EMBEDDED PC MONTHLY SECTION

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#103 FEBRUARY 1999

REAL-WORLD PROGRAMMING

Getting Down to Speed with Practical Control

Step into Pressure Sensing

Real-Time Linux Development Tools

C++ Embedded and Efficient

TASK MANAGER

Avoiding the Fuzzy Faux Pas

For several years, I've edited issues focusing on fuzzy logic for *Circuit Cellar INK*. But this is the first time that "Fuzzy Logic" hasn't been the theme even though three articles in this issue use this approach.

As many people suggest, there's a stigma associated with the word "fuzzy." Perhaps for some of the hardboiled hardware types out there, it's just too cute. I can appreciate that.

In fact, I'll let you in on a little behind-the-scenes action. During the recent holiday season, we were pondering cover photo concepts for this issue. And if you were in the States this past year, you know what every kid wanted for their holiday gift, right? Furby, of course!

For those of you who haven't made his acquaintance, this little bug-eyed furry creature "acquires" language by being talked to, or so the ads say. (I won't even start a tirade on what it means to acquire language, although if you see me at the Embedded Systems Conference next month and ask my opinion, I'll be happy to give you the benefit of ten years of graduate study in linguistics!) By the way, if you want to learn about some of the hardware truths beneath Furby's furry exterior, check out www.phobe.com/furby.

Anyway, as I was saying, Furby was on our minds and someone said, "Wouldn't it be funny to put Furby on a Fuzzy Logic cover?" Sure, but we knew it was going to be next to impossible to get our hands on one of these popular toys. Besides, I'm not sure it uses fuzzy logic. So, that idea went out the window.

We had to come up with something else, but with the words "Fuzzy Logic" on the cover, it just wasn't working. All we could envision were fuzzy animals, fuzzy houseslippers,.... Those cuddly images would just get us into trouble with people who seem to believe that fuzzy logic is imprecise and unprofessional simply because it uses *that* word.

To get beyond the fur fest, we started thinking about what fuzzy logic is: balancing, weighing variables, a certain way of evaluating parameters. But since the last two Fuzzy Logic covers have embodied the "different way of looking at the situation" concept, that now-stale idea went out the window along with Furby.

Clearly, putting "Fuzzy Logic" on the cover wasn't going to happen. By now, I was getting really frustrated and started walking around the office growling, "What's wrong with 'fuzzy' anyway? Do you mean to tell me that more people would appreciate fuzzy logic if we called it something pompous like 'multiparameter weighting analysis'?"

Fortunately, once we thought about how Walter Banks and Constantin von Altrock discuss this approach in their articles, we realized that fuzzy logic is just a practical way of using embedded computers to handle real-world problems. And that's when it hit us: Real-World Programming.

Although we changed the name of the theme, I hope you'll look past any biases that "fuzzy" brings up as you read the fuzzy-logic articles. Don't let words get in the way of a helpful solution.

Eli

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER Steve Ciarcia	**ASSOCIATE PUBLISHER** Sue Skolnick
MANAGING EDITOR Elizabeth Laurençot	**CIRCULATION MANAGER** Rose Mansella
TECHNICAL EDITORS Michael Palumbo Rob Walker	**CHIEF FINANCIAL OFFICER** Jeannette Ciarcia
WEST COAST EDITOR Tom Cantrell	**ART DIRECTOR** KC Zienka
CONTRIBUTING EDITORS Ingo Cyliax Ken Davidson Fred Eady	**ENGINEERING STAFF** Jeff Bachiochi
NEW PRODUCTS EDITOR Harv Weiner	**PRODUCTION STAFF** Phil Champagne John Gorsky James Soussounis
	PROJECT EDITOR Janice Hughes

EDITORIAL ADVISORY BOARD

Ingo Cyliax   Norman Jackson   David Prutchi

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

ADVERTISING

ADVERTISING SALES MANAGER
Bobbi Yush   Fax: (860) 871-0411
(860) 872-3064   E-mail: bobbi.yush@circuitcellar.com

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

CONTACTING CIRCUIT CELLAR INK

SUBSCRIPTIONS:
INFORMATION: www.circuitcellar.com or subscribe@circuitcellar.com
TO SUBSCRIBE: (800) 269-6301 or via our editorial offices: (860) 875-2199
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 INK, 4 Park St., Vernon, CT 06066
AUTHOR CONTACT:
E-MAIL: Author addresses (when available) included at the end of each article.
ARTICLE FILES: ftp.circuitcellar.com

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

CIRCUIT CELLAR INK®, THE COMPUTER APPLICATIONS JOURNAL (ISSN 0896-8985) is 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, 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 INK Subscriptions, P.O. Box 698, Holmes, PA 19043-9613 or call (800) 269-6301.
Postmaster: Send address changes to Circuit Cellar INK, Circulation Dept., P.O. Box 698, Holmes, PA 19043-9613.

INSIDE ISSUE 103

READER I/O

HARVARD VS. VON NEUMANN APPEALED

In their article "In-System Programming" (*INK* 101), Craig Pataky and Bill Maggs imply that all embedded micros use Harvard architecture rather than Von Neumann. In fact, the Motorola 68HC05 and 68HC11 families employ the Von Neumann architecture.

Their statement, "The main reason microcontrollers have clung to the Harvard architecture is because by keeping data and code memory separate, it's impossible for the machine to inadvertently corrupt its own code and go insane" (p. 15) implies that code memory in a Von Neumann system is always volatile.

Also, the statements "Von Neumann designs are flexible but inherently unstable. Harvard designs are rock-solid but immutable" (p. 15) are extremely misleading. If my code is in EPROM or some other firm memory area, I doubt a misguided program could overwrite the code and cause the system to go insane.

A corrupted program counter register in a Harvard design is no safer than one in a Von Neumann design.

Calvin Krusen
ckrusen@meeco.com

The point of our article was to address the flexibility and cutting-edge technology of in-circuit programming in flash-based designs. Our reference to Von Neumann versus Harvard design was based more on historical implementation than technical definition.

You are correct, not all micros use the Harvard architecture. But, our statement that microcontrollers which keep code and data memory separate cannot corrupt their own code is factual. And, yes, a corrupted program register could happen to any micro, but that's generally what watchdog timers are for.

Aside from being inexpensive, the Von Neumann design permits dynamic creation and destruction of programs on-the-fly—only the minimum code required for basic operation is fixed. On the other hand, Harvard designs have almost always been ROMed.

We agree that if the micro is Von Neumann and an address region was set aside for ROMed code, the code is every bit as secure as a Harvard design. Unfortunately, this gives rise to the same problems that the article was trying to solve in the first place.

Craig Pataky and Bill Maggs

Circuit Cellar ONLINE

www.circuitcellar.com

Newsgroups

If you miss the Circuit Cellar BBS, then the cci newsserver is the place to go for on-line questions and advice on embedded control, announcements about the magazine, or to let us know your thoughts about Circuit Cellar. Just visit our home page for directions to become part of the newsgroup experience.

The February Design Forum password is:

Fuzzy

New!

- Circuit Cellar Online is hosting the proceedings from the First Annual 1998 **Embedded Internet Workshop**. Visit our homepage to find the workshop proceedings including slide presentations, lecture notes, and complete papers.
- All aboard the **INFO *Express***! Loaded with the latest news and information about *Circuit Cellar INK* as well as any additions or changes to our web site, the INFO *Express* stops right at your e-mail address. Visit our homepage to sign up for this new service from Circuit Cellar.
- While you're working on your entry for Design99, don't forget to check the **Design99 Rules Update** section for the latest updates on the contest guidlines.

Design Forum

Be sure to visit the Circuit Cellar Design Forum this month for more great online technical columns and applications. February's Design Forum password is your key to great new columns, monthly features, and PIC Abstracts.

Silicon Update Online: Zilog Lives?!—Tom Cantrell
Lessons from the Trenches: Getting a Head Start on Software Development—Linking Embedded Code—George Martin
A 68HC11 S-Record Disassembler: Frank Kuechmann

NEW PRODUCT NEWS

Edited by Harv Weiner

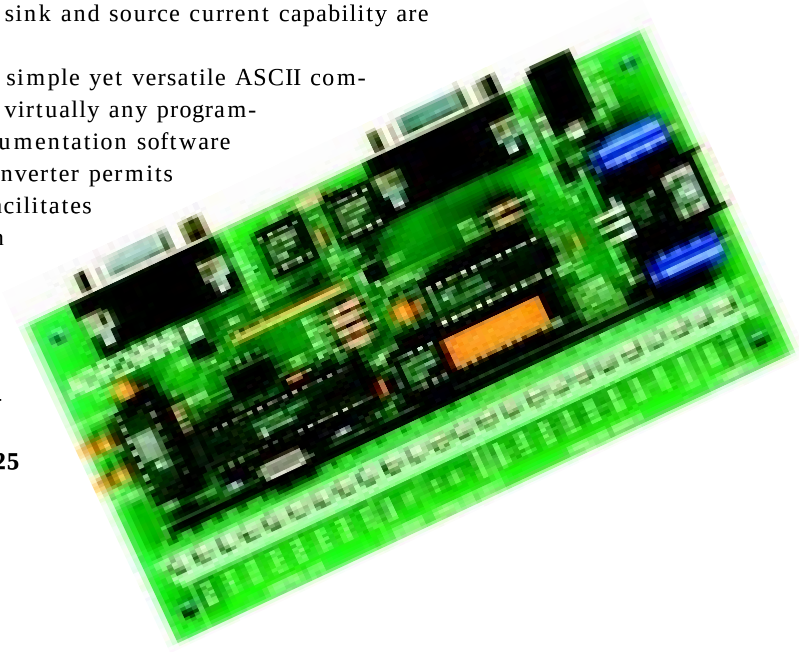
DATA ACQUISITION MODULE

The **ADR2000** is a serial data acquisition and control interface that features a host of peripherals. The board includes eight 12-bit analog inputs with a 0–5- or ±5-V input range (software selectable), two analog outputs (either 12-bit analog [V.A] or 10-bit PWM [V.B]), and a 16-bit event counter. Eight digital I/O lines with 20-mA sink and source current capability are available.

The ADR2000 can be programmed using simple yet versatile ASCII commands, and host software can be written in virtually any programming language or with most graphical instrumentation software packages. An onboard RS-232–to–RS-485 converter permits the use of hosts with either port type and facilitates daisy-chaining via RS-485. The converter on each daisy-chained unit enables other vendors' RS-232 devices to be connected to any node on the chain.

Power is applied via standard 9-V wall adapter. An auxiliary, regulated 5-VDC output is available to power external circuitry.

Pricing for the ADR2000 ranges from **$225 to $265**.

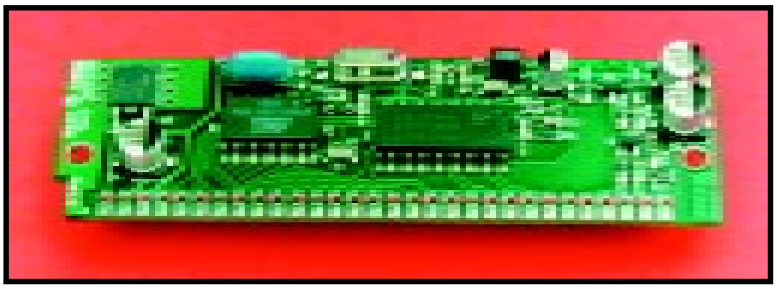


Ontrak Control Systems, Inc.
(705) 671-2652
Fax: (705) 671-6127
www.ontrak.net

VIDEO TEXT DISPLAY MODULE

A video text display module that lets a PC or microcontroller display up to 11 lines of 28 characters (308 total) on standard NTSC or PAL composite video systems has been announced by Decade Engineering. Applications include remote video inspection, robotics, recreational vehicles, home automation, security and surveillance, remotely piloted vehicles, amateur TV, and industrial process monitoring.

BOB-II generates matte background video onboard or genlocks to an external video source and superimposes its text over the image. Video mode switching is fully automatic. The video output contains a small positive DC bias, which is common to many video sources and well tolerated at the inputs of most video equipment. Internal video background signal is automatically generated if video is not supplied to the input pin. Composite sync output (separated from input video) is provided.



BOB-II's character patterns are 12 × 18 pixels. Basic character graphics and Euro-language support are provided. Character transparency and brightness are variable using external pots. Character-blinking or background-display options draw attention to important messages.

Programmed control is effected by simple commands and text sent as plain ASCII codes through a 9600-bps serial data link. Direct control from the PC keyboard is possible with any standard ASCII terminal program.

BOB-II uses a 30-pin SIMM form factor and snaps into a common 30-pin SIMM socket. It requires 65 mA from an unregulated l2-VDC source and offers a regulated +5-V output to power associated equipment.

BOB-II (NTSC) sells for **$79.95**. The PAL version costs slightly more. Application code is available at the company's web site.

Decade Engineering
(503) 743-3194
Fax: (503) 743-2095
www.decadenet.com

NEW PRODUCT NEWS

SERIAL REAL-TIME CLOCK MODULE

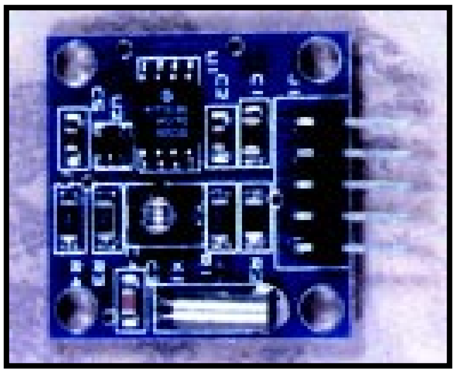
The **Pocket Watch B** contains a real-time clock, a calendar, and advanced timing features. The clock module keeps track of seconds, minutes, hours, days, months, and years. Adjustments for leap year are automatic, and the module is year-2000 compliant.

The Pocket Watch B communicates via an asynchronous, one- or two-wire, serial communications interface. Data-transfer rates of 2400, 4800, and 9600 bps are supported with autobaud detect. The module is packaged in a SIP format that measures 1″ × 1″ and operates on standard TTL levels.

The Pocket Watch B contains four advanced timing features that are accessible with the alarm command. There is a standard-level alarm, a single-shot alarm with a duration of up to 18 h, an astable alarm pulse with pulse lengths of up to 4 min. and repetition rates of up to 4 h, and an astable alarm pulse with pulse lengths of up to 4 h and repetition rates of up to 10 days.

The Pocket Watch B sells for **$24.95**. Complete datasheets and application notes are available at the company's web site.

Solutions Cubed
(530) 891-8045
Fax: (530) 891-1643
www.solutions-cubed.com



UNIVERSAL MEMORY EMULATOR

The **I-Series Universal Memory Emulator** enables a user to debug embedded systems with the same level of features as a microprocessor ICE. The emulator provides real-time access to memory devices in embedded systems, enabling the user to monitor and change target data and bus information. Complex triggers can be set on any combination of data values, address values, and external inputs from other units. These trigger events can be set to signal the host PC or external equipment and/or to force an interrupt on the target system.

