronsensors.com

**T2250**
Dallas Semiconductor
(972) 371-4167
Fax: (972) 371-3715
www.dalsemi.com

FEATURE ARTICLE

Tom Napier

# Count the Digits

## Designing a Frequency Meter

Sure, you could just buy a frequency meter, but if you're like Tom, you probably have all the necessary parts sitting around, so why not build your own? Before this project is done, you'll have a better understanding of frequency meters.

It's sensible to buy an accurate frequency meter, but a decent one costs more than $500. You can get similar performance from a home-built product that costs about $70 to build. I had most of the parts lying around, so I decided to design and build my own six-digit autoranging frequency meter. The leftover parts slightly hindered the design, so learn from my experience.

I liked the fact that this project involved tradeoffs among analog, digital, and firmware designs. It also involved tricky mechanical design. It was like a commercial design project without the marketing department leaning over my shoulder. And, I didn't have to worry about the manufacturing cost.

### WHAT IT DOES

A frequency meter is a pulse counter that turns on for an accurately known time and displays the accumulated count. Dividing the output of a crystal clock sets the on time. The end count is a linear function of the input frequency.

Twenty-five years ago, you would have strung together a crystal, TTL decade divider chips, display drivers, and as many seven-segment display chips as needed. The only analog part of the circuit was the high-speed comparator, which turned the input signal into appropriate TTL level pulses. It and the crystal oscillator were designed from discrete transistors and resistors, but the rest of the circuit was made from standard TTL building blocks.

Today, the comparator and the oscillator are standard blocks and the dividers and display drive are firmware functions. I used a PIC16C55 microcontroller to count the input pulses and drive the display. Normally, you would use an off-the-shelf LCD unit. I had old seven-segment display chips I wanted to use, so I compromised. Hence, my display has 0.3″ LED digits.

### HOW MANY DIGITS?

Two primary design decisions concern accuracy and resolution. Resolution refers to how many different results you can display. I used six digits, giving a resolution of one part per million (ppm). You can add more, but each extra digit increases the counting time by a factor of 10. Resolution is cheap, but it means nothing without accuracy.

Accuracy refers to how well the result compares to a standard. In a frequency meter, the accuracy is a function of the crystal oscillator. You can buy new crystal oscillators for $3 or surplus ones for $1. But, how accurate are these? The standard specification is ±100 ppm.

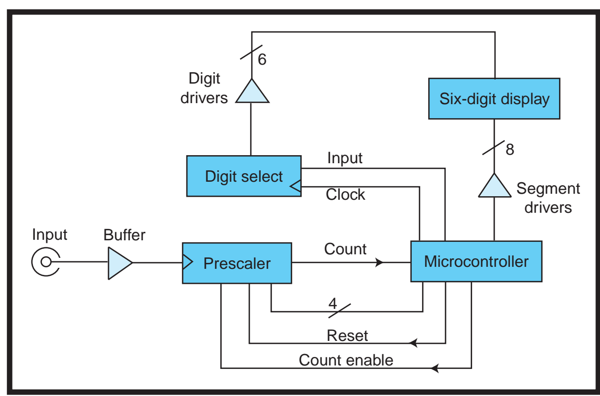



**Figure 1**—*The frequency meter uses a microcontroller as its counting and display device. A prescaler extends its maximum input frequency to 50 MHz.*

Does that mean that measuring frequency with six-digit resolution is a waste of time? The answer is no, for a couple of reasons. One reason is that six digits give a best resolution of 1 ppm, but only 5- to 10-ppm resolution when the first digit of the frequency is one. Another reason is that even if the absolute accuracy of a measurement is low, it can measure the difference between two frequencies with a high resolution.

However, a ±100 ppm oscillator can be accurate. Manufacturers state that over a wide temperature range (0° to 50° C), the frequency generated is within ±100 ppm of the frequency marked on the crystal. This error band allows room for how accurately the maker tuned the crystal to the correct frequency and how the frequency varies with temperature.

In most situations, the temperature variation is a parabola or an S-shape. This means that at room temperature, the actual frequency is closest to the nominal value and the variation per degree is less than around the extremes of the range. If you don't plan to use the frequency meter on cold days, you should have no problems and the temperature stability will work well.

The absolute accuracy of the crystal should be better than ±25 ppm. The number of digits stamped on the oscillator is a good guide. An oscillator marked 10.000000 MHz is probably more accurate than one marked 10.000 MHz, although it is unlikely to have the 0.1-ppm accuracy implied by the label. An oscillator claiming ±50-ppm accuracy is worth buying, but a more accurate crystal isn't worth the extra money.

## HOW IT WORKS

The frequency meter counts input cycles for a fixed time period, usually a decimal fraction of one second, then displays the result. Adjust the time period to maximize the resolution so the resulting count is between 100,000 and 999,999. Then move the decimal to give a result in megahertz or kilohertz.

The frequency resolution is the reciprocal of the counting period. For example, when displaying frequencies greater than 10 MHz, the counting period is 1/100 of a second and the resolution is 100 Hz. At the opposite end of the scale, the counting period is 10 s, the resolution is 0.1 Hz, and frequencies up to 100 kHz can be displayed.

Most of the counting is done in the PIC's on-chip registers. What input rate can the PIC handle via its timer input pin, RTCC? Because you need to count every input pulse, you can't use the on-chip prescaler. Although this would let you handle higher input frequencies with the bare chip, you would lose resolution because there is no mechanism for reading the residual count in the prescaler. This count represents the bottom two decimal digits of the count.

Given a 50% on/off ratio in the input, the timer pin can handle periods of 40 ns greater than the instruction period. With a 20-MHz clock, the instruction period is 200 ns, so you can count frequencies up to 4.17 MHz. Because I wanted to measure frequencies greater than 30 MHz, this wasn't fast enough.

## PUSHING THE ENVELOPE

The answer is to add a four-bit high-speed prescaler chip. This allows the input frequency to be 16 times greater. You don't lose resolution because the external prescaler's terminal count can be read after the count period. The theoretical maximum counting rate is now more than 66 MHz. By limiting the frequency

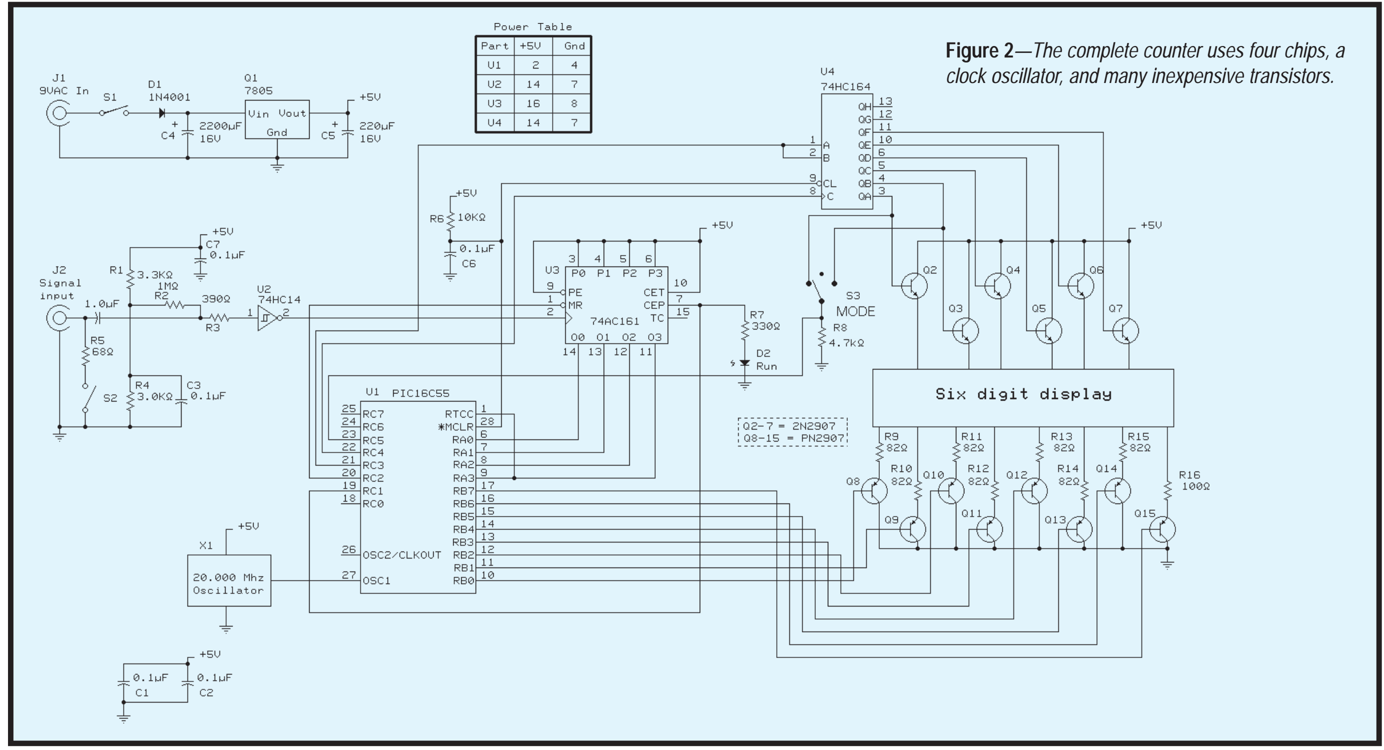



**Figure 2**—*The complete counter uses four chips, a clock oscillator, and many inexpensive transistors.*

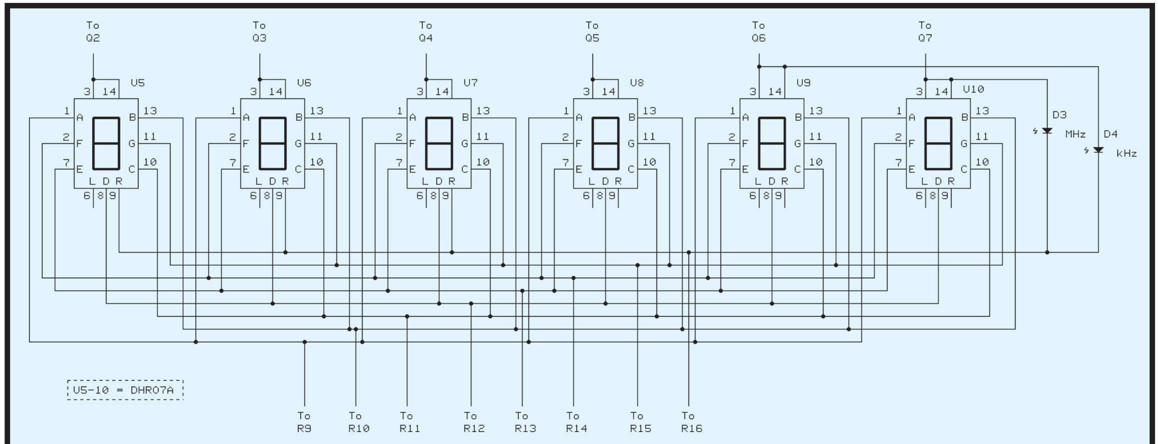



**Figure 3**—*The six seven-segment display digits are wired with the segment cathodes in parallel. The drive to the digit anodes is time multiplexed. Only three decimal points are connected.*

meter's specified range to 50 MHz, you gain a safety margin.

The 74AC161 prescaler chip has a count enable input that is switched on and off by the PIC to start and stop counting. The prescaler is reset before each count starts.

One advantage of using a prescaler is that it reduces the effect of an ill-formed or irregular input signal. The 74AC161 handles pulses 2-ns long, arriving less than 10-ns apart, and passes a 3-MHz squarewave to the PIC. Hence, this counter not only measures sinewave inputs, but also the mean arrival rate of pulses that occur in bursts. The rate is sometimes referred to as the "sequency" to distinguish it from a regularly occurring frequency.

The PIC synchronizes its timer pin input to its internal clock and accumulates a pulse count in its 8-bit timer register. So, you can read the timer register without fearing that it will change as you read it. By sampling more than once every 80 µs, you can detect when it overflows and add one to another 8-bit register. The total capacity of this register, the timer register, and the prescaler is 20 bits, or 1,048,576. That's sufficient to store a six-digit full-scale count.

## FIRMWARE CONSIDERATIONS

Because the on-chip timer counts the input, it can't keep track of real time. So, execute a fixed number of instructions during sampling periods. The timing period must be an integral number of instruction times, and the clock frequency must be chosen accordingly. I used a 20-MHz clock.

To update the display, the six LED digits are time multiplexed for 2 ms each and are continuously refreshed. An 8-bit PIC port drives the segments of the six digits in parallel.

Ideally, you would display the input count as it accumulates, but this isn't practical because you can't easily read the prescaler on the fly. Also, you are counting in pure binary and don't want to do binary-to-decimal conversions continually, a 20-bit conversion takes 300 µs.

To help, I converted a bug into a feature. The firmware is either in a count or a display loop. It does the binary-to-decimal conversion between the two. In the count loop, it tests for timer overflow every 50 µs and keeps accurate track of real time. In the display loop, it displays a static result. It tracks passing time to determine when to re-initiate count mode.

However, its time granularity is 2 ms (the digit multiplexing period), and there's no need for accurate timing. Shutting down the display loop during counting simplifies the code. The segment drive is turned off, and the display goes blank. A front panel LED indicates that a count is in progress. The whole job was completed with less than 256 instructions.

Getting the timing correct is tricky. For example, the timer register is sampled every 50 µs (250 instructions), but you can't execute a 1-µs loop 50 times. It takes eight instructions to compare the timer against its last value and to increment the next register if there is an overflow. This means that the loop count has to be only 47.

The next loop counts 200 of these 50-µs loops. It is executed 1, 10, 100, or 1000 times. Whenever a carry occurs in the counter register, the extra code is padded to an even number of microseconds, and the count for the next 50-µs loop is decremented accordingly. The first 50-µs loop has to be shorter than the rest to leave room for the code that turns off the counter chip when the count is done.

## DRIVING THE DISPLAY

The PIC port can't handle the peak segment current (~25 mA per pin) so I buffered the pins with eight TO-92 PNP transistors connected as emitter followers. A commercial product would use a driver chip.

The common anodes of the display chips are driven in sequence. The drive current, if all seven segments and the decimal point are lit, is approximately 200 mA, which is beyond the capacity of a PIC port pin. Again, I used emitter followers. Small NPN switching transistors can handle more than 500 mA. Their mean power dissipation is low. I used generic transistors from Radio Shack.

There weren't enough spare port pins to drive the digit transistors separately. Normally, the alternative is a decoder chip, but it is has an active-low output. The 74HCT259 can act as an active-high decoder, but I found a solution that only required two PIC pins.

I hooked up a 74HCT164 shift register. If one PIC pin drives its clock and a second pin drives its data input, it's easy to insert a one-bit to step through six outputs to drive the six digits.