The I-Series emulates virtually all types of memory chips, including EPROM, flash memory, EEPROM, and DRAM. These emulators have capacities up to 32 Mb, and support is provided for devices from 8 to 56 pins with dual 8- and 16-bit capability. Full support for flash memory command registers is provided. The emulators offer fast access times (70 ns) and ultra-fast downloading capabilities (over 400 kbps).

Reconfigurable memory resources enable the internal memory to be used in a variety of emulation and debugging applications simultaneously. Transferring file data to and from the emulator is easily achieved with any mix of file formats. The open software structure means that the user can build the display form from predefined values or from more complex user-defined structures.

Advanced debugging capabilities integrated into the design enable developers to use whatever debugging approach they desire, including a virtual serial port and live memory display and editing.

On all models, 3- and 5-V support is standard. Advanced power-management features and a low-power design facilitate reliable data backup and limit target power-supply loading.

Pricing for the I-Series emulators starts at **$349**.

Scanlon Design, Inc.
(902) 425-3938
Fax: (902) 425-4098
www.scanlondesign.com

NEW PRODUCT NEWS

VOICE PLAYBACK MODULE

The **VM-1608** is an EPROM-based digital voice module that plays back up to 128 preprogrammed messages via two independent channels with up to 64 messages per channel. This self-contained module requires only a power supply, a speaker, and a few trigger signals to operate. Applications include verbal industrial control, talking displays, exhibit sound effects, and vending-machine voice output.

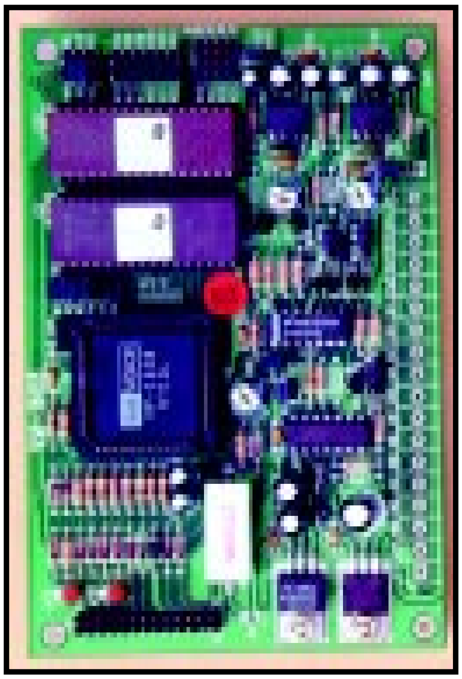
Desired messages are preprogrammed into the module with the Quikvoice development system. A user-friendly interface provides random access to messages as well as separate or mixed audio output for the two channels. The total combined message time is 8.5 min. maximum. Operation requires a single 12–24-VDC power supply.

Two EPROM sockets provide a memory capacity from 2 to 16 Mb. Each audio channel uses half of the EPROM capacity. Each has its own power amplifier, but the amplified output may be mixed together on the board. If line-level output is selected, onboard mixing is not possible.

When the board is playing a message, it ignores any further triggering until the playback is over. To stop the playback prematurely, pulse the RESET pin high momentarily.

The VM-1608 sells for **$68** in quantities of 50.

Eletech Electronics, Inc.
(626) 333-6394
Fax: (626) 333-6494
www.eletech.com



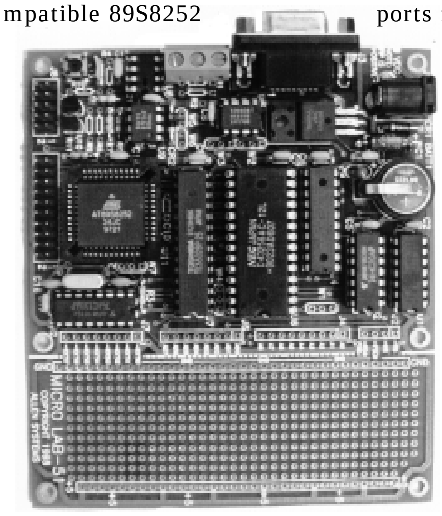
NEW PRODUCT NEWS

8051 SINGLE-BOARD COMPUTER

The **Micro Lab-51** is an 8051-based SBC that works well in educational and product development settings. In an academic environment, instructors can demonstrate real-world applications and students can implement projects. In the development lab, design engineers can create and test prototypes of their 8051-based products.

The computer uses the 8051-compatible 89S8252 microcontroller, which runs at up to 24 MHz. The chip has 8 KB of flash memory for program storage, and its flash memory can be programmed serially, so no programmer is needed.

The board also contains two 28-pin memory sockets—the first supports 32 KB of RAM, and the second supports 28 KB of EEPROM, EPROM, or battery-backed RAM. Other board features include RS-232 serial port buffering (RS-485 is optional), processor supervisory circuit with manual reset switch, and a 3.5″ × 1.5″ prototyping area. Data, address, and control lines are brought out on connectors next to the prototype area to facilitate the adding of custom user circuitry.



In its standard configuration, the Micro Lab-51 has a full-featured BASIC interpreter that supports floating-point math and most BASIC commands. An advanced version comes with system monitor debugger software and an optional 8051 cross assembler. This configuration is suited to 8051-based product development because the engineer can write assembly-language code and run it in real time under monitor control.

Prices range from **$99 to $139**. Options and projects are also available.

Allen Systems
(614) 488-7122
members.aol.com/allensys

FEATURES

FEATURE ARTICLE

Walter Banks

Fuzzy Logic Becomes Velvet Programming

We don't hear too much about fuzzy logic anymore. And, Walter wondered why. But once he started looking at the applications around him, it became clear that there's a reason for all the hush-hush. Fuzzy logic has hit the mainstream.

The real mark of a mature technology is when it is used and no one notices. Let me explain what I mean.

Just a few months ago, I was at dinner with a Japanese friend of mine and I noted that the products they developed and sold no longer advertised that they used fuzzy logic. I wanted to know why.

He responded that they use fuzzy logic now more than ever, but these days, it wasn't a feature that advertising hype found useful. He pointed out that the product still used fuzzy logic in its implementation and that, in fact, the rule base had been extended and reimplemented.

Looking back over our own fuzzy-logic experiences, I can see how it went through all the stages of a new technology from its initial idea and early users to hard evaluation by reputable scientists and finally engineering acceptance.

It took us a long time to find an application that could actually be implemented with fuzzy logic as a

Listing 1—*This small snippet of code is part of the fuzzy function for a rice-cooker temperature-control system.*

```
FUZZY cooker_temperature;
{
  IF bowl IS cold THEN
    heat is ON
  IF bowl IS too_hot THEN
    heat IS OFF
  IF bowl IS boiling THEN
    heat IS OFF
}
```

suitable engineering choice. Over time, however, engineers in various countries have found several classes of applications that are well served by fuzzy-logic technology.

Rice cookers in Japan were one of the first commercial products that proclaimed fuzzy logic as an advantage to the consumer. This mainstream commercial product is used for every meal in almost every Japanese household. It also represents one of the application areas where fuzzy logic is a particularly good solution.

Practical fuzzy-logic applications are divided into four general types—fuzzy control systems, knowledge bases, imprecise comparisons and application scaling. In this article, I want to briefly address each of those areas.

FUZZY CONTROL SYSTEMS

Fuzzy control systems are voting systems that evaluate a number of strategies to solve a particular problem. They then weigh those options to find the solution that best fits the current requirements. The Japanese rice cooker fits into this category.

This simple control problem has many variables. Temperature, time, volume, and mix all affect the outcome. For example, as the rice cooks, some of the properties of the mix change (e.g., the boiling point).

Fuzzy logic is one of the few solutions in which it is simple to implement nonlinear control systems that depend on finding a workable solution for many different circumstances.

Consider this classic fuzzy control system. The controller software consists of a fuzzy function and consequence functions for each of the controlled variables.

Listing 1 shows part of the temperature control system for a rice cooker. The present state of the cooking bowl is tested against several fuzzy functions and the consequence (i.e., heat position) is controlled as a result of these tests. As in most fuzzy-logic control systems, each fuzzy rule represents a simple common-sense statement about the application.

Such control systems are regularly implemented with very small embedded processors, especially in consumer products. Other good examples of fuzzy control systems include camera focus and household environmental control.

IMPRECISE COMPARISONS

Because we use the term “fuzzy,” fuzzy-logic systems are often thought of as having imprecise comparisons. But the truth is that, unlike conventional binary comparisons, a fuzzy comparison returns a result that is scaled between 0 and 1. In other words, these systems are more—not less—precise.

Listing 2 shows an industrial weigh-scale C-code fragment of just such a fuzzy comparison. The main line call (i.e., `x = ABOUT(bag,200,10)`) checks the bag weight of 200 g within 10 g and returns a value between 0 and 255 to indicate the degree of truth.

A bag value of 200 returns 255, and a bag value of 195 or 205 returns 128. If the bag is outside the range 190–210, then the returned value is 0.

The required code for imprecise comparisons is easily implemented on most embedded processors.

Listing 2—*The fuzzy comparison for equality is implemented with a small C function called* `ABOUT`.

```
#define F_zero 0
#define F_one 255

unsigned char ABOUT (int v , int cp , int delta)
{
  int r = ABS(cp - v);
  if (r > delta)
    return (F_zero);
  else return((r / delta) * (F_one - F_zero));
}

void main (void)
{
  …
  x = ABOUT(bag,200,10);
}
```

APPLICATION SCALING

The concept of application scaling is a big win in embedded systems applications. Consequently, it is a common practice.

If the significant variables in an application are all scaled to a standard range of fuzzy 0 to fuzzy 1 with out-of-range values limited by the value of either fuzzy 0 or fuzzy 1, then the amount of application code needed to make decisions with these variables is significantly reduced. In all the applications I’ve seen (with one exception), the variables have been scaled to be within a single-byte range.

Fuzzy-logical operators—fuzzy AND, OR, and NOT—can manipulate and combine the variable data. This

Listing 3—*Many applications may be simplified by scaling the +Scaling significant variables.*

```
#define F_OR(a,b)  ((a) > (b)) ? (a) : (b)
char Ftemp,Fhumidity,swelter;
int temp,humidity;

unsigned char Scale(int v, int lower, int upper)
{
  if (v > upper ) return (F_one);
  if (v < lower) return (F_zero);
  return (((v - lower)/(upper-lower) ) * (F_one - F_zero));
}

void main (void)
{
  …
  Ftemp  =  Scale(temp, 50,90);
  Fhumidity = Scale(humidity, 60,100)
  Swelter =  F_OR(ftemp, Fhumidity);
}
```

approach to application implementation has kept low-cost processors in many high-volume consumer products.

The code fragment in Listing 3 shows how application scaling can be implemented. In this example, humidity and temperature are scaled over the significant range of interest. A new fuzzy variable, `Swelter`, is set based on the discomfort level that the temperature or humidity produces.

The `F_OR` function referenced is the minimum of two variables. This function takes very little embedded system resources in an application.

KNOWLEDGE BASES

Fuzzy-logic rules carry with them information about a problem. It is common, for example, to say that if the room is hot then turn down the heat. "Hot" is an abstract value that becomes important when it's associated with an action.

Fuzzy control systems decide actions based on a database for fuzzy functions, each competing for resources. Information databases use fuzzy logic to weigh facts and arrive at conclusions.

There are some obvious uses for this approach to knowledge bases. Determining the potential causes of an illness is a good example.

One novel example of a fuzzy knowledge database uses fuzzy-logic tools to create a computer player in a role-playing game. The behavior of a fictitious alien race is described with a fuzzy function.

The current circumstances are sent to several different alien descriptions, producing consequences for each simulated character. It takes surprisingly few rules to create new alien species that are recognizable by their behavior.

TOO CLOSE TO SEE

My point: fuzzy-logic tools and technology have matured to the point that they've become mainstream tools. The underlying principles have survived the validity tests of science and the usefulness tests of time.

Fuzzy logic enables you to encapsulate a large problem in a form that can be solved by a very small processor. These days, it helps build huge databases of important information as well as solve many of the tricky problems associated with nonlinear control systems.

My Japanese friend was right. We don't talk about fuzzy logic much anymore. We just use it.

Walter Banks is president of Byte Craft Limited, a company specializing in software tools for embedded microprocessors. His interests include highly reliable system design, code-generation technology, programming-language development, and formal code-verification tools. You may reach him at walter@bytecraft.com.

FEATURE ARTICLE

Constantin von Altrock

Truck Speed Limiter Control

Is your teen learning to drive? Are you worried? If you think they lack logic, maybe you should consider fuzzy logic. Constantin uses fuzzy rules to consistently limit the speed of commercial trucks, despite weight variables and truck type.

In Europe, commercial trucks with a maximum load of over 12 tons are legally required to be equipped with a device that limits their maximum speed to 53.3 mph (86 km/h). Designing an algorithm for this control problem was no easy task.

The speed-limiter device needed to be compatible with a variety of trucks, all of which exhibit different behaviors. Additionally, the dynamic behavior of a truck varies depending on whether it is fully loaded or empty.

Conventional control algorithms, like PID controls, assume a linear model of the process under control, so they couldn't be used. A mathematical model of the truck would be tough to build and would require too much computational effort from an 8-bit micro. So, the designers turned to fuzzy logic.

This article focuses on the electropneumatic design of the speed limiter. The pneumatic cylinder mechanically limits the throttle-opening angle of the fuel-pump arm, and a pulse-proportional electromechanical valve controls the cylinder pressure. The electromechanical valve connects to an electronic control unit that uses a microcontroller to drive the valve according to the speed of the truck.

CONTROL REQUIREMENTS

When the truck approaches the maximum velocity, the pneumatic valve reduces the throttle-opening angle of the fuel-pump arm so the maximum velocity (V_S) is not surpassed. Even if the driver continues to push the accelerator, the speed limiter has to ensure a smooth ride at the maximum velocity.

However, because of the dead time and nonlinearities involved with this control action, an actual overshoot and hunting occur when using a proportional or on/off controller. Adding a differential and integral part yields a PID controller model.

A PID controller generates the command value as a linear combination of the error (*P*), the derivative of the error with respect to time (*D*), and the integral of the error with respect to time (*I*). To tune a PID controller, the combined weights of these three components must be chosen so they approximate the nonlinear behavior of the process under control at its operating point.

Although this technique works with processes that are at only one operating point, it fails when the operating point moves. With the truck-speed limiter, the operating point moves because of different load situations such as driving uphill or downhill, as well as driving empty or with a full load. Additionally, the characteristics of the pneumatic valve and the fuel

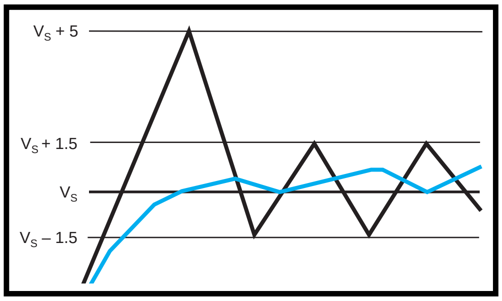


Figure 1—*By law, the speed limiter is allowed a +5-km/h overshoot when reaching maximum speed. Thereafter, it must control the speed within a ±1.5-km/h band. The blue line shows that the fuzzy-logic controller provides much smoother performance compared to the conventional controller (black line). The overshoot is eliminated, and the fluctuations are only about one-fourth of the allowed maximum.*

injection are highly nonlinear and vary from one truck to another.