The shift register must be cleared when power is turned on to avoid activating several digits simultaneously. To display a count, the shift register is clocked at 500 Hz. A new one-bit is inserted every six clocks, and the appropriate segment pattern for each digit is multiplexed out the 8-bit port. The decimal points and the range indicators are treated as the eighth segment.

| | |
|---|---|
| Display: | Six-digits |
| Frequency ranges: | 0–50 MHz (sampling time = 10 ms) |
| | 0–10 MHz (sampling time = 100 ms) |
| | 0–1 MHz (sampling time = 1s) |
| | 0–100 kHz (sampling time = 10 s) |
| Input: | 68 Ω or 1 MΩ > 1.5 $V_{p-p}$ |
| Mode 1: | Continuous update, 3-s display between sampling periods |
| Mode 2: | Take one sample and hold |
| Frequency reference: | 20-MHz crystal, 50 ppm |

**Table 1—***This frequency meter has specifications comparable to a general-purpose commercial instrument.*

To minimize front panel controls, the meter is autoranging. It takes a series of measurements of each input, starting from the 10-ms test period. If the most significant digit of the resulting count is a zero, the firmware switches to the next longer period. If an overflow is detected during the count or during the binary-to-decimal conversion, the autoranger switches to the next shorter count period.

The decimal point position is set according to the range and two LEDs flag MHz or kHz. Because only three decimal point positions were needed, I wired the range indicator LEDs as if they were two more decimal points. This saved using discrete driver pins.

The mode control switch has auto, hold, and start positions. When it is in auto position, the meter cycles continuously. It counts for as long as necessary, then displays the result for 3 s.

If you need longer than 3 s to note a reading, you can switch to hold. If you switch to start, the meter will execute one count and display the result indefinitely. You have to switch to hold, then back to start to take another reading. If you purchase a commercial design, this function would be accomplished with a spring-loaded switch.

One PIC pin is needed to read the switch and up to six exclusive switch positions can be read. The same shift register that drives the display drives the two outer switch pins, and the common pin goes to a PIC input pin. Hence, as it multiplexes the display, the firmware also checks for switch closures. The switch cannot be read during a count.

## ANALOG

The signal input is a BNC connector. A 68-Ω resistor can be switched across it as a compromise between 75-Ω and 50-Ω cable termination. The input is AC-coupled.

Originally, I planned to make the input impedance 1 MΩ to be compatible with a standard 10× scope probe. A fast comparator would have given an input sensitivity of 50 $mV_{p-p}$. This plan collapsed when I noticed that the

comparator had a specified input bias current of 16 µA.

My back-up plan was to substitute a 74AC14 Schmitt trigger chip for the comparator. This has a high input impedance but requires a 1.5-$V_{p-p}$ input to generate an output. So much for using a scope probe.

## IN A BOX...

I spent more time on the mechanical design than on any other aspect of the meter. A six-digit display needs a wide front panel but no great height, so I used a 1.5″ × 5″ plastic box from Radio Shack. The box has grooves for a front panel and a vertical circuit board half an inch behind the panel.

If I had mounted the display chips on a board in the rear grooves, they wouldn't have reached the front panel, even if I put them in sockets. So, I cut a piece of prototyping board and mounted it on spacers above a support board that fits into the rear grooves.

I soldered a strip of 16 wire-wrap pins to the display board. These pass through the supporting board and mate with a 25-pin single-in-line connector mounted edge-on to the main circuit board. The latter is horizontal and is screwed to the box's molded stand-offs.

The input socket and switches are glued to the support board and poke through holes in the front panel. The Schmitt trigger chip is mounted on the back of the support board. Electrical connections from the support board are made by more wire-wrap pins, which also line up with the 25-pin connector on the circuit board.

I had a transparent red filter panel from another box that was large enough to cover the display chips. It is standard procedure to glue it to a cutout in the front panel, but I used an old optical designer's tactic: if possible, eliminate air spaces between surfaces.

I spread a thin layer of clear silicone rubber over the front of the display chips after they were mounted in the board, before they were soldered. Then, I pressed the red filter on top of the display and squeezed out the air bubbles. I turned the assembly face down on a smooth surface and pressed down on each chip to minimize the thickness of the rubber. If you cover the filter with clear tape during assembly and testing, you will be able to read the display without scratching the filter.

The silicone rubber couples the light from the display into the plastic sheet without reflections, making a brighter, clearer display. There is a tiny, invisible gap between the red plastic and the front panel. Replacing a bad display chip won't be easy!

Despite my careful mechanical design, I drilled the front panel holes 0.1″ too far up and had to redo it. That's why the front panel looks fine, but the back panel has odd ventilation holes in it. Also, I forgot an on/off switch. Fortunately, there was room on the front panel for a slide switch, but it looks awkward.

## POWER TRIP

The unit requires 5 V at approximately 200 mA. A 9-VDC adapter and a three-terminal regulator on a heat sink provide the power. For compatibility with other instruments, I used a 9-VAC transformer and added a rectifier and storage capacitor to the box.

If you're wondering why I didn't use pin 0 of Port C of the PIC, it's because I once inserted my 16C55 backward in a socket. One bond wire blew, but everything else works. I have looked for a good home for that chip ever since.

*Tom Napier is a physicist and engineer who parlayed his design experience into an electronics consulting business. His compulsion to share his knowledge drives him to write magazine articles, but he regrets that he cannot offer individual design assistance. He will start using e-mail once the bugs have been worked out.*

## SOFTWARE

The project firmware is available on the *Circuit Cellar* web site.

## SOURCE

**PIC 16C55 microcontroller**
Microchip Technology, Inc.
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

FROM THE BENCH

**Jeff Bachiochi**

# Building on Familiar Ground

## Part 1: Adding Analog to the 8051 Core

Jeff's been an Analog Devices fan for years. After scouting AD's MicroConverter, he's sure that this addition to the lineup will improve their position in the microcontroller standings.

"Hey, getcha program. Getcha program heah. Ya can't tell da playahs without da program."

I can't remember the last time I heard that. Was it at a circus? WWF Smackdown? ELECTRO 2000? Or could it have been at a Red Sox game? The objective of programs, above removing an extra $8–$20 from your wallet, is to provide information beyond what you would experience at the actual event. If you're hip, you already know the players. Even so, programs provide you with the real skinny, added info you wouldn't necessarily expect.

When I first read about the MicroConverter, not only did I find that it contains ADC and DAC in one package, there was more. Digital and analog are no longer segregated. It's been slow in coming, but it came in spite of the fact that we've been warned for years to keep digital signals clear of analog signals. This is still good practice, but today we are seeing analog and digital circuitry being combined in the same devices, and I'm not just talking about the interfacing circuitry.

One of the most prominent manufacturers in the analog field is Analog Devices. So, when I heard that the MicroConverter was being manufactured by AD, I immediately got the warm fuzzies. In all the years I've used AD parts, there has never been cause for concern, not in how the part performed nor in its availability. I bought the program so I could learn more. As it turns out, the MicroConverter is a multi-channel 12-bit ADC/DAC with embedded flash MCU memory. I had to check the cover of the program again. Yep! It says Analog Devices. I guess AD has officially entered the microcontroller market.

### THE CORE

Analog Devices chose to use an 8051 core for its ADuC812 MicroConverter. Take a look at the diagram in Figure 1. Here you will see the familiar pieces of the 8051 core.

All this and I haven't even mentioned the ADC/DAC yet. Let's look at the enhancements to the 8051 core in more detail before we get to the crème de la crème.

The trouble with most small 8051 applications is, because the 8051 is a ROM device, you need to have debugged code before signing off that it is acceptable for ROMing by the manufacturer in the frequently large runs. The prototype necessary for this debugging doesn't normally look anything like the finished product because it has to have external memory. This means adding an external memory device, the de-multiplexing latch, and decoding circuitry to support any external memory devices. Or, if money is not a concern, you might use an expensive quartz windowed micro and the necessary tools for programming and erasing the device.

The ADuC812 has user friendliness written all over its ports. The 8 KB of on-chip flash memory is not only electrically erasable, it doesn't require special programming voltages. The boot loader can be used to program the 8-KB flash memory directly through the UART port. The boot load code is executed on reset if the PSEN line is pulled low.