If a PID control algorithm is used in a truck-speed limiter, it can only be tuned accurately for one operation point and one type of truck. For other operation points and different truck types, overshoot and hunting occur.

To compensate for this, European legislation permits speed limiters to operate within a certain tolerance, as shown in Figure 1. Once the maximum speed is reached, an initial overshoot of 5 km/h is tolerated. Afterward, the speed is kept constant within an interval of ±1.5 km/h. But, the overshoot and hunting tolerated by the legislation result in annoying speed fluctuations.

MECHANICAL DESIGN

Figure 2 shows the outline of the mechanical design for the speed limiter. An electronic control unit (ECU) compares the digital pulse signal from the speedometer with the maximum speed value preset in the device.

Based on this comparison, the ECU computes the command value for the pulse-proportional valve (PPV) that controls the air pressure in the cylinder. The air stems from the vehicle's pressured-air system. In a nonlinear but proportional ratio, the cylinder shortens the arm linking the accelerator pedal to the fuel pump, thereby throttling the fuel pump.

The ECU is designed as a mixed digital and analog circuit. Speedometer signal processing, diagnosis functions, and the fuzzy-logic control algorithm are all computed by an 8-bit PIC. The MCU uses an external EEPROM to store parameters of the truck and speedometer such as the maximum velocity and diagnosis variables.

The MCU generates a PWM signal that is amplified by a power stage to drive the PPV. The analog part is responsible for preprocessing and filtering the speedometer signal.

FUZZY-LOGIC CONTROLLER

Fuzzy logic is an innovative technology for solving multiparameter and nonlinear control problems. It uses human experience and experimental results rather than a mathematical model to define a control strategy.

As a result, fuzzy logic often delivers solutions faster than conventional control techniques. As well, fuzzy-logic implementations on microcontrollers are very efficient when it comes to code space and execution speed [1, 2].

The entire fuzzy-logic algorithm was developed, tested, and optimized using Inform's *fuzzy*TECH software tool. This integrated design environment features automatic assembly code generation on all PIC families [3, 4].

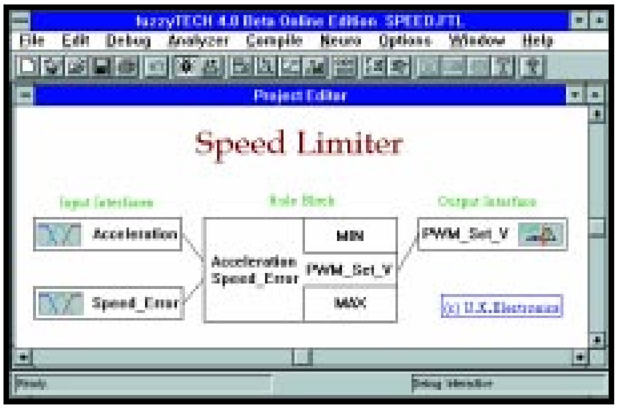


Photo 1—*The fuzzy-logic controller for the speed limiter uses acceleration and speed error as inputs to determine the set value for the pressure valve.*

Photo 1 shows the Project Editor featuring the structure of the fuzzy-logic system. On the left side, two input interfaces fuzzify the two input variables `Acceleration` and `Speed_Error`.

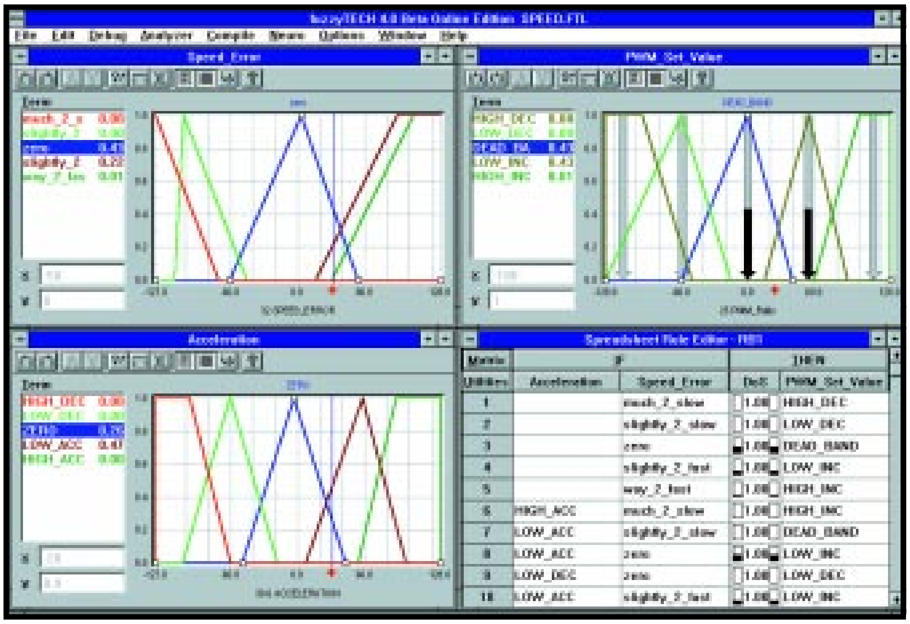
The rule block in the middle contains all the fuzzy-logic rules that represent the system's control strategy. On the right side, the output variable `PMV_Set_Value` is defuzzified in an output interface.

The linguistic variables are displayed in a variable-editor window, and the rules are shown in the Spreadsheet Rule Editor window (see Photo 2). Each linguistic variable contains five terms and membership functions (standard type) that are connected by a total of 12 fuzzy-logic rules. As a defuzzification method, the Center-of-Maximum (CoM) method is used [1].

All rules in the fuzzy-logic system let the designer define the best reaction (output variable value) for a situation. The situations are described by the combination of the input variables.

A number of different analyzer tools are used to verify the system's performance. In the 3-D plot in Photo 3, the two horizontal axes show the two input variables, `Acceleration` and `Speed_Error`. The vertical axis plots the output variable (`PWM_Set_Value`), which is the set value for the PWM unit on the microcontroller.

Rule 1, as shown in Photo 2, states that if `Speed_Error = much_2_slow`,



Matrix	IF		THEN	
Utilities	Acceleration	Speed_Error	DoS	PWM_Set_Value
1		much_2_slow	1.00	HIGH_DEC
2		slightly_2_slow	1.00	LOW_DEC
3		zero	1.00	DEAD_BAND
4		slightly_2_fast	1.00	LOW_INC
5		way_2_fast	1.00	HIGH_INC
6	HIGH_ACC	much_2_slow	1.00	HIGH_INC
7	LOW_ACC	slightly_2_slow	1.00	DEAD_BAND
8	LOW_ACC	zero	1.00	LOW_INC
9	LOW_DEC	zero	1.00	LOW_DEC
10	LOW_ACC	slightly_2_fast	1.00	LOW_INC

Photo 2—*The fuzzy-logic system is designed graphically using the* fuzzy*TECH software development system. Linguistic membership functions are drawn with the mouse, and control rules are represented as tables.*

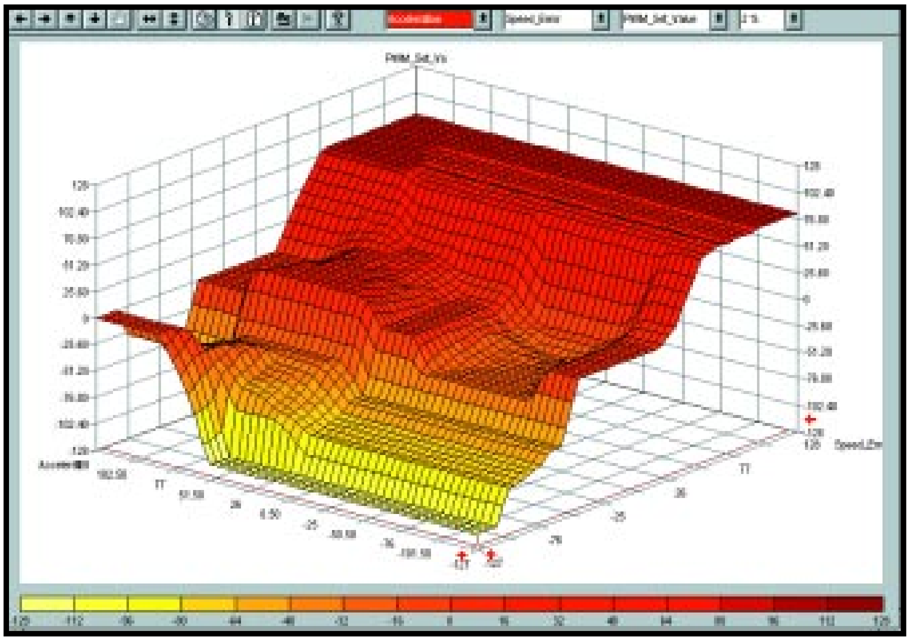


Photo 3—*The 3-D analyzer of* fuzzy*TECH plots the transfer characteristics of the fuzzy-logic controller. This type of controller approximates the nonlinear characteristics of the truck much better than a linear PID-type controller.*

then PWM_Set_Value = HIGH_DEC. This rule represents the engineering knowledge that if the truck is under the speed limit, no pressure should be applied to the cylinder. The membership function of the term much_2_slow is also shown in the respective variable editor in Photo 3.

The 3-D analyzer plots the transfer characteristic as a result of rule 1. In the front part of the curve, the value of the output variable is very low (color of surface light). As you proceed to the left along the Acceleration axis, the output variable value increases, which is a result of rule 6.

Rule 6 states that if Acceleration = HIGH_ACC and Speed_Error = much_2_slow, then PWM_Set_Value = HIGH_INC. This rule represents the engineering knowledge that in the case of a high acceleration, the result is medium pressure on the cylinder. This action ensures that the cylinder already contains some pressure in case the truck reaches the limit quickly. Without this rule, a speed overshoot would occur.

IMPLEMENTATION

Because *fuzzy*TECH can simulate the fuzzy-logic system without the target hardware, a fair amount of optimization was accomplished off-line on the PC. Final optimization and verification of the system were conducted on real trucks.

The target system and the MCU for the electronic control unit is mounted in the truck and connected to the development PC (a laptop in the truck's cabin) by a serial cable. Serial connection enables modification of the running fuzzy-logic controller on-the-fly.

This development technique is efficient because it lets the developer analyze how a certain behavior of the fuzzy-logic controller is caused by the membership-function definition and the rules. And because modifications can be done in real time, the effects can be felt on the truck instantly.

One way to enable on-the-fly debugging is to link the *fuzzy*TECH real-time remote cross-debugger (RTRCD) module to the fuzzy-logic controller that runs on the MCU and connect it to a serial driver. The RTRCD module consumes about half a kilobyte of ROM, a few bytes of RAM, and some computing resources to serve the serial communication on the MCU.

Because the PIC used in this application can't provide such resources, the serial debug mode of *fuzzy*TECH is used rather than the RTRCD module. In serial debug mode, the values of the input variables are sent from

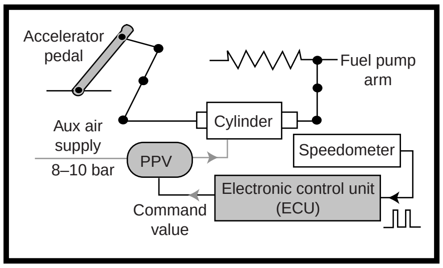


Figure 2—*The speed limiter consists of an electronic control unit (ECU) that reads in the speedometer signal and controls the pulse proportional valve (PPV). The PPV manipulates the air pressure in the cylinder connecting the accelerator pedal and fuel-pump arm.*

the MCU to *fuzzy*TECH running on the development PC via the serial cable and the results are sent back.

*fuzzy*TECH shows the entire fuzzy-logic computation in its editor and analyzer windows and enables on-the-fly modifications. To support serial communication, a piggyback board containing a MAX232 driver IC is mounted on the speed-limiter board.

Using *fuzzy*TECH's serial debug mode rather than the RTRCD module means that the code size and computation effort previously required for the fuzzy-logic computation on the MCU can be saved and used for the serial communication.

The disadvantage of the serial debug mode is that it computes the system's results on the PC, where real-time response cannot be guaranteed. Also, unlike the RTRCD module, any crash of Windows, the PC, or the serial communication will halt computation in serial debug mode.

RESULTS

After optimizing the fuzzy-logic rule strategy on different trucks and various load conditions, the speed limiter demonstrated the response curve shown in Figure 1 (blue line). The fuzzy-logic controller achieves a much smoother response, doesn't show overshoot behavior, and provides a higher accuracy of keeping the speed limit compared to a conventional controller.

The final fuzzy-logic system was compiled to PIC assembly code by *fuzzy*TECH and required 417 words of ROM space and 32 bytes of RAM. The RAM space can be used for other computation tasks such as preprocessing and filtering while the fuzzy-logic system isn't running. The entire fuzzy-logic system needs less than 2 ms to compute on the PIC16 MCU.

I've demonstrated that, with a little bit of fuzzy logic, you can solve difficult control problems using conventional design techniques and by putting your own engineering experience to work.

Not only that, but you can design a solution using visual software tools and generate highly optimized assembly code for most microcontrollers at the push of a button. Now you're on the way to significantly reducing design time. ■

Constantin von Altrock began research on fuzzy logic with Hewlett-Packard in 1984. In 1989, he founded and still manages the Fuzzy Technologies Division of Inform Software Corp., a market leader in fuzzy-logic development tools and turn-key applications. You may reach him at cva@inform-ac.com.

REFERENCES

[1] C. von Altrock, *Fuzzy Logic and NeuroFuzzy Applications Explained*, Prentice-Hall, Englewood Cliffs, NJ, 1995.

[2] H.J. Zimmermann, *Fuzzy Set Theory—and its Applications*, Boston, MA, 1991.

[3] Inform Software Corp., fuzzy-*TECH 5.2 Edition Manual*, 1998.

[4] Microchip Technology, fuzzy-*TECH* fuzzy*LAB Manual*, 1998.

SOURCES

PIC microcontroller
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

***fuzzy*TECH**
Inform Software Corp.
(630) 268-7550
Fax: (630) 268-7554
www.fuzzytech.com

MAX232
Maxim Integrated Products
(800) 998-8800
(408) 737-7600
Fax: (408) 737-7194
www.maxim-ic.com

FEATURE ARTICLE

Fuzzy Footfalls

Constantin von Altrock

Pressure Sensing in a Medical Shoe

If you've ever had knee surgery, you know how long it takes to recover. Imagine a pressure sensor that enables you to monitor the pressure you're putting on a bad knee. Fuzzy logic and a tension sensor make it all possible.

Analyzing complex data sets often requires human-like evaluations or decisions that are severely limited if mathematical models are used. In many applications, the data material is noisy or contains artifacts. Under such conditions, it doesn't take long for conventional data-analysis methods to reach their limits.

Most data-analysis methods derive structural information from given data sets. This structural information represents the system that produced the data sets. The goal is to identify internal parameters of the system that can't be directly measured.

There are many different methods and algorithms for data analysis, but most have difficulties coping with noisy data and data containing artifacts. These applications have to use very robust data-analysis techniques that cope with the errors and artifacts.

Take a look at Figure 1 to see how fuzzy logic can help. To the human eye, this graphic looks like a nonsensical collection of gray squares. The human eye is a precise sensor that can distinguish about 100 shades of gray, but even so, Figure 1 remains a collection of gray squares. Only by squeezing your eyelids so tight that the entire picture becomes fuzzy can you see that the collection of gray squares is actually a picture of Abraham Lincoln.

The lesson? Even the most precise methods can't reveal the picture. If you modify the grayness of some of the squares, the most accurate picture still comes from the fuzzy look.

Employing fuzzy logic can be just as successful in data-analysis applications to provide solutions for everyday problems. For example, after knee surgery the doctor tells you to limit the strain on your knee. But what constitutes too much strain? And how do you know when you're approaching the limit?

The solution is really quite fuzzy, but I'll get to that in a moment. Using fuzzy logic for data analysis, whether for medical or practical purposes, can be accomplished with many different combinations of fuzzy logic and conventional techniques.

Of course, choosing the best combination of techniques depends on the application. The most common combinations are fuzzy cluster analysis, fuzzy rule-based methods, and adaptive fuzzy rule-based methods. Let's review these before getting into the application.

CLUSTER ANALYSIS

Fuzzy cluster analysis maps objects to predefined classes [1, 2, 3, 4]. In a quality-control system, for example, the classes could be either good or bad. For this mapping, a vector of parameters describes each object, and each parameter denotes a certain property of the objects.

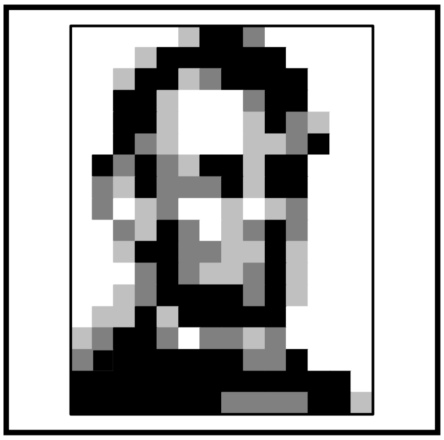
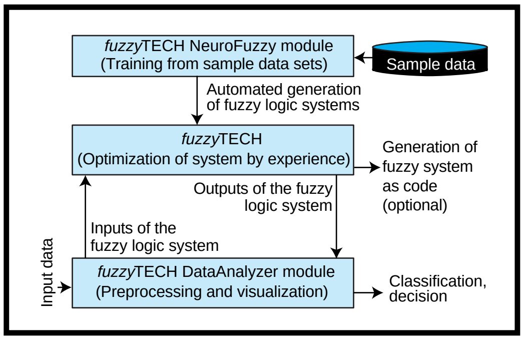


Figure 1—*Only by squeezing your eyelids so the picture becomes "fuzzy" can you recognize Abraham Lincoln.*



fuzzyTECH NeuroFuzzy module (Training from sample data sets)
Sample data
Automated generation of fuzzy logic systems
fuzzyTECH (Optimization of system by experience)
Generation of fuzzy system as code (optional)
Outputs of the fuzzy logic system
Inputs of the fuzzy logic system
Input data
fuzzyTECH DataAnalyzer module (Preprocessing and visualization)
Classification, decision

Figure 2—*Linking the DataAnalyzer module with the NeuroFuzzy module and fuzzyTECH creates an integrated development environment for fuzzy data-analysis systems.*

These parameters for classifying acoustic signals can be the result of a Fourier transform. Most cluster-analysis algorithms use training algorithms to configure themselves from given sample data sets.

Using fuzzy logic in cluster analysis enables fuzzy classes to be defined. So, even when a unique classification of some parameters isn't possible, a good final solution can still be derived.

Cluster analysis derives all necessary structural information by training from given sample data sets. This situation demands very high-quality sample data sets. Also, there are no explicit modifications of the resulting system, so optimization and verification are difficult tasks.

RULE-BASED METHODS

With fuzzy rule-based methods, if-then rules represent the entire classification. This method is similar to the application of fuzzy logic in intelligent control. But in contrast to fuzzy logic in control applications, fuzzy-logic classification uses different inference and defuzzification methods.

The benefit of rule-based methods over cluster analysis is that the information flow in the system is completely transparent. Because fuzzy-logic systems are self-explanatory, explicit optimization and verification are easy.

But rule-based methods have a disadvantage as well. The entire system has to be built up manually, and unlike cluster analysis, no automated training exists. Despite these problems, however, fuzzy rule-based methods are the basis of many successful applications for data analysis and signal classifications [5].

ADAPTIVE RULE-BASED METHODS

The advantage of cluster analysis lies in its trainability, and the advantage of rule-based methods lies in the inherent transparency of the system. But, some applications need trainability and transparency at the same time.

Combining a training algorithm with a fuzzy rule-based method makes for a successful solution. Because the training algorithm adapts the fuzzy rules and membership functions, and the behavior represents the sample data sets, this combination is called the adaptive fuzzy rule-based method.

Although there are many ways to adapt a fuzzy-logic system, the most widely used approach in industrial applications is the neurofuzzy technique. With this technique, learning algorithms developed for neural nets are modified so that they can also train a fuzzy-logic system [5].

This method learns from given sample data sets, and the learned result can be further enhanced by hand. For applications where only partial information for the solution stems from sample data, adaptive fuzzy rule-based systems are best. Another advantage of this method is a pure fuzzy-logic system that can be implemented even on inexpensive hardware platforms. You'll see more on this later.

SOFTWARE TOOLS

In most fuzzy data-analysis systems, the input data needs extensive preprocessing before reaching the fuzzy-logic system. Preprocessing can include filtering, linearizations, or Fourier transforms. These functions aren't part of most fuzzy-logic applications in industrial control and aren't usually part of fuzzy-logic software development tools.

For the *fuzzy*TECH development system [6], data-analysis functionality is provided by an add-on tool called the DataAnalyzer module [7]. Figure 2 shows how *fuzzy*TECH and the DataAnalyzer and NeuroFuzzy modules can be linked to form an integrated design environment.

Linking these components enables the design of adaptive fuzzy rule-based solutions. If the NeuroFuzzy module is left out, only fuzzy rule-based solutions are possible.

Photo 1 shows a fuzzy data-analyzer solution that supervises the wear of a machinery tool during operation. The two upper-left function blocks drive A/D channels on a standard PC plug-in board. The upper channel links to a pressure-stripe sensor that acts as a microphone. The lower one links to a temperature sensor mounted at the tool.

The acoustic signal is preprocessed by a spectrum block (i.e., FFT) and input to the fuzzy-logic function block. The second input is the temperature signal filtered by a low-pass filter.

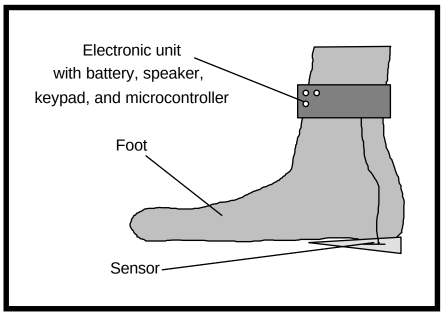
The third input is the direct temperature signal after a threshold function block. And the fourth comes from a visual inspection and is input by a slide in a separate window.

Meter and spectrum scopes display the outputs of the fuzzy-logic function block. If an overload occurs, a D/A channel sends the machine a speed override signal to avoid destroying the tool.

A file function block writes the evaluation result on disk. The NeuroFuzzy module sits on top of *fuzzy*TECH (see Figure 2), which is why you don't see it in Photo 1.

FUZZY SHOE

Now that you understand fuzzy-logic methods, let's look at the knee-surgery recovery problem I mentioned. Patients have to limit the strain on the knee during recovery. But, the knee has no



Electronic unit with battery, speaker, keypad, and microcontroller
Foot
Sensor

Figure 3—*The orthopedic shoe consists of an electronic unit attached by Velcro over the ankle and a sensor in a silicone inlay.*

strain sensor. By the time pain occurs, the knee has already suffered damage.

To solve this problem, I used a pressure sensor and fuzzy-logic data-analysis system to design a biofeedback shoe inlay. The silicone inlay contains a pressure sensor made of conducting polymer and is wired to an electronic unit in a belt that attaches to the ankle by Velcro (see Figure 3).

The electronic unit contains the microcontroller, battery, speaker, and keypad. The speaker warns the user when the strain limit is reached, and the 8-bit micro runs the A/D conversion, signal preprocessing, and fuzzy data-analysis system.

The objective for the fuzzy data-analysis system is to estimate the internal strain in the knee from the pressure signal. If 80% of the maximum acceptable load is reached, a beep warns the user to take it easy.

If the 90% level is reached, a repeating beep tells the user not to use the leg for a while. And, if the strain is over the maximum threshold set by the doctor, the beep sounds continuously.

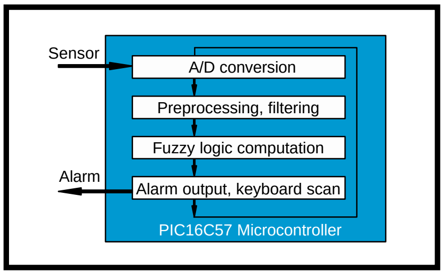


Figure 4—*The complete control loop, reading in sensor data, preprocessing, fuzzy-logic computation, and data output, is all implemented on one microcontroller.*

The difficulty in this situation is estimating the internal strain in the knee from just the pressure signal. Photo 2 displays the pressure sensor signal for a typical sequence of steps. In order to get more information from this signal, preprocessing derives additional inputs to the fuzzy data-analysis system.

In total, there are four inputs to the fuzzy-logic function block:

- Act_Peak—peak pressure of the current step
- Act_Slope—slope of the pressure signal of the current step
- Hist_Short—feedback of the average output signal from the fuzzy-logic function block for the last 5 min. and an indicator of the current strain on the knee
- Hist_Long—feedback of the average output signal of the fuzzy-logic function block of the last 48 h and an indicator of long-term strain on the knee

Photo 3 shows the fuzzy-logic system in the data analyzer, with the upper window showing the system structure. The output variable Alarm stems from a rule block with the input variables ActualLoad and TimeLoad, which are outputs of other rule blocks.

ActualLoad computes from the two input variables (Act_Slope and Act_Peak) that are input variables of the fuzzy-logic function block. The data-analyzer module computes these variables from the pressure sensor signal.

TimeLoad computes from the inputs Hist_Short and Hist_Long. The data-analyzer module computes these inputs using low-pass filtering from the system's output signal.

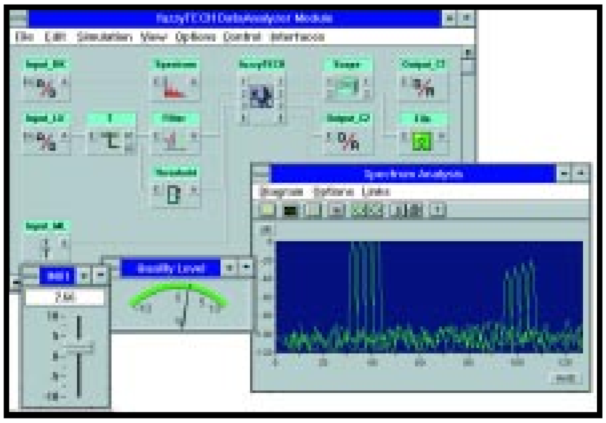

fuzzyTECH DataAnalyzer Module
File Edit Simulation View Options Control Interfaces
Spectrum Analysis
Quality Level

Photo 1—*Developing a fuzzy data-analyzer solution is completely graphical. Configuring predefined function blocks integrates conventional signal-analysis techniques and fuzzy logic. Fuzzy logic is also represented as a function block.*

The lower-left window shows a membership-function definition. Most linguistic variables use two or three membership functions of standard MBF type. The lower-right window shows four rules of the upper rule block. The 39 rules in the system were derived in close cooperation with orthopedists.

Because no good sample data for the strain estimation in a knee exists, the NeuroFuzzy module wasn't used. Under the nomenclature I discussed earlier, this application uses fuzzy rule-based methods.

Using fuzzy logic here means that the rule set is easy to modify. As an example, by designing a rule set that evaluates steps and their fit to an optimal curve, runners can improve their running style with this intelligent biofeedback technique.

Figure 4 shows the total implementation on a PIC16C57 microcontroller. The ADC transforms the resistance of the pressure sensor into a digital 10-bit value. Preprocessing and filtering get the four input variables for the fuzzy-logic computation shown in Photo 3.

Depending on the strain rating, the speaker outputs the alarm and the keypad is scanned. The fuzzy-logic system requires ~20 bytes of RAM and 590 words of ROM on the PIC16C57 [8]. The code was generated by the *fuzzy*-TECH Edition that generates fuzzy-logic systems as assembly code for standard microcontrollers.

This case study demonstrates how you can use innovative software design techniques like fuzzy logic to implement even complex functions within the limits of a low-cost microcontroller.

With graphical software development tools, you can focus on solving your problem. Turning your solution to highly optimized assembly code for microcontrollers becomes a matter of a mouse click.

The fuzzy shoe is a good example of how intelligent software design techniques lead to innovative products.