There is no need to add an external serial EEPROM for that calibration and configuration data because the MicroConverter has 640 bytes of on-chip data flash memory. The 640 bytes

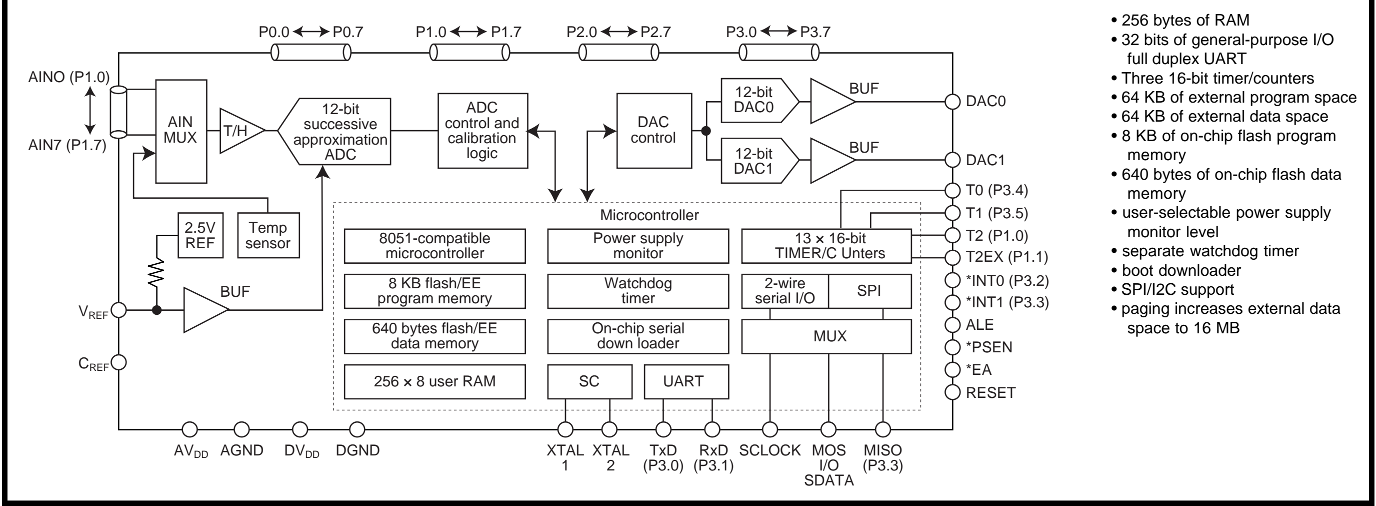



**Figure 1**—*Check out these points of interest. Analog Devices has jumped into the microcontroller market with a great product that includes both analog and digital I/O.*

are accessed through special function registers (SFRs) as 160 four-byte pages.

To protect against data corruption from VDD droop, the ADuC812 has user-selectable voltage trip points. Another new SFR, the PSMCON register, allows interrupts to be triggered on either the analog or digital supply dropping below the selected trip point between 2.6 and 4.6 V.

Along these same protective lines, a watchdog timer, which is not dependent on the main oscillator, can interrupt errant code execution. A watchdog must be reset periodically by "your good code" to prevent a timeout. If for some reason, code execution goes where no code has gone before and doesn't reset the watchdog, the watchdog overflows and creates an interrupt. This may be a result of programming error, electrical problems, or RFI. Watchdog timeouts are user-selectable from 16 to 2048 ms.

To facilitate downloading executable code into the 8-KB internal flash memory of the ADuC812, internal boot code executes upon reset when PSEN is held low during reset. This boot code allows the user to do one of five functions—erase the 8 KB of code memory, erase code and the 640 bytes of data memory, write to the code memory, write to the data memory, and jump to the user code. The communication path is the onboard UART. Analog Devices supplies a PC program to handle this communication for you, although technical note uC004 explains the serial download protocol in case you're the type who must take complete and utter control of the situation.

As with many of today's micros, both SPI and $I^2C$ are supported on the ADuC812. Although SPI has a faster overall throughput, it usually requires individual chip selects for all connected peripherals. On the other hand, the $I^2C$ protocol includes addressing within the packet that reduces interconnections but lengthens packet size and reduces overall throughput.

The addressable external data space has been expanded to 16 MB. The high-order byte (A16–A23) is output on A8–A15 during ALE (at the same time that A0–A7 is output on AD0–AD7). This high-order byte comes from a new data pointer register, the Data Pointer Page register (DPP).

## A/D

The ADC conversion block (as shown in Figure 1) incorporates a 5-µs, 8-channel, 12-bit, single-supply A/D converter. The front end track-and-hold multichannel mux shares the Port 1 I/O, allowing any of the eight Port 1 I/Os to be used as analog inputs. The analog input range is 0 V to $+V_{REF}$, where $+V_{REF}$ can be the 2.5-V internal factory calibrated low drift reference, or any input from 2.3 V to $A_{VDD}$. Conversions can be initiated by software, an external input, or by the special DMA mode. In this special mode, continuous conversions can be directly shifted into external RAM space without interaction from the MPU. Try to get that kind of continuous throughput with other systems where you need to handle converted data on interrupt!

Factory programmed calibration coefficients can be overwritten by the user, if necessary, to tweak the effect of a change in acquisition clock frequency, reference, or supply voltages.

The DMA mode requires the user to pre-configure the external RAM as a table with entries of the A/D channel to sample. This may simply be a single channel number placed in the upper nibble of each of the RAM's double-byte locations and used to hold a 12-bit conversion resultant. A high nibble of FF indicates an end of table or DMA STOP command. With this preconfigured table, any channel can be converted in any order based on what the user has placed in the RAM table. This makes for some interesting possibilities.

In addition to the eight input channels, an absolute on-chip temperature can be sampled. At 25°C, the output is 600 µV. Temperature changes increase or decrease this output at a rate of 3 µV per degree. This ninth channel is also available in the DMA mode.

## D/A

The internal dual 12-bit DACs are truly a significant innovation in on-chip peripherals. With a full-scale (0

to $+V_{REF}$) settling time of 15-µs, these pacesetters round out the Micro-Converter's feature list. Both DACs can be updated together using the sync bit in the DACCON SFR. For instances where eight bits are enough, 8-bit mode automatically routes the 8-bit values into the top eight bits of each DAC.

## INTERRUPTS

Three new interrupts are added to the original six, the power supply monitor, end of conversion, and the SPI interrupts. Of these, the power supply monitor is pre-configured as high priority only. All other interrupts can be ordered as either high or low priority. This allows any interrupt to be configured as the most important and interrupt another interrupt (except for the power supply monitor).

## WWW

At Analog Devices' web site, you can find datasheets and other technical documentation about the ADuC812. You can also download a box of tools there. You'll find four important products—an assembler, simulator, debugger, and downloader.

The 8051 assembler was written by MetaLink. This supports most of the offshoots of the 8031 core parts manufactured today. This cross assembler will take your source code file, written with your favorite text editor, and create the Intel hex file compatible with the ADuC812.

The simulator was written by Vault Information Services. ADSIM812 loads an Intel hex file into a simulated ADuC812 environment. This allows you to step through your code line by line, verifying that your choice of syntax interacts with the processor in the manner you intended. Almost every register and function can be explored. Not only is this tool great for evaluating the device, but you will find it extremely useful for debugging purposes without ever touching a piece of hardware.

The debugger was written by Accutron Ltd., and it provides control of the program running on the target system via the serial port. The UART, timer1, RXD and TXD pins cannot be used by your program. Similar in function to the simulator, the debugger runs the code on the target system. This allows you to check the physical devices attached to the processor. Real data can now be processed.

For an even less restricted target check, Accutron offers a hardware emulator that removes the serial port restriction from the mix. (Note: the emulator costs extra and is not included in the free tool kit.) However, the debugger does give you complete control to erase and download code to both the on-chip program and data flash memory areas.