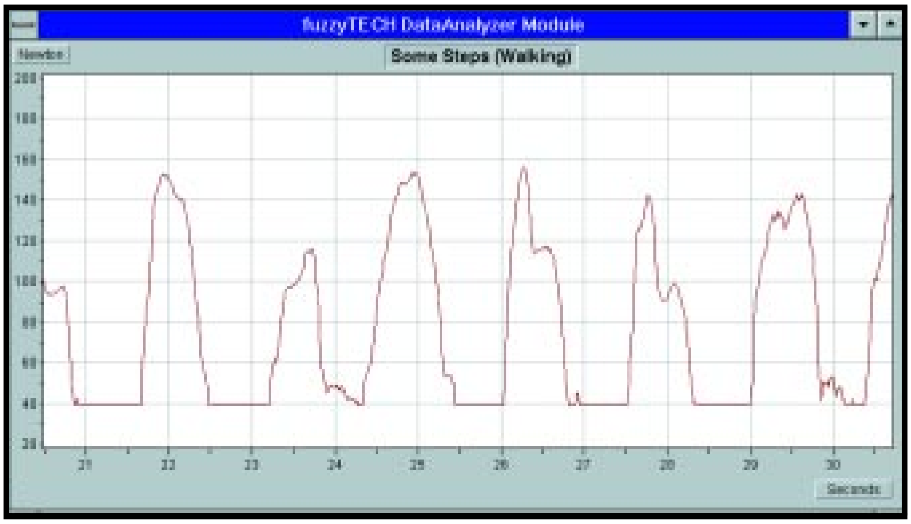


Photo 2—*A walking sequence shows the amount of pressure placed on the sensor with each step.*

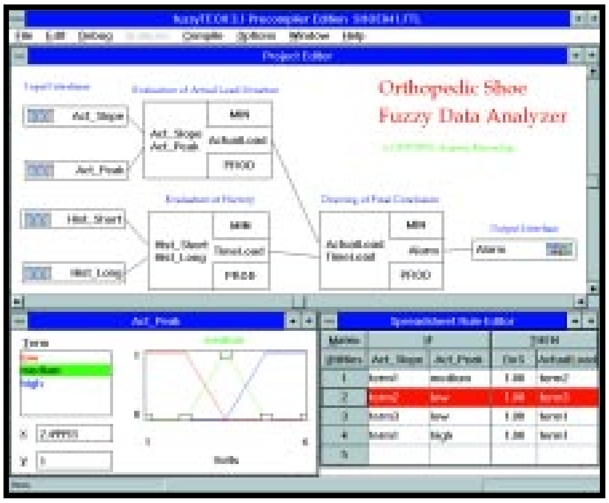


Photo 3—*Here's the structure of the fuzzy-logic function block for the fuzzy data-analysis system. The four inputs are compared to the set rules, and the appropriate response is generated.*

Just imagine shoes and other devices that enable you to improve your exercise and avoid damaging your body! ◼

Constantin von Altrock began research on fuzzy logic with Hewlett-Packard in 1984. In 1989, he founded and still manages the Fuzzy Technologies Division of Inform Software Corp., a market leader in fuzzy-logic development tools and turn-key applications. You may reach him at cva@inform-ac.com.

REFERENCES

[1] J. Bezdek, E. C-K. Tsao, and N. Pal. "Fuzzy Kohonen Clustering Networks," FUZZ-IEEE Conference, 1992, 1035–1043.

[2] A. Kandel, *Fuzzy Techniques in Pattern Recognition*, New York, NY, 1982.

[3] V. Watada, "Methods for Fuzzy Classification," *Japanese Journal of Fuzzy Theory and Systems*, **4**, 149–163, 1992.

[4] H.J. Zimmermann, *Fuzzy Set Theory—and its Applications*, Boston, MA, 1991.

[5] C. von Altrock, *Fuzzy Logic and NeuroFuzzy Applications Explained*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[6] Microchip Technoloy, *fuzzyTECH MP Edition Manual*, 1994.

[7] Inform Software Corp., *fuzzyTECH 5.2 DataAnalyzer Module Manual*, 1998.

[8] Inform Software Corp., *Fuzzy Logic Benchmarks for Standard MCUs*, www.fuzzytech.com, 1994.

RESOURCES

C. von Altrock, B. Krause, and H.J. Zimmermann, "Advanced fuzzy logic control of a model car in extreme situations," *Fuzzy Sets and Systems*, **48:1**, 41–52, 1992.

C. von Altrock and B. Krause, "On-Line-Development Tools for Fuzzy Knowledge-Base Systems of Higher Order," Proceedings of the 2nd Int'l Conference on Fuzzy Logic and Neural Networks, 1992.

C. von Altrock, B. Krause, and H.J. Zimmermann, "Advanced Fuzzy Logic Control Technologies in Automotive Applications," IEEE Conference on Fuzzy Systems, 831–842, 1992.

C. von Altrock, "NeuroFuzzy Technologies," *Computer Design Magazine*, June 1994.

C. von Altrock, H.O. Arend, B. Krause, C. Steffens, and E. Behrens-Rommler, "Customer-Adaptive Fuzzy Control of Home Heating System," IEEE Conference on Fuzzy Systems, 1994.

SOURCES

PIC16C57
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

***fuzzy*TECH**
Inform Software Corp.
(630) 268-7550
Fax: (630) 268-7554
www.fuzzytech.com

C++ Lite

FEATURE ARTICLE

John Carbone

A C++ Dialect for Embedded Systems

Want to be able to maintain valuable C++ concepts while eliminating those responsible for boosting memory requirements and reducing efficiency? John shows us how with EC++, a new dialect that targets embedded systems.

In many embedded system designs, programmers can leverage all the features of C++, such as classes, templates, exception handling, and class inheritance, that have proven so useful for mainstream computer applications. But, some of these features bloat the compiled code, significantly increasing memory requirements while reducing execution speed.

A new C++ dialect, Embedded C++ (EC++), has been developed by an industry-standards committee to address the limitations of C++ in embedded applications, where memory is limited and 32-bit processors are prevalent.

EC++ maintains the most valuable C++ concepts while eliminating those most responsible for boosting memory requirements and reducing efficiency. Ideally, designers will be able to choose whether to use EC++, C++, or a hybrid of the two to match their application requirements.

The object-oriented features of C++ simplify source code and the development process by reusing code modules and placing burdensome housekeeping functions (e.g., memory allocation and range checking) in the class definitions and away from the main application.

Although C++ code is typically more readable than standard C code, compiled C++ code can swell by a factor of five or more relative to a standard C implementation. This increase caused a group of companies, led primarily by Japanese microprocessor vendors, to develop the EC++ specification and to prompt the development of the first EC++ compiler.

EC++ is a proper subset of C++. Among the C++ features it omits are multiple inheritance, virtual base classes, templates, exceptions, runtime type identification, virtual function tables, and mutable specifiers (see sidebar "EC++ Excludes Problematic Features" for details).

Although each of these features is useful in its own right, none is compelling enough for a sufficiently broad range of embedded applications. Support for some of the features tends to bloat the generated code, whether or not the features are used in an application.

Exception handling proves to be one of the worst offenders and can adversely affect the deterministic response to external events required in real-time systems. Eliminating support for a number of C++ features substantially reduces the size of the compiled code and improves run-time efficiency.

To understand the range of features that differentiate standard C, EC++, and C++, consider two examples of EC++ and C++ constructs. Listing 1 illustrates some of the key advantages that EC++ offers over straight C.

The concept of classes is probably the single most important concept brought about by C++ and one that is supported in EC++. Classes build on the data structures found in standard C.

Besides allocating memory for a number of variables of mixed types, classes can initialize variables, dynamically allocate additional memory for variables and arrays, perform range checking, and perform other useful duties. In C programs, these tasks were typically scattered throughout the main code.

CLASSES AND OBJECT DEFINITION

An embedded application such as a data-acquisition system might use such a class to create arrays for storing data

Listing 1—*This embedded C++ example of a simple `Array` class does range checking on the creation of an array and on array subscripting operations.*

```
#include <iostream>
using namespace std;
extern "C" void exit(int);
    // Deal with run-time error. A real embedded application would
    // probably choose a different error-handling strategy.

void die(const char *msg, int n)
{
  cout << msg << n << endl;
  exit(1);
}

class Array{                          // the integer array class
  private:
  int *elements;                      // array elements
  int element_cnt;                    // array size
  public:
  Array(int n) : element_cnt(n) {     // construct new array
    if (n > 0)
      elements = new int[element_cnt];
    else
      die("Bad Array size ", element_cnt);
  }
  int & operator [](int indx) const {
    // overloaded subscripting operator
    if (indx < 0 || indx >= element_cnt)
      die("Bad Array index ", indx);
  return elements[indx];
  }
  int size() { return element_cnt; }  // return size of array
};

main()
{
    Array a(6);
    for (int i=0; i<a.size(); i++)
    a[i] = i;
    for (int i=0; i<a.size()+1; i++)  // error on a[7]
    cout << i << ". " << a[i] << endl;
};
```

Listing 2—*Templates are a powerful feature but should be used carefully in space-sensitive applications.*

```
template <class T>
class Array{
  private:
    T *elements;                      // array elements
    int element_cnt;                  // array size
  public:
    Array(int n) : element_cnt(n){    // construct new array
      if (n > 0)
        elements = new T[element_cnt];
      else
        die("Bad Array size ", element_cnt);
    }
    T & operator [](int indx) const{
    // overloaded subscripting operator
      if (indx < 0 || indx >= element_cnt)
        die("Bad Array index ", indx);
      return elements[indx];
    }
    int size() { return element_cnt; // return size of array
};
```

samples. The array class named `Array` in Listing 1 includes two integer members. The first is an `elements` pointer to members of an array, and the second tracks the array size.

As the `main()` portion of this code implies, you can use the declaration `Array a(6)` to create an array object with the name `a` that contains six elements. The class definition includes several important features such as the constructor code necessary to create the array and to ensure that the size specified is greater than zero.

The constructor is located in the public section of the class definition. It provides a window through which code located outside the class definition can access the elements and `element_cnt` class members.

Each time an array of type `Array` is created, the compiler automatically calls the constructor function `Array (int n)`. This function assigns the value passed in the array declaration to `element_cnt`, checks for a valid size, and allocates space in main memory for the array by calling `new`.

In this simple example, a bad array size (e.g., zero or a negative number) causes the constructor to call a simple `die` function that outputs an error message. An embedded system would probably use a more elaborate scheme to handle run-time errors.

The `Array` class also demonstrates two other key features of classes in EC++ or C++—function definitions within a class and overloaded operators. First, consider the `size()` function, which illustrates the simpler of the two concepts.

Because `element_cnt` is a private member of the class, code outside the class can't directly access the counter. However, the `size()` function enables the two `for` loops located in `main()` to indirectly access `element_cnt` for use as an upper limit of the loop.

OVERLOADED OPERATORS

The class definition also includes an example of overloaded operators. Operator overloading enables you to develop new definitions of standard C/C++ operators such as =, >, or + that are customized for the type of object defined in a class.

For example, you could develop a class that defined an object like a circle or sphere. Having done so, you might want to compare the size of two objects (A and B) that were created using the class definition.

The expressions A>B or A==B have understood meanings when A and B are integers, but the compiler won't know how to compare abstract entities such as spheres of a given size or composition. To make it possible to operate on complex user-defined types while using a readable C-like syntax, C++ and EC++ allow operator overloading.

The code in Listing 1 overloads the subscripting operator [] that stores the index for an array. In this case, the overloaded function gets called each time an indexed array reference occurs (e.g., a[i]=i).

Rather than changing the effective meaning of the subscripting operator, the overloaded operator automatically detects out-of-range array indices. Note that, should you execute the sample program, the output statement used in the second loop would generate an error on the sixth pass through the loop because a[7] exceeds the valid index test.

Listing 3—*This more capable main program uses templates to enable multiple uses for the `Array` class.*

```
main()
{
  Array<int> a1(6);
  for (int i=0; i<a1.size(); i++)
    a1[i] = i;
  for (int i=0; i<a1.size()+1; i++)   // error on a[7]
    cout << i << ". " << a1[i] << endl;
}
```

As you can see, classes significantly streamline the mainline code in an EC++ or C++ program. A C program requires explicit data-structure definitions for every array declared, while EC++ or C++ handles creation of all similar objects with a single class.

Also, C programs require memory-allocation, error-checking, and element-count code in the main part of the program or in dedicated C functions. The compiled-code overhead of EC++, relative to standard C code, is minimal as well. Adding classes and overloaded operators only marginally increases the generated code size.

Understand that you can't realize these code savings simply by avoiding the memory-hungry C++ features in an application and then compiling the code with a standard C++ compiler. The consequences of this approach are detailed in the sidebar "Why You Need an EC++ Compiler."

ADDING TEMPLATE SUPPORT

Despite the advantages offered by EC++, some omitted C++ functions are extremely attractive for certain applications. Compiler vendors can provide programmers some flexibility as to what C++ functions are available for use in each application.

For example, you can use a compiler directive with Green Hills' C++/EC++ compiler to limit the source code to the EC++ subset and get the savings in code size and the boost in efficiency. You can also use compiler switches to add support for one or a few specific C++ functions that are left out of EC++.

The granular support for optional C++ features enables programmers to trade off compiled code size with ease of development and maintainability. Green Hills' C++/EC++ enables the programmer to choose libraries appropriate to the application, eliminating significant amounts of unnecessary library code.

Templates are a good example of a C++ feature that isn't included in EC++ but that provides significant advantages in development with only a modest increase in memory requirements. Listings 2 and 3 illustrate the benefits.

Why You Need an EC++ Compiler

Because EC++ is a proper subset of C++, you might think you can achieve the efficiency promised by EC++ by avoiding certain C++ features. You can compile EC++ code on a C++ compiler, but you'll find that the resulting code still requires significantly more memory than if you'd used an EC++ compiler.

Three problems arise when you try to increase efficiency using EC++ code and a standard C++ compiler. First, the compiler still uses C++ libraries and links a lot of code that isn't required for your application into the finished product. EC++ uses libraries that are optimized for the new dialect and generates smaller, more efficient code.

As well, EC++ compilers can achieve superior optimization relative to C++ compilers. They optimize code without presuming that complex features like exception handling may be used.

And lastly, standard C++ compilers have no way to enforce EC++ compliance within a programming team. With a standard C++ compiler, one out of many programmers on a team can use an offending feature and spoil the efficiency of the entire code base.

Green Hills Software developed a sample EC++ program to demonstrate memory efficiency. The program solves a form of the classic traveling-salesman problem that's often used to teach programming. When compiled on the Green Hills C++/EC++ compiler using EC++ mode, the total code size is 57 KB. When the same source file is compiled in C++ mode with no exception-handling library, the code size is 322 KB. Adding an exception-handling library increases the code size to 378 KB.

In Listing 1, Array is defined so that all members have to be integers. With EC++, you need to define additional classes to handle arrays for short, long, char, or floating-point data types. Templates permit a single class definition to support creation of arrays for any valid C++ data type.

Listing 2 shows that the only real addition to the Array class definition with a template is the template construct that precedes the class definition and the use of the T specifier each time the code addresses an element of the array. Consider Listing 3 and you'll see that Array works equally well to instantiate any type of an array.

The example shows an array, a1, that stores integer data types. Simply changing the int to short permits support for the short data types. The main() code segment can reuse the Array class to define multiple arrays with any data type. An embedded system performing data acquisition, for example, could require floating-point arrays.

Using the templates shown here would result in little or no increase in compiled code size. But, be aware that the code size you realize depends on your application.

Much of the code bloat in C++ code comes not from using a feature like a template but from referencing templates found in large C++ libraries. Reference a standard template and you end up compiling many unnecessary things in the library.

COMPILER SUPPORT

EC++ gives embedded-system programmers a valuable path to leverage the most significant aspects of an object-oriented language. With formal approval of the EC++ standard imminent, programmers should demand C++ compilers with EC++ support.

However, to minimize development time and simplify code maintenance, programmers shouldn't automatically dismiss all the C++ features eliminated in EC++. Experienced C++ programmers will be able to use a number of C++ features in an EC++ environment and not decrease efficiency.