Luckily, a standalone downloader written by Accutron is included. This no-nonsense loader will pass your Intel hex file to the ADuC812 where it will be programmed into flash memory and executed upon reset.

Code examples are supplied demonstrating various functions of the ADuC812. They can be used as a guide when writing your own programs, or simply downloaded into the tools for exploration purposes. To help with support, schematics and gerber files are included for their evaluation board. If you've got the guts, you can make your own eval PCB. If you're under a time constraint, you may want to buy an assembled evaluation unit from Analog Devices.

## DESIGN IN

You will inevitably be forced into SMT to use the ADuC812 because it is only available in a 52-pin PQFP. But don't let that stop you. As you will see next month, it didn't stop me. I'll continue this discussion with what I chose as a flexible complement to the ADuC812.

*Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on* Circuit Cellar*'s engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

## SOURCE

**ADuC812 MicroConverter**
Analog Devices, Inc.
(781) 329-4700
www.analog.com

## SILICON UPDATE

Tom Cantrell

# We Ride the Waves

Follow Tom's column for any amount of time and you'll find out just how he feels about wires. With the introduction of Zilog's new Wave chip, Tom's wire woes may be coming to an end, and that's good news for all.

I've said it before, and I'll say it again. I hate wires. My long-suffering wife has learned to tolerate my packrat tendencies, but even I have to agree that my pile of cables has gotten out of hand. Somewhere along the way, the collection reached a critical mass, at which point it became easier to just purchase a new cable rather than sort through the knotted mess. Of course, with each new addition matters become worse, perpetuating the action of buying as opposed to organizing.

Some people collect balls of old string. There was a time when I'd laugh and think "get a life," but now the joke's on me. Brace yourself while I run down the lengthy list—there's DB-9s, -15s, 25s, Centronics, floppy, SCSI, USB, keyboard, ADB, AppleTalk, phone, video coax, speaker wire, AC, ribbon with plenty of permutations of male and female, not to mention a full quiver of gender changers, null modems, and breakout boxes.

Of course, once you have the proper cable in hand, the fun has just begun. How often have you ended up under the desk, cursing while trying to find the mythical blind insertion point? There's always at least one wrong way to line things up and it amazes me how often that's my first guess.

Seeing how hard it is to make a connection in the first place, it should be easy to break it, right? Fishing a cable out of the knot is an accurate description. Have you ever noticed how closely connectors with thumbscrews resemble a fishhook? It's inevitable that the cable you're yanking on will get hung up so badly even Moby Dick wouldn't have a chance of breaking through.

I look forward to the day when wires meet their maker, and slowly but surely I think we're making progress. Thanks to radio, there are currently all manner of wireless schemes afoot that offer the promise of cutting the cord for PCs, just as it's been cut for phones.

Indeed, the idea of drafting a cordless phone for data duty has crossed my mind more than once. It could be as simple as using a regular modem with acoustical coupling. It is likely that the older, simpler 300- and 1200-bps transfer standards would work, though I have my doubts about high-speed modes like the 56k. If speed or modem interoperability isn't an issue, you could just homebrew your own simple modulation scheme in software.

It seems the cordless phone idea struck a chord at Zilog too. The difference is they happen to have a cordless phone chip on the market already. Why not just tweak the phone software to add wireless data capability?

### DOUBLE-DOUBLE-DUTY DSP

The Zilog Wave chip looks a lot like the Z89xxx DSP I covered a few years back in "Double-Duty DSP" (*Circuit Cellar* 91). The title reflected my musings about the trend towards blurring the distinction between processors and DSPs, a subject that's been around for a while (see also "To DSP or Not to DSP" by Michael Smith in *Circuit Cellar* 28).

The point of my article was that with Harvard architecture, fast multiply and add, and other architectural trinkets (e.g., zero overhead loops), the Z89*xxx* is well suited to carry the DSP label. However, with built-in serial and A/D ports, timers, and a $10 price tag, the Z89*xxx* could just as

easily be considered a 16-bit controller. Have it your way.

Moving back to present day, take a look at the Zilog Wave chip (aka, Z87L0x). You'll swear you are seeing double and, in fact, you are. That's because the Wave integrates two complete Z89*xxx*-type DSPs (see Figure 1).

As an aside, the idea of packing multiple processors on a single chip isn't really new, especially when one of them is relegated to a co-processor role. Of course, desktop chips brought their formally external floating point co-processors onboard long ago. Another example that comes to mind are the Motorola micros that incorporate an autonomous Time Processing Unit (TPU).

The idea, via the Wave chip, of integrating multiple copies of the same core is a bit more radical. However, as the ability to milk ever more MIPs from a single processor tapers off, expect to see more Chip Multiprocessors moving from research and development labs to market.

## HOP ALONG CAPACITY

The Wave chip is designed to do the heavy lifting in cordless telephone applications. One chip goes into the base station and one into the handset. The minor difference in functionality between base and handset boils down to who gets the most sleep (the handset, to conserve battery power) and can be accommodated with a single ROM code and jumper setting.

The chip is designed to take advantage of the 900-MHz ISM (Industrial, Scientific, Medical) band. The good news is that, unlike other low-cost RF solutions, the 900-MHz band isn't subject to the strict FCC restrictions of transmit power and duty cycles that rule out the use of garage door openers and keyless remote class technology for data applications.

The first DSP (DSP1) manages an external 900-MHz RF transceiver (see Figure 2) and implements the Frequency Hopping Spread Spectrum (FHSS) wireless protocol. This DSP is supplemented with the special purpose circuits that comprise a SuperHeterodyne radio, including Local Oscillator and FSK modulator and demodulator.

The spread spectrum technique changes or spreads the radio frequency every 4 ms. The bandwidth of the ISM band (902 to 928 MHz) is partitioned into 142 channels of 182.044 kHz. Time division multiplexing devotes half of each 4-ms frame to transmitting and half to receiving for virtual full-duplex operation (see Figure 3).

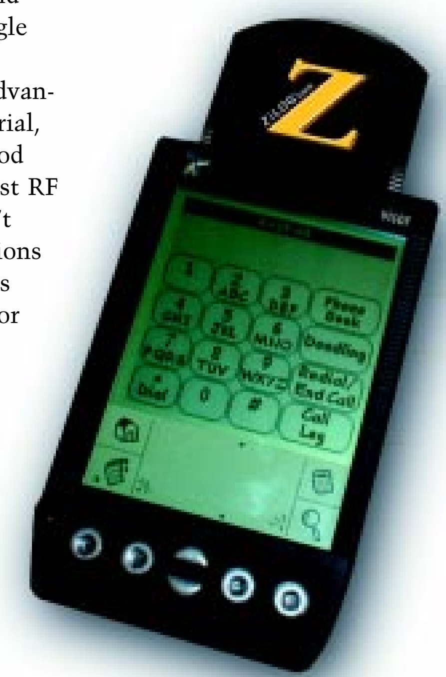


**Photo 1—***In Zilog's "extremely connected" vision, every hand-held wireless gadget in your house (phones, TV, stereo, X-10, etc.) can get rolled into one unit like this prototype Handspring Communicator.*

The spread spectrum approach has a number of well-known advantages including immunity to narrow-band interference and a measure of security because the output is difficult to distinguish from noise. A potential eavesdropper would need to know the hopping sequence and have a similar frequency agile receiver. Gone are the days when anyone can listen in if they own a portable scanner.

To further enhance robustness, the Wave chip utilizes adaptive frequency hopping. At any given moment, 64 of the 142 channels are active. The chip consistently monitors the integrity of the link, and will dynamically reassign marginal channels to ones that are clear.

In phone applications, the second DSP (DSP2) handles voice processing including Adaptive Delta Pulse Code Modulation (ADPCM) transcoding that converts and compresses the raw digitized voice signal. It's also responsible for other audio functions such as DTMF decode and generation, call progress monitoring, caller ID, and more.

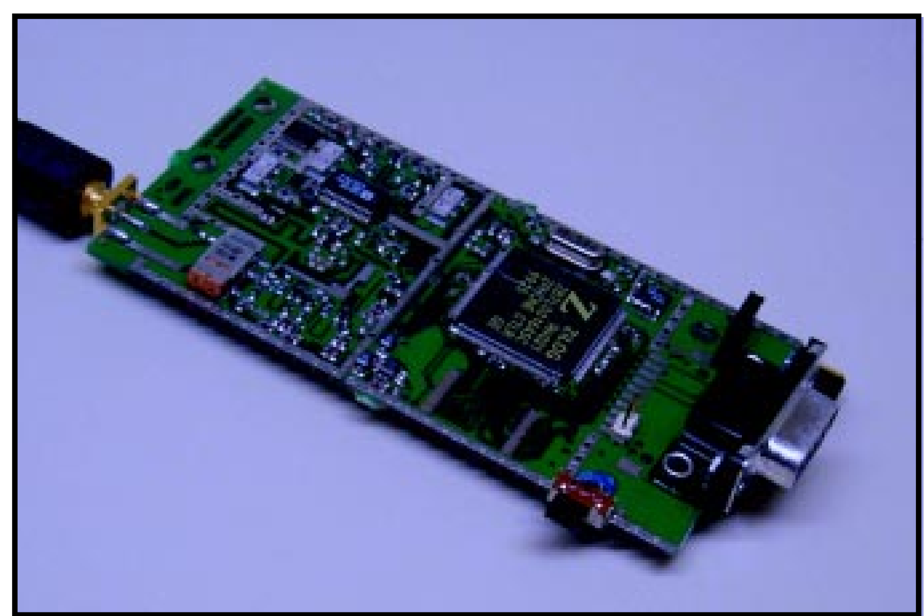


**Photo 2—***Zilog offers a Wave-based OEM and evaluation wireless data modem, with RF on one side and RS-232 on the other.*

Each DSP has its own complement of memory. DSP1 includes 32K × 16 program ROM, 512 × 16 data RAM, and 1K × 16 scratchpad RAM that can handle code and data. DSP2 gets by with 16K × 16 ROM, 512 × 16 RAM, and 1K × 16 scratchpad. Notice that the DSP2 scratchpad is accessible by both DSP cores.

This arrangement provides a mechanism for data exchange and dynamic configuration. There's also an 8-bit channel that provides a direct interface between DSP1 and DSP2.

One version of the part (Z87L03) comes in a higher pin count package (144- instead of 100-pin QFP), which supports external program memory (i.e., EPROM or flash memory) for DSP1 for those designers who aren't ready or willing to commit to ROM (Z87L02).

There's also a hybrid module version (Z87M02) that offers onboard flash memory while maintaining the ROM version pinout for easy prototyping and pre-production.

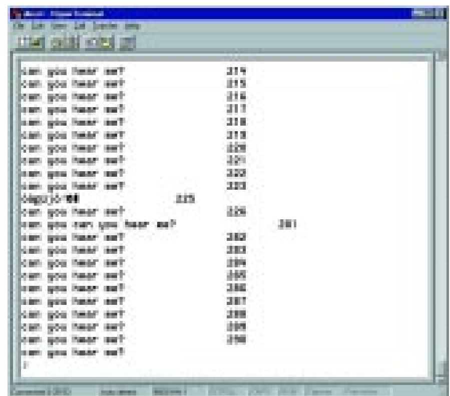


**Photo 3—***Real-world (i.e., through walls, no shielding) range of the 0.25-W modem is a few hundred yards. Here the link goes marginal and is lost at the limit, then recovered when it is back within range.*

Rounding out the chip are various ports, including parallel and serial ($I^2C$) I/O. A three-channel multiplexed 8-bit A/D converter can be used for functions such as battery monitoring and signal strength indication. A 4-bit DAC conveys the desired transmit power level so software can dictate the minimum power required to maintain the link, thus extending battery life.

## DATA TIME

Zilog envisions an "extremely connected" future in which something like the prototype Handspring Communicator shown in Photo 1 will become as indispensable as your TV remote. I must admit, it would be nice to have one gadget that does everything, without any wires of course.

In the meantime, because voice data is already handled digitally these days, it's trivial to put the Wave to work as a basic wireless modem. It's simply a matter of stripping out DSP2's ADPCM processing and replacing it with a simple software UART. Between the dual DSPs, there's spare horsepower available to handle extras that are application-specific such as display, keypad, touchscreen X-10, IR, web access, and so on.

To make things even simpler, Zilog is offering a simple RF modem module both as an evaluation kit and as an OEM unit (see Photo 2). For versatility, the module utilizes the '03 Wave variant with an external flash memory chip.

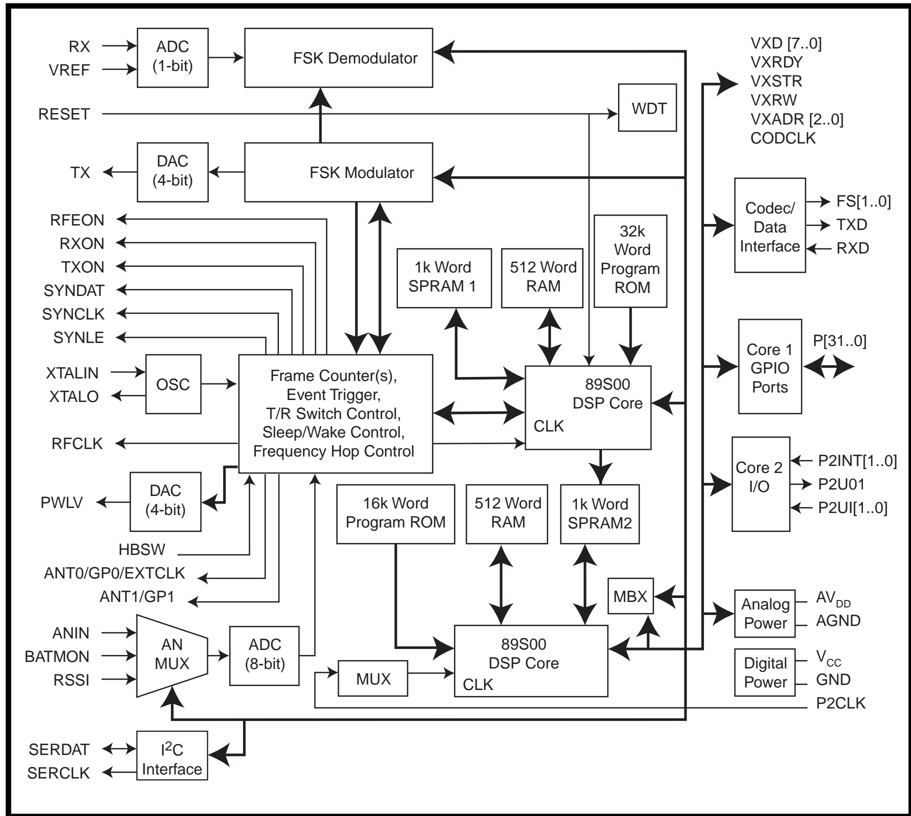



**Figure 1—***With two complete DSPs onboard, the Zilog Wave chip has enough horsepower to handle wireless voice and data applications.*

After booting from the external flash memory at power-up, DSP1 downloads the RS-232 driver code from the flash memory into DSP2's scratchpad RAM. After being downloaded, DSP1 releases DSP2 from reset and allows it to run. At this point, DSP1 is handling the 4-ms (i.e., one frame per hop) control loop, while DSP2 is sending and receiving RS-232 data as required.