EC++ eliminates features such as templates, namespaces, mutable speci-

EC++ Excludes Problematic Features

Although EC++ retains the most important features of C++ (e.g., classes), it eliminates other features because of inefficient memory, difficulty of use, or lack of popularity. Consider the list of omitted features:

- multiple inheritance and virtual base classes
- new-style casts
- mutable specifiers
- namespaces
- run-time type identification
- exceptions
- templates

Listing 1 illustrates the basics of classes, but multiple inheritance and virtual base classes fall at the complex end of the C++ class concept. Programmers often use a hierarchical structure to define complex classes.

For example, a base class called Shape might define any geometrical shape and have an attribute such as color. Through the concept of inheritance, classes like Circle or Square could be derived from shape with additional attributes such as radius or width.

Both EC++ and C++ enable programmers to build multilevel class hierarchies in a linear fashion. C++ also enables multiple inheritance in which the programmer defines a new class based on two or more peer classes.

A programmer could define one class called Box and a second class called Contents (see Figure i). A new class called Shipment can be derived from Box and Contents. Multiple inheritance can be valuable in a number of applications and is regularly used in graphical desktop environments.

In Microsoft Windows, a useful object class can be derived based on a Window class with display attributes such as size or borders, a Menu class with attributes like menu names and styles, or a Display class with attributes that describe objects displayed in a window. Virtual base classes can be used with multiple inheritance to share a base class that's inherited multiple times in a derivation hierarchy.

Some embedded applications, including embedded-PC applications, can make use of multiple inheritance and virtual base classes, but they're not nearly as useful as in desktop applications. Supporting multiple inheritance in a compiler carries a significant burden and the technique is tricky. That's why it was omitted in EC++.

Explicit Type Conversion

While multiple inheritance may be difficult to use, other C++ features were omitted from EC++ because they are seldom used in any application. Good examples are the dynamic_cast feature that was added as part of the new-style casts, and mutable specifiers.

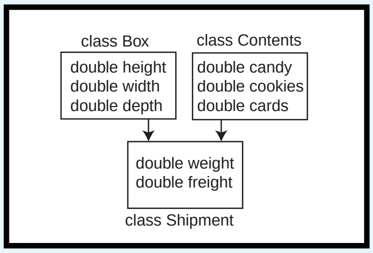


Figure i—*Multiple inheritance allows Shipment to use the properties of both Box and Contents.*

Both C and C++ support the concept of casting to convert from one data type to another. In general, programmers should strive to minimize the need for casting because it often indicates a poorly structured program. EC++ and C++ class structures already reduce the need for casting compared to typical C programs.

Mutable specifiers also represent an arcane feature of C++ that is a special

(continued)

(continued)

case of explicitly type conversion. You can use the keyword `mutable` to cast a data member of a class in a way that it can be modified even though the class is logically constant. Because new-style casts and mutable specifiers are rarely used, they're not justifiable for EC++, especially in light of the code overhead and the complexity of correct usage.

Namespaces and Run-time Type ID

Some other C++ features, including namespaces and run-time type identification (the mechanism used by `dynamic_cast` and other features), are useful in large applications. More specifically, these features are useful in cases with many programmers working on a single code base and in cases where a programmer must interface an application with multiple libraries and code modules from different sources.

However, namespaces are difficult to use correctly, and using them inappropriately adds unnecessary complexity to code. Embedded-system programmers rarely need namespaces and shouldn't have to deal with the associated complexities.

Exception Handling

Unfortunately, not all C++ features omitted from EC++ can be dismissed so easily. Exception handling provides a robust mechanism through which a programmer can centralize and organize code to handle run-time errors or exceptions. Exception handling is also the leading offender when it comes to bloated code.

Typical exception-handling libraries and user code can result in bloated code even when the feature isn't used in a C++ application. Also, programmers can't determine the latencies associated with C++ exception handling, and quick response is paramount to many real-time embedded applications. The feature was omitted in EC++ because, in most cases, programs can't afford the luxury of a general-purpose high-level exception-handling scheme.

fiers, and new-style casts more because of the complexity of using the features properly than because of inherent inefficiency. C programmers may have a tough time using the features correctly, but experienced C++ programmers shouldn't have any problems.

Modern compilers let programmers selectively enable features on an application-by-application basis so they can simplify development and maintain reasonable run-time efficiency. To offer such capabilities, companies should establish a level of C++ support that falls somewhere between EC++ and C++.

This approach, combined with the careful implementation and use of the added features, will provide the best of both worlds. Such products can be considered scalable C++, leaving programmers to decide on the best mix of features for each application they work on. ■

John Carbone is vice president of marketing for Green Hills Software and heads up the marketing of development tools for embedded applications. He has 30 years of experience in embedded software development, project management, as well as sales and marketing of real-time systems. John helped develop the early market for DSP-Array processors with CSPI and SKY Computers. You may reach him at johnc@ghs.com.

SOURCES

EC++ compiler
Green Hills Software
(805) 965-6044
Fax: (805) 965-6343
www.ghs.com

CAD-UL, Inc.
(602) 945-8188
Fax: (602) 945-8177
www.cadul.com

FEATURE ARTICLE

Stuart Ball

Multiprocessor Communications

Part 2: Serial Communication Methods

It's not always true, but there are times when more distance is a good thing, right? (Maybe the tax man or your mother-in-law come to mind….) But when that message still has to get across, Stuart's methods work both locally and over long distances.

Last month, I talked about methods of communicating between multiple processors sharing the same bus or PC board. To finish this two-part series, I look at useful methods for communicating over longer distances. And in case you don't have any long distances to cover, some of these methods are also suitable for local communication.

Plenty of situations require you to have a multiprocessor system with processors that are physically separated from each other. You could have two or more motors that are some distance apart, but maybe you want the controlling processor to be near the motor. There could be redundant components in different rooms or even different buildings.

In one industrial scenario, I had processors sitting side-by-side but in different cabinets. This situation required equipment with multiple cabinets, each containing motors, solenoids, image subsystems, and similar components. Each cabinet contained at least one microprocessor, and the cabinets communicated via a cable interface.

The simplest communication technique is a variation on the register–and–flip-flop method I introduced last time. In Figure 1, CPU 1 transmits data to CPU 2 via a long cable. As I showed you, CPU 1 writes data to a 74*xx*374 (*xx* = LC, HC, ACT, etc.).

The read and write strobes from either CPU are usually too fast to send down a cable of any length. Fast read and write strobes generate a lot of ringing, and there's a potential problem with signal skew. Therefore, you need to match the fast CPU with the slower capability of the cable.

The circuit in Figure 1 accomplishes this by using a 74*xx*374 at the transmit end. Data written by CPU 1 is written to the register, and the act of writing the data sets the D flip-flop. Once the D flip-flop is set, the data available (DAV) signal is sent to CPU 2. At the CPU 2 end of the cable, a 74*xx*244 tristate buffer enables CPU 2 to read the data.

There's a good chance that the data bits will not all arrive at CPU 2 at the same time. Some may even arrive after the DAV signal changes states. But unless CPU 2 is very fast or the cable is very long, everything should settle down by the time CPU 2 reads the data.

When CPU 2 reads the data, the set/reset flip-flop at the CPU 2 end of

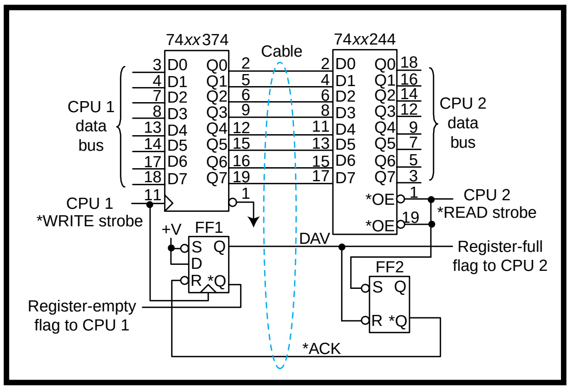


Figure 1—*The register-based technique described in Part 1 can be adapted for communication between different boards using a ribbon cable. The handshake signals are needed to make the timing work.*

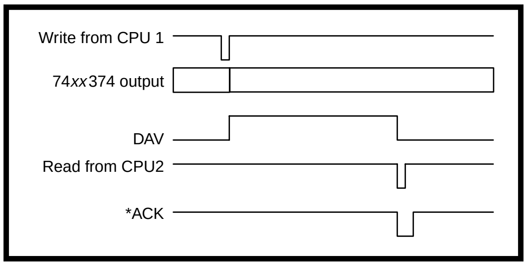


Figure 2—*Here, the timing for the transfer of one byte uses the register-based technique. DAV indicates data available, and ACK acknowledges the transfer.*

the cable is reset, returning the acknowledge (ACK) signal back to CPU 1 and resetting the D flip-flop and DAV signal.

Again, there will be some ringing on these signals, but unless CPU 1 is very fast, everything will settle by the time CPU 1 is ready to write another byte. Figure 2 shows the timing for transmitting one byte via this interface.

The simple circuit in Figure 2 is suitable for short cable runs of less than 20¢ For longer distances, use Schmitt-trigger buffers on all the inputs as well as some kind of cable termination. With additional timing logic, you can transfer data using DMA by this method.

The primary drawback is the number of wires needed. Ideally, you need a ribbon cable with a ground on every other wire, which means 20 wires. You can get by with two or three grounds and a 14-wire cable, but that's still a lot of wires.

SIMPLE SERIAL INTERFACE

Because parallel interfaces require complicated cabling, most multiprocessor systems that must communicate over any significant distance use a serial technique. Figure 3a shows a simple serial interface connecting two 8031 microprocessors. The serial transmit (TxD) line from one processor is connected to the serial input (RxD) line of the other processor, a technique that works for any microcontroller with asynchronous serial I/O.

This technique is similar to connecting two computers together using serial cables. However, it permits only two processors to communicate. Adding more CPUs means adding more serial ports.

MULTIPLE PROCESSOR SERIAL INTERFACE

Figure 3a shows a variation of this technique that supports more than two processors. As you can see, one master 8031 communicates with two slaves. The addition of open-collector drivers for the slave transmit signals makes this possible.

Each slave drives the master receive signal through an open-collector driver. The open-collector drivers can be open-collector (or open-drain) gates or they can be discrete transistors.

When no slaves are transmitting, the receive signal to the master is in the high state (because the marking state for a serial interface is high). When either slave transmits, the common line is driven and the master receives the data.

Transmission to the slaves requires no buffering because the master drives both slave Rx lines from its Tx line and anything the master transmits goes to both slaves. Transmissions from the master must be addressed to one slave device, and slaves must transmit only when commanded by the master.

This process is usually accomplished with a multibyte transmission protocol where each transmission consists of multiple bytes. For example:

- byte 1—destination (slave 0 or slave 1)
- byte 2—length
- byte 3, etc.—data
- last byte—checksum

If the master needs data from a slave, it must request it. The slave cannot transmit data without being commanded to because multiple slaves may try to transmit at the same time.

This next approach can be used for multiple processors on a single board and enables the processors to communicate using two wires and a ground. The processors don't have to be the same type, but they all need to support asynchronous serial communication. Of course, all the serial interfaces must be programmed to operate at the same transfer rate.

RS-422 SERIAL INTERFACE

For longer distances, the circuit in Figure 4a replaces the open-collector connection with an RS-422 connection. RS-422 uses differential voltages and, as shown on the timing diagram in Figure 4a, an RS-422 driver has one input and two outputs.

A high on the driver input causes one of the outputs to go high and the other to go low. A low on the input reverses the two output signals.

Instead of comparing the input voltage to ground, an RS-422 receiver compares the two wires to each other. Because most noise induced on cables is common-mode and affects both wires equally, this differential technique provides excellent noise immunity.

In the open-collector scheme, the transmit line from the slaves back to the master can be driven by either slave because the line is left in the correct state when neither is transmitting. In the RS-422 scheme, the slaves have to turn the drivers on and off to prevent bus contention.

As you see in Figure 4a, another port pin from each slave is required. Probably the most common mistake in using RS-422 in a shared-bus system is getting the timing right. A slave that wants to transmit must turn on the RS-422 driver when it starts to transmit, and turn it off at the end of the transmission but not before.

The process is further complicated by the fact that most microcontroller UARTs have a bit that tells when the transmit buffer is empty, but nothing tells when the last byte has actually been shifted out of the transmit shift register.

To resolve this problem, the slave sets up a timer at

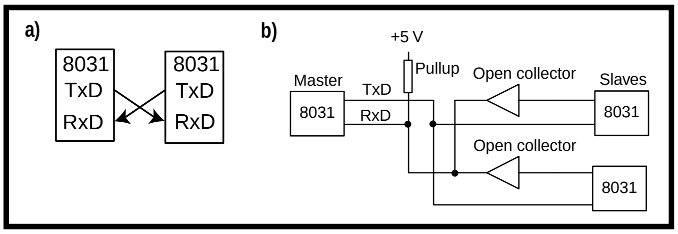


Figure 3a—*Microprocessors that have internal asynchronous serial interfaces may communicate by cross-connecting the receive and transmit signals.* ***b***—*Adding open-collector buffers to the transmit lines enables multiple processors to share a single asynchronous serial line.*

the beginning of the last byte transmitted and turns the driver off when the timer expires. The timer period is the time needed to transmit one byte.

Another solution is for the master to send an acknowledge after every message. It tells the slave to turn off the driver and frees up the common line pair. The pull-up/pull-down resistors on the common receive lines ensure that the lines stay in the correct (high) state while idle.

The RS-422 serial-interface method provides excellent noise immunity, but at a price. For this interface, four wires plus ground are required.

Maximum input to the drivers is +12/–7 V, but circuits with a great distance between devices may have ground potentials differing by more than this. A common ground between devices helps ensure this value isn't exceeded. When this ground connection is used, earth ground is attached at only one end.

RS-422 PARTY LINE

Figure 4b shows a further extension to the RS-422, where both the transmit and receive pairs are combined into one pair of wires. This party-line system enables multiple processors to communicate over a single pair of wires (plus ground).

The RS-422 serial-interface approach, with separate receive and transmit lines, enables data to go both directions at once. With the party-line approach, by contrast, the processors can only communicate one at a time. So, the price to pay for such a simple system is reduced bandwidth.

The second difficulty with the party-line approach is that the timing gets more complex. Every processor, including the master, must turn its transmit buffer on and off. Like the four-wire RS-422 solution, terminating resistors are needed on the common receive line to make sure it stays in the marking (high) state when idle.