The overall scheme supports up to 16 modules (each with a unique unit ID number) that comprise a miniature neighborhood of sorts. Initially, all modules operate as slaves in a sleep and wake cycle. Periodically (every 5 s), each unit wakes up and listens for a link request.

An interrupt from DSP2 (i.e., incoming RS-232 character) turns a module into a master, at which point it exits the sleep and wake cycle and starts transmitting a link request, including the destination unit ID. Within 5 s the ID'd unit wakes up and detects and acknowledges the link request. At this point, both master and slave are executing the 4-ms control loop, alternately sending and receiving data. After the needed data transfer is complete, both units revert to the sleep and wake cycle.

The RS-232 code (about 300 words) running on DSP2 relies on a 100-µs tick interrupt. Because the communication channel between the DSP cores is only 8-bits wide, 4 bits of the RS-232 data and 4 bits of control information are transferred every tick as shown in Figure 4.

## RADIO ACTIVE

I managed to get a couple of early prototypes of the modules for evaluation that had a few cut-and-jump attributes as well as some firmware limitations. The RS-232 port parameters were fixed at 9600 8-N-1, for example.

Different from the miniature neighborhood peer-to-peer model that I described earlier, the prototypes that I was testing operate in beacon mode. In beacon mode, one unit is assigned as the base station and all others are handsets through a DIP switch setting.

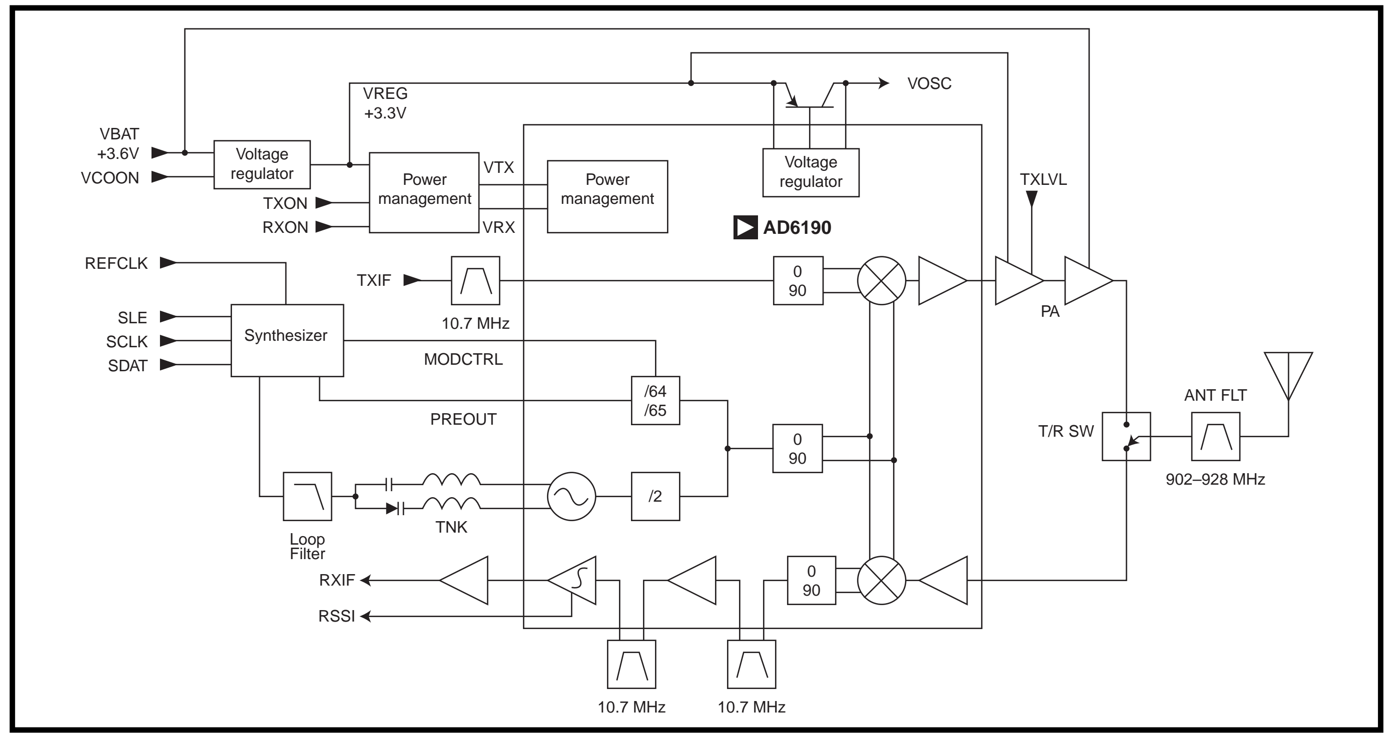



**Figure 2—***The Wave-based wireless data modem module takes advantage of 900-MHz cordless phone technology by utilizing an Analog Devices RF subsystem.*

Finally, the module is designed to utilize RS-232 handshake lines to initiate and acknowledge a link request. The idea is to assert DTR to the module to request the RF link establishment. After link establishment is complete, the module sends DSR back. However, the units I received had a small switch patched in for manual operation.

Note that in a real application, especially one where power consumption is an issue (e.g., battery-driven), it is imperative that you utilize software to shut down the link between transfers. When the link is up, power consumption of the module is almost 1 W (3.6 V at 250 mA) because the Wave chip and 0.25-W RF transmitter are running full bore.

The one-page datasheet I received with the modules mentioned a precaution about using the RS-232 handshake lines for link control. After you complete an RS-232 transmission to the module, you should wait a while before de-asserting DTR to make sure the last character makes it across the air before the link is brought down.

To start, I connected the modules to the serial ports on my desktop and laptop PCs. Fortunately, I have a bench power supply with adjustable output that gave me easy access to the 3.6 V required. Otherwise, I would have had to scrounge for some batteries, or wire up an adjustable regulator.

With all systems go, it was a simple matter of running a terminal program on both PCs to verify that all was well (i.e., whatever you type on one PC's keyboard appears on the other screen too).

Having passed the first test, I decided to explore further. I connected one of the modules to an SBC and wrote a no-brainer program that simply outputs a `Can You Hear Me` message with a sequence number once per second. Then I carted an admittedly ugly lash-up comprised of the laptop, the Zilog modem, the variable power supply, and a cigarette lighter 12-V to 120-VAC inverter out to the car and started driving.

As you can see in Photo 3, the link held up for a few hundred yards. The signal was passing through a number of walls, not to mention the body of the car and, due to the layout of my neighborhood, I couldn't even see my house by the time reception was lost. Furthermore, the prototype units I received didn't have the shielding on the circuit boards that Zilog intends for production units, no doubt upping the interference ante.

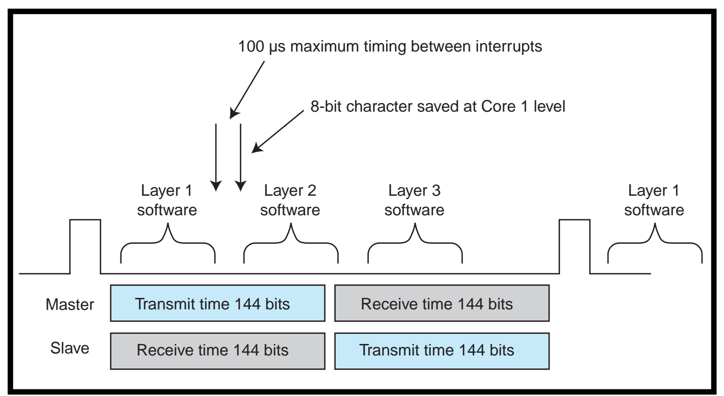



**Figure 3—***The Wave chip transmits and receives for 4 ms between frequency hops. The software running on DSP1 consists of three layers. Layers 1 and 2 manage the RF link and messaging among layer 3 and other transceivers. Layer 3 manages the user interface and the data path between DSP1 and DSP2 via an interrupt service routine.*

Zilog estimates that the unit should be good for a range of up to one mile, but that's in free air, line of sight, with proper shielding. They are investigating options for built-in software error detection and correction. Of course, you could add those features (such as `CRC` and `Retry`) to your application level software as well.

In any case, the smaller range I achieved with the prototypes was promising. Certainly adequate coverage for the typical homestead, unless you happen to live on the Ponderosa.

## BYE BYE CABLE GUY

There's no shortage of radio-based initiatives on the table—Bluetooth, HomeRF, wireless Ethernet, and others. Frankly, it's difficult to keep track of them all. Eventually, as usual, I suppose we'll muddle through the confusion and commercially successful solutions will emerge.

In the meantime, why not take advantage of the 900-MHz cordless phone technology? It may not be a whiz, but it works.

Whatever the answer turns out to be, it can't come soon enough for me. I've had it with cable chaos and can't wait to see the look on my wife's face when the day finally comes that I cart the whole tangled mess out to the trash can where it belongs. ■

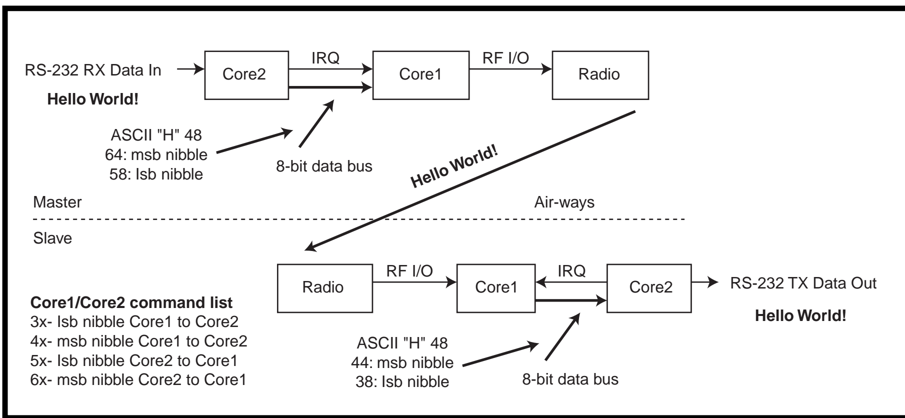



**Figure 4**—*Notice that DSP2 passes RS-232 data to DSP1 for RF transmission. Because the channel is only 8 bits wide, each byte of data requires two transfers, each made up of 4 bits of data and 4 bits of control information.*


*Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for several years. You may reach him by e-mail at tom.cantrell@circuitcellar.com or by telephone at (510) 657-0264.*


## SOURCE

**Wave Chip**
Zilog, Inc.
(408) 558-8500
Fax: (408) 558-8300
www.zilog.com

# CIRCUIT CELLAR Test Your EQ

**Problem 1**—Why is it a generally a bad idea to write lengthy ISRs?

**Problem 2**—Significant effort continues to be put into both cable/fiber and satellite technologies for wide-area communications. Why would you pick one over the other for a particular application?

**Problem 3**—What is the difference between an interpreter and a compiler?

**Problem 4**—You are given two light bulbs in a physics lab and you are asked to measure their current at different voltages. You get the data shown in the following table. What's going on here? (Try sketching a graph of the data.)

| Applied Voltage | Bulb #1 Current | Bulb #2 Current |
|---|---|---|
| 10 V | — | 105 mA |
| 20 V | — | 133 mA |
| 30 V | 95 mA | 155 mA |
| 40 V | 132 mA | 179 mA |
| 50 V | 168 mA | 195 mA |
| 60 V | 208 mA | 216 mA |
| 70 V | 250 mA | 235 mA |
| 80 V | 296 mA | 252 mA |
| 90 V | 345 mA | 268 mA |

What's *your* EQ?—The answers and 4 additional questions and answers are posted at www.circuitcellar.com.

You may contact the quizmasters at eq@circuitcellar.com.

8 more EQ questions each month in Circuit Cellar Online see pg. 2

# PRIORITY INTERRUPT

## First on the Block

OK, I'll admit that I am a technology junkie. I was the first on my block to have a PC, a VCR, a projection TV, and a digital camera. I have TVs and video monitors everywhere (come on, I can't be the only one with a TV/home control monitor in the bathroom). In fact, I was so ahead of my time that today's automotive computers are old hat compared to the dash-mounted DAS system I had in the '70s.

For the most part, new technology is easy to justify because it just plain works better. You buy a pocket LCD TV to catch the news or add a little entertainment for the daily train commute. You buy the GPS for the car so that no one has to argue about who's going into the gas station to ask for directions. You subscribe to cable TV, install the mother of all YAGI TV antennas, and add an 18″ DSS dish next to the 9′ C-band so you can justify enough signal sources to buy a new HDTV.

In truth, however, I've mellowed over the years. I don't go out (as much, anyway) and just buy something because it's new these days. I'd like to say the reason is because I'm studying the technology and maximizing the price-performance before I buy, but the truth is simpler. A lot of this new technology really isn't ready as far as I'm concerned.

I remember when I bought my first cellphone (to give you an indication of how long ago that was, let me just say that I paid $3500 for it). I needed it to keep in touch with the office. Unfortunately, there were so few cellsites in Connecticut at the time that I could throw a rock through the office window from further than I could call on the cellphone. You wouldn't have thought that from their advertising before I bought it, of course.

OK, it took 15 years and they've cleaned up their act. I've had six cellphones since and there are very few "no service" locations left in Connecticut. I should be happy, right? Well, ordinarily yes, but now I'm being barraged by all the communication companies to sign up for wireless services. My gadget-happy side loves the idea of wireless e-mail, real-time traffic and GPS updates, downloaded music, and wireless Internet connections.

Most people presume it's just a matter of updating their old analog cellphone to a new digital version. Well, not quite.

First of all, there is no guarantee that the digital networks can handle all this wireless traffic they want us to buy into. Worse yet, they haven't even settled on a standard. You have a 75% chance of picking the wrong phone if you aren't careful to scope out the options first. There are four distinct digital phone networks: Code Division Multiple Access (CDMA), Global Standard for Mobile Communications (GSM), Nextel National Network, and Time Division Multiple Access (TDMA). All of these networks operate on 1900 MHz but are incompatible with each other.

I suppose that even if these guys don't sort out the mess, market demand will sustain adequate coverage for all the services but that doesn't help you every place. Reviews in recent magazines suggest that you should choose CDMA if you want the best signal quality. If you want the maximum geographic coverage in the US then choose TDMA. If you travel to Europe you will want to go with GSM. If you are going to Asia, stick with CDMA.

Finally, there is the issue of cost and speed. At the present time, surfing the wireless 'Net isn't cheap. Most nontrivial wireless Internet uses utilize a cellular digital packet data card (CDPD) and require an ISP subscription. Most of us are used to paying about $20 a month for a 56k modem connection (if my part of CT would ever get out of the dark ages we might actually have cable or DSL modems some day). Wireless digital services currently average $50–$60 per month. The good news is that someday it will have respectable speed. The bad news is that today, wireless communication is limited to 9600 bps–19.2 kbps, depending upon the network. I know that's fine for e-mail but try downloading one MP3 music selection to your PDA for the train ride and you'll be home before you get to listen to it.

I'm going to try to ignore the hype for a while. Wireless Internet sounds great and I really want this bandwagon to succeed. Before I start reaching for the phone to order, however, I need to forget that it was more than 10 years ago that I saw HDTVs at the Consumer Electronics Show. The claim then was that they'd be in general use within 3 years!

steve.ciarcia@circuitcellar.com