Another potential drawback here is interrupt servicing. If all the processors get one interrupt for every byte trans-

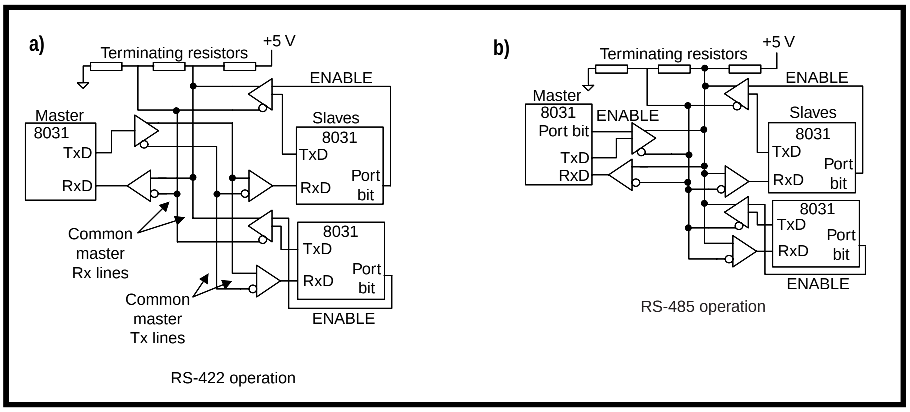


Figure 4a—*Adding RS-422 differential drivers and receivers enables multiple processors to communicate over long distances using asynchronous serial I/O.* ***b***—*The RS-422 receive and transmit lines can be combined into a single bidirectional pair of wires, normally called RS-485 in this configuration.*

mitted and received, whichever processor is transmitting at any given time gets two interrupts for every byte transmitted. It gets one interrupt for the transmitter and one when the same byte comes back into the receiver.

But, this situation isn't as bad as it seems. Sometimes you can turn off one interrupt while transmitting. Or, because the interrupt won't occur until the complete byte is received, you can use the receive interrupt as an indicator to turn the transmit buffer off.

HIGH-VOLTAGE OPEN-COLLECTOR

Figure 5a shows a variation on the open-collector communication method that permits use over longer distances. Like the previous example, there's one common party-line signal for communication in both directions. All transmitters drive the common line with open-collector drivers.

What makes this approach a little different is that the line is pulled to +12 or +24 V and the received data is passed through high-speed voltage comparators. The wider voltage swing

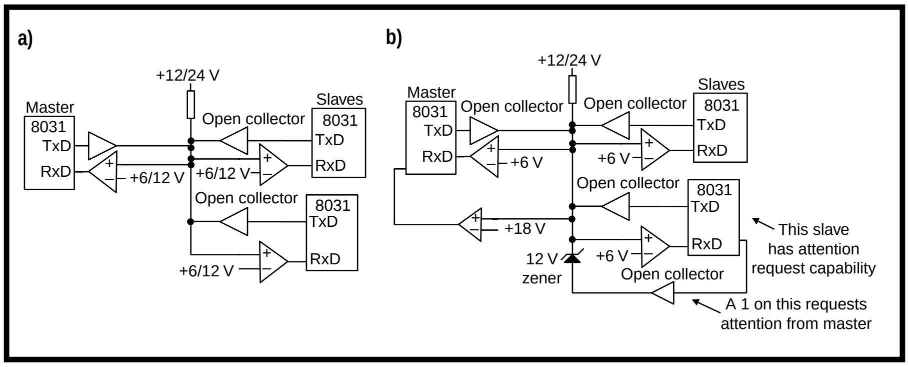


Figure 5a—*By using comparators as data receivers and pulling the line to +12 or +24 V, a single bidirectional open-collector serial interface can be implemented.* ***b***—*Adding a another comparator on the master processor enables the single-wire interface to add an attention request capability. Slaves request attention from the master by pulling the party line low through a 12-V zener diode.*

(compared to TTL) permits operation over longer distances.

The voltage comparators in this circuit act as buffers with a 6- or 12-V threshold (for 12- or 24-V operation, respectively). This arrangement provides about the same noise immunity as RS-232, but the open-collector party line permits bidirectional communication on a single wire.

One important consideration in this scheme is the value of the pull-up resistor. For 12-V operation, a 600-W pullup dissipates almost 0.25 W when any transmitter drives the line low. For 24-V operation, a 2400-W resistor provides the same dissipation.

Because the rise time of the common line depends on the value of the resistor and the capacitances in the wiring, 12-V operation enables faster communication. Also, if you have long cabling connecting your processors, the maximum data rate is limited because the resistance/capacitance limits the signal's risetime. But, this scheme enables multiple processors to communicate over a single wire (with ground).

All of these schemes depend on the master processor polling the slaves for data. Sometimes you need a slave to be able to request attention from the master. Figure 5b shows a variation on the open-collector party line that permits the slaves to request attention from the master.

The voltage comparators that act as receivers are all referenced to 6 V, but the circuit operates with a 0–24-V swing. The master processor has an extra comparator referenced to 18 V and connected to a port bit (or an interrupt input). Slaves that need to request the attention of the master pull the common line low through a 12-V zener.

When a slave requests attention from the master, the zener clamps the line to +12 V. The 18-V comparator on the master sees the transition, but the 6-V (serial buffer) comparator does not. The master sees the request and polls the slaves, one at a time, to see which one requested attention.

The key to making this method work is that the master must ignore transitions on the 18-V comparator while transmitting and receiving data. The 18-V comparator sees data transitions, so it must only be monitored when the line is idle.

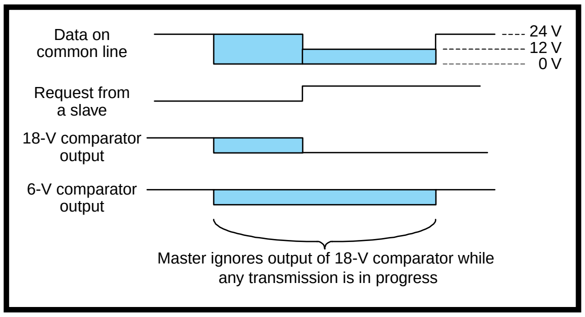


Figure 6—*Note that the output of the 18-V comparator follows the serial data until a slave requests attention. The output of the 18-V comparator then goes low and stays there until the slave removes the request.*

Figure 6 shows the timing and voltage changes. As you can see, a slave requests attention during a transmission. The party line initially swings 0–24 V, but when the slave requests attention, the voltage swing changes to 0–12 V.

At the end of the transmission, the output of the 6-V comparator returns to the 24-V level. But, the party line is held at 12 V by the zener diode, so the output of the 18-V comparator is low, indicating an attention request. This approach permits multiple processors to communicate, with an attention request, over a single wire plus ground.

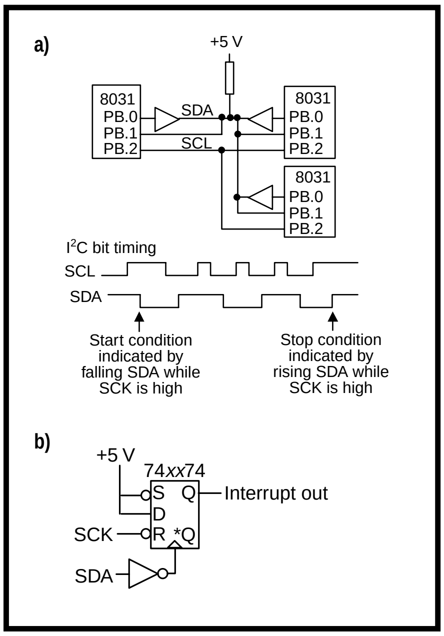


Figure 7a—*The I²C bus, which normally allows a processor to control peripheral ICs, can also be used for communication between processors.* ***b***—*A 74xx74 flip-flop provides an interrupt when the I²C start condition occurs, eliminating the need to poll for it.*

I²C BUS

Most of these techniques depend on the serial port. But, what if you're using the serial port for something else or are using a processor that doesn't have an internal serial port?

For these applications, you can use the I²C bus. As Figure 7a shows, the I²C bus uses a clock line controlled by the bus master and a bidirectional data line. All of the microcontrollers drive the data line through open-collector buffers and receive.

Figure 7a shows the timing for the I²C bus. Each transmission begins with the start condition, indicated by a falling edge on the data line (SDA) while SCL is high. The stop condition is indicated by a rising edge on SDA while SCL is high. When data is transferred, SDA changes only when SCL is low.

I²C data transfers typically consist of a start condition followed by an address and then data. Data bytes are acknowleged by the receiver. The complete I²C bus details are available from a number of sources (e.g., Philips Semiconductor).

The I²C bus is typically used to communicate with peripheral devices like EEPROMs and LCD drivers. It's not uncommon to find a single microcontroller connected to several I²C bus devices. In such a system, it makes sense to connect other microcontrollers as slave devices to the existing I²C bus.

For long-distance communication over cables, the I²C bus should be buffered using RS-422 or RS-485 buffers. You can use open-collector buffers, but all the inputs should use Schmitt-trigger devices to avoid noise problems.

One drawback to the I²C bus is the need to constantly poll for the start condition. Peripheral I²C devices do this in hardware, but a microcontroller that implements the interface in software wastes a lot of time polling for the start condition.

The circuit depicted in Figure 7b provides an output that can generate an interrupt when the I²C start condition occurs. Remember that the start condition is indicated by a falling edge on SDA while SCL is high. The circuit captures that condition in the flip-flop.

For this circuit to work, the SDA line must be noise-free, which means you need RS-422 or Schmitt-trigger buffers on the inputs if you're communicating over a cable. The interrupt line goes high on the falling edge of SDA and goes back low on the first falling edge of SCK.

Of course, using this technique means that you need three port bits and an interrupt to implement the I²C bus as a communication interface. As well, you need fast interrupt response to avoid missing any bits.

An additional consideration for a software-based I²C interface is throughput. The I²C bus can run at speeds up to 100 kBps. But, because every device on the bus must receive and decode every message, the clock rate is limited to the speed of the slowest device. If you add a software-based I²C slave to your design, the overall throughput to all the devices may have to be lowered.

If you do mix an onboard I^2C bus with external cable-connected slave processors, you need to buffer the I^2C signals where they leave the master board. The onboard peripherals probably can't drive the long cable reliably.

The simplest way to buffer the signals is to have the master control the direction of the buffered SDA line. The master knows which devices it's talking to, so it can enable the offboard buffer when it is receiving from an offboard processor. This arrangement needs an additional control line from the master processor to control the buffer.

GROUNDING

All of these interfaces depend on a good ground between the processors. The only way to avoid grounding issues is to use an optically or magnetically isolated interface.

Some interfaces are more tolerant of ground differences than others. TTL signals, with their 0–5-V swing and a threshold around 1 V, tend to be susceptible to grounding problems. It doesn't take much of a noise pulse to upset a TTL signal on a long cable, so TTL interfaces should be limited to less than ~10¢

The open-collector serial interface using 12- or 24-V swings is fairly immune to ground offsets, to about the same degree as RS-232.

The RS-422 interface is extremely insensitive to noise, but it can be more susceptible to grounding problems than the serial interface. RS-422 receivers detect a voltage difference between the two signaling wires, but the common mode voltage they can tolerate is typically 6–8 V.

I've seen RS-422 receivers destroyed when the systems are on two different grounds. In one case, an instrument was powered from a multiphase input and connected via an RS-422 interface to a PC running off the 115-V wall power. An air-conditioning compressor switched on and yanked the multiphase ground far enough to destroy the receivers on the PC end of the interface.

You can run into this same problem with two different 115-V outlets that have a ground offset. So, if you connect processors over distance, make sure the two grounds are the same or you may run into strange problems.

I didn't cover some of the more complicated interfaces like USB or Ethernet, but I looked at a number of interface techniques that are suitable for simple communication. Hopefully, one of them is suitable for your application.

Stuart Ball works at Organon Teknika, a manufacturer of medical instruments. He has been a design engineer for 18 years, working on projects as diverse as GPS and single-chip microcontroller designs. He has also written two books on embedded-system design. You may reach Stuart at sball85964@aol.com.

REFERENCE

Philips Semiconductor, *The I^2C Bus and How to Use it*, Application note, 1995.

VIDEO CAPTURE BOARD

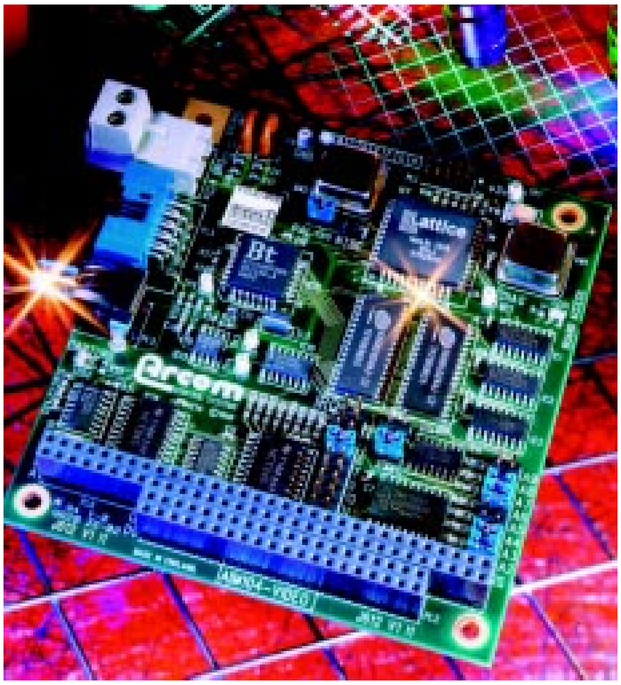
The **AIM104-Video** is a PC/104-format video-capture board designed for use in security surveillance and machine imaging applications. It inputs color or monochrome images from industry-standard devices such as digital video cameras and recorders. The board supports both PAL and NTSC video formats and provides digital pixel resolutions of 800 ´ 256 monochrome and 400 ´ 256 color. With 24-bit color depth, the AIM104-Video board offers a 16.7M color palette.

Input to the board is via a standard phono connector with an alternative 10-way connection. Power for an external video camera is supplied via a single two-part screw terminal connector with an onboard +12-V supply.

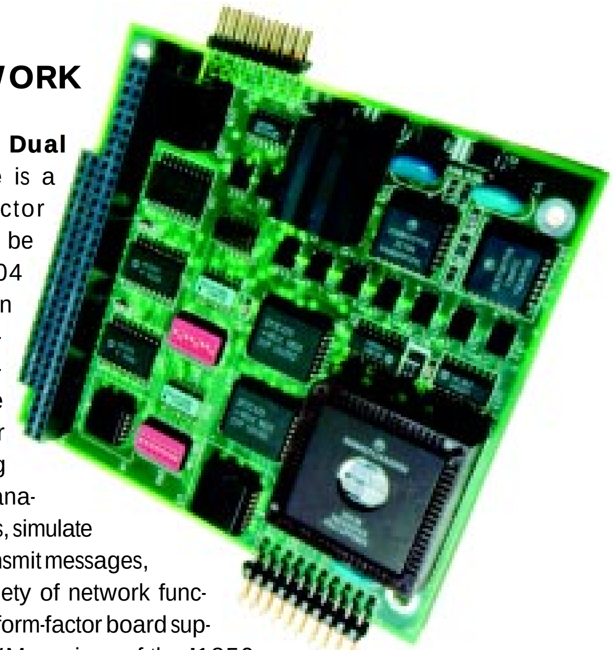
Each module is supplied with DOS drivers and C source code for capture, decoding, and .BMP format file storage. Precompiled executable files, useful for configuration during system development, are also supplied.

The ATM 104-Video sells for **$325**.

Arcom Control Systems, Inc.
(816) 941-7025
Fax: (816) 941-7807
www.arcomcontrol.com



AUTO NETWORK INTERFACE



The **AVT-931 Dual J1850 Interface** is a PC/104–form-factor board designed to be mated to a PC/104 host computer. It can be connected directly to a J1850-equipped vehicle and used to monitor network traffic, log network activity, analyze communications, simulate a network node, transmit messages, and perform a variety of network functions. The PC/104–form-factor board supports VPW and PWM versions of the J1850 vehicle networking standard, so it can be used with a majority of vehicle makes, including Chrysler, Ford, and GM.

The AVT-931 provides an electrically isolated interface between the host computer and the J1850 network of the vehicle under test. The board performs the necessary protocol conversions and all required communications translations that enable the user's host computer to communicate with the vehicle network. It uses the Motorola DLC device for VPW communications and the Motorola HBCC device for PWM communications. The VPW DLC device supports both transmit and receive operations in 4´ mode and block transfer operations. (The 4´ mode may be required by GM for some operations.)

Operation is controlled by software commands from the host to the hardware interface. The AVT controller software was designed for use in Windows 3.1x, but it also runs under Windows 95. The AVT-931 is supplied with a 16-bit DLL for use with 16-bit applications like Visual Basic (16-bit version) and other custom applications. A 32-bit DLL is under development.

The included hardware user's manual contains technical information on communications between the board and the host, connectors, and memory map. Available separately is a cable set that enables the AVT-931 to connect directly to the vehicle under test through the OBD-II connector now found in nearly all vehicles sold in the U.S.

The AVT-931 Dual J1850 Interface has a list price of **$1275**, which includes shipping.

Advanced Vehicle Technologies, Inc.
(410) 798-4038
Fax: (410) 798-4308
www2.ari.net/avt-inc

Nouveau PC

edited by Harv Weiner

TIME REFERENCE CARD

The **Model NTR2000-P** is a low-cost, real-time clock card for IBM PC–compatible computers that bypasses the system clock and the PC's BIOS to handle the year-2000 (Y2k) rollover. This device is capable of keeping highly accurate time and provides a stability of ±1 s per month—a vast improvement over standard DOS clocks, which are often more than 20 min. off. The unit is ideal for control, network, test, and measurement applications that require a more stable and accurate clock than is provided with most DOS systems.

The heart of the card is the National Semiconductor DP-8570A real-time clock IC linked to a stable crystal with a known aging rate. To further enhance precision and accuracy, the oscillator is calibrated to normal PC operating temperatures for the best reference possible.

The card also features ±0.01-s accuracy and user-settable auto updates from 1 min. to 45.5 days, extended at interrupts. A Novell driver is included, and the card comes with a lifetime guarantee.

The NTR2000-P sells for **$299**, including the board, manual, and software.

Industrial Computer Source
(800) 523-2320
(619) 677-0877
Fax: (619) 677-0815
www.indcompsrc.com

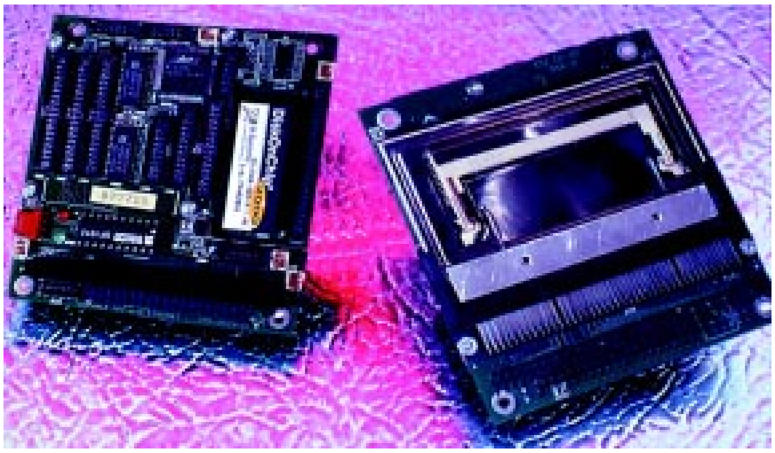
SCALABLE CPU

The **Scalable CPU II** provides the features most needed for demanding embedded computer applications as well as all standard desktop-PC features. The field-swappable credit-card–sized CPU may be scaled across the range from a '486-16 with 1 MB of RAM to a Pentium 200 with 128 MB of RAM. The Scalable CPU II is ideal for onboard vehicle, equipment control, robotics, military, aerospace, telecommunication, DSP, and other processor-intensive applications.

Standard desktop-PC features include support for VGA/SVGA display, IDE hard disk and floppy disk drive, two serial and one parallel port, and PS/2 mouse and keyboard interfaces. Specialized embedded-computing features include an NE2000-compatible 10BaseT Ethernet port, an SVGA LCD interface compatible with a wide range of flat-panel displays, and an alphanumeric LCD interface compatible with a wide range of character-based LCD modules. The unit has 16 bidirectional digital I/O lines compatible with scanning matrix keyboards and other devices. It also includes an M-Systems DiskOnChip socket that accommodates rewriteable flash memory modules from 2 to 72 MB, accessible as a standard hard drive using built-in ROM code.

The Scalable CPU II has been tested with PC-DOS; MS-DOS; Windows 3.*x*, 95, and NT; and QNX operating systems. It is currently under test with Windows 98.

parvus Corp.
(801) 483-1533 • Fax: (801) 483-1523
www.parvus.com



Nouveau PC

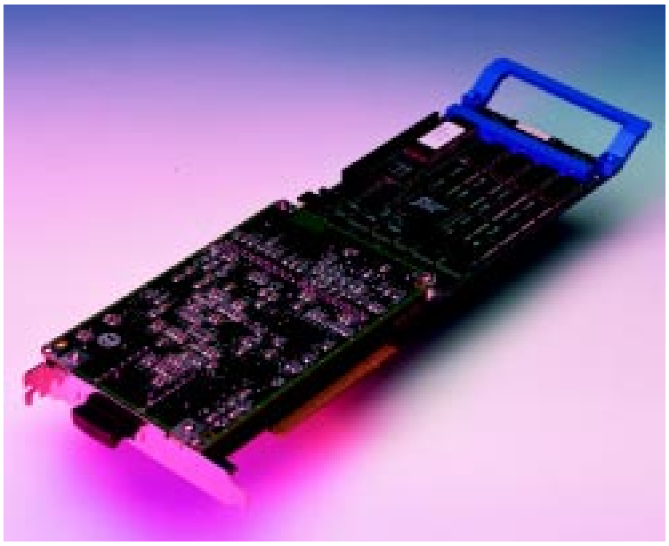
ATM ADAPTER

The **TNS1210 Streamliner** is an asynchronous transfer mode (ATM) adapter for exchanging MPEG video streams in PC-based video systems. The adapter's unusually fast throughput and constant bit rate enable video servers to provide a significantly higher number of simultaneous video streams in broadcast, Intranet, on-demand, and other video applications.

Standard ATM adapters rarely deliver more than 60 Mbps of constant bit rate streams, but TNS1210 has a 148-Mbps aggregate throughput with constant bit rate. This high throughput is achieved via multimemory/multibus architecture, onboard processing, a 1-Gbps cell memory, optimized drivers, and packetized burst DMA bus transfers.

The adapter comes with a full Windows NT software suite and offers concurrent management of various data flows including classical IP streams, MPEG-2 transport streams, and native AAL5 or AAL1 (ATM adaptation layers). Software supports PVCs (permanent virtual circuits) and SVCs (switched virtual circuits) with UNI 3.1 signaling. A physical layer interface running at 155 Mbps with SONET/SDH framing is available for multimode fiber, single-mode fiber, or UTP5 cables.

Up to 512 simultaneous connections are possible, including 64 dedicated to MPEG video streams. Mapping of the MPEG-2 transport stream into AAL5 is programmable for each connection, with up to 48 transport stream packets mapped into one AAL5 PDU.



An optional card supports AAL1 with a video convergence sublayer for error-free transport of video streams. Various compressed video formats and qualities are supported from 1.5 to 50 Mbps. Video streams are sent over AAL1 with ±25-ppm clock resolution and accuracy.

Tekelec Temex
+33 476-596034
Fax: +33 476-630030
www.tekelec-temex.com

PORTABLE FLOPPY DISK SYSTEM

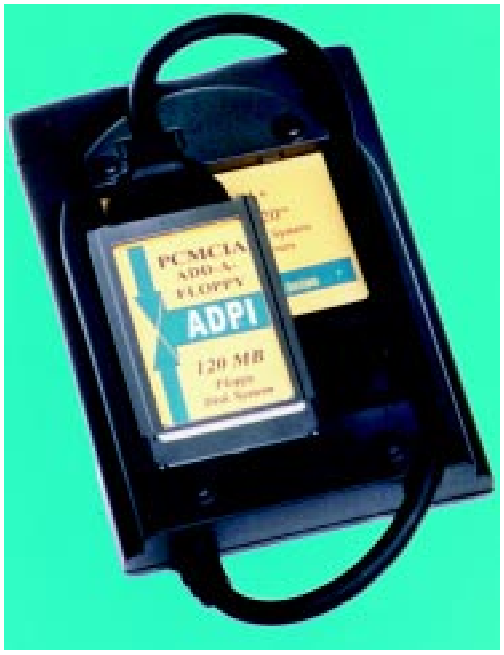
The **PCM-120 Add-A-Floppy** attaches to any computer with a PCMCIA Type II or Type III slot to read and write 120-MB disks as well as standard 3.5² disks. It is plug-and-play compatible with Windows 95 and Windows NT and can be used on virtually any laptop, notebook, PDA, or any computer with a PC-Card slot. Applications for the unit include data acquisition, storing and transporting large text or graphic files, storing sensitive data off-line, and freeing up space on a hard drive.

The PCM-120 is easy to transport and store, measuring just 4.5² ´ 7.25² ´ 1.66² (115 mm ´ 185 mm ´ 35 mm) and weighing 12 oz. The drive features a holder on the back of the case to store the PC Card, cord, and driver software disk. Power is obtained via external keyboard connector and averages only 0.09 W.

The PCM-120 requires a '386 or faster processor and a minimum of 400 KB of RAM. Software requirements include MS-DOS 3.3; Windows 3.1, 95/98, NT, or CE; and PCMCIA Card/Socket Services 2.1.

The PCM-120 sells for **$249**.

Analog & Digital Peripherals, Inc.
(937) 339-2241
Fax: (937) 339-0070
www.adpi.com



Nouveau PC

EPC

Thomas Wall

&

Mike Bauer

An OS for Test and Measurement

Looking to improve on some existing products, Burleigh Instruments did some comparison shopping for a new operating system. The result? Windows CE came out on top. Check in with Thomas and Mike to see why.

Aside from its often-cited advantages, including the de facto standard API and tools, a number of technical factors help Windows CE fit into the test and measurement industry. Consider its scalability coupled with broad product lines.

Many instrument manufacturers have a broad range of products with low to medium volumes, so they're always looking for leverage across product lines. Previously, there was no operating system that scaled well. But with Windows CE, these manufacturers can implement a single OS platform in everything from high-end handheld test devices to benchtop systems.

Windows is no stranger in the test and measurement industry. On the OEM side, we've seen a steady stream of Windows-based products, including the Inifium oscilloscope family from HP, the TLA 700-series logic analyzer from Tektronix, and EXFO's FTB-300 Universal (fiber) test system.

On the end-user and lower-volume OEM side, National Instruments has been selling LabVIEW along with their I/O boards, enabling users to build instruments out of almost any PC. The PC-based instrumentation market is huge, and many of these target systems run under Windows 95 and NT.

One common denominator for all instruments and test systems is a user interface, whether it's an LCD character display or a full Windows GUI. Windows CE has the ubiquitous Windows user interface and can also be used to develop custom GUIs.

Most test and measurement applications can be classified as soft real-time. These instruments primarily sample data and analyze it (sometimes at very high rates), but they're not involved in tight control loops where a highly deterministic response is needed.

The bounded determinism of Windows CE in the 150–250-µs range is more than adequate for the majority of applications. Traditionally, RTOSs have been used because of their small size and embedded capabilities rather than their real-time response capabilities.

SHIFTING FOCUS

The industry is being driven toward instruments that make more sophisticated measurements. Many engineers believe that software is now the gating technology.

By moving to a software (and even hardware) platform standard, engineering resources can be focused on the value-adding application and time to market can be reduced. As a de facto standard, Windows CE provides a variety of off-the-shelf tools and application building blocks.

These factors make test and measurement an early-adopter industry. In this article, we examine the process and applications of a test and measurement company, Burleigh Instruments, to see how Windows CE fits in with their applications.

Burleigh serves the fiber-optic–based telecommunication and semiconductor markets, which is evolving rapidly. Consequently, there's great pressure to increase the speed of product development.

Also, engineering expenses—those relative to the volume of instruments sold—

preclude developing custom hardware and software for each new product. And from the customer's perspective, no recognizable value is added by Burleigh's extra efforts to develop a unique, totally embedded product.

One way to reduce development time, cost, and effort is via commercial off-the-shelf (COTS) components. Low-cost, feature-rich SBCs are available, as are inexpensive data-acquisition and communication boards complete with drivers, function libraries, and configuration utilities. These components enable a low-volume instrument maker to tightly integrate a PC-based instrument to perform a specific function quickly and at minimal engineering cost.

FRIENDLY BONUS

Although Burleigh relies on standard data-acquisition, I/O, and communication boards in its systems, it still develops custom instrument hardware. Therefore, hardware usually arrives late in the product-development cycle.

How does a PC-friendly OS reduce time to market? Thanks to the architectural similarities between Burleigh's instruments and the PC, most instruments are easily simulated on the desktop development computers. COTS components are quickly integrated because vendors support the OSs used.

A major problem in the software world is the shortage of experienced programmers, especially those trained with the tools and particular APIs of a proprietary RTOS. But, the Windows CE application development tools are the same tools used for conventional Windows programming.

Other embedded Win32-based OSs, like Phar Lap, require you to purchase different tools. Because Windows CE uses a subset of the Win32 API, programming can be done with existing tools.

Burleigh instruments are configured with a variety of options. Some have flat-panel VGA displays, full keyboards, and hard and floppy disk drives. Others have only multiline, character-based displays with a minimal keyboard and no drives.

Windows CE has a lower degree of assumed architecture than Windows or even RTOSs, and the embedded toolkit (ETK) enables you to build a version tailored to your devices. The difficulty of the command-line–driven ETK can be overcome with products like Intrinsyc Software's Integration Expert, which gives the ETK an intuitive interface (modeled after Microsoft's Developer Studio). Thus, a single OS can be standardized.

A single OS has obvious advantages (e.g., eliminating the cost of supporting two technologies and multiple toolsets), but it can also reduce inventory. DOS and Windows 95 are bought in lot multiples, which may not match the number of instruments in production at the time of purchase.

Burleigh develops the software that their production department uses to qualify instruments for functionality and accuracy. If the OS used by the engineering and production departments has an API that's similar to the one the instruments use, test software can be a derivative of the instrument's application software rather than being completely independent and unique. Big savings in software development and support are possible if the applications share the same code base.

DOS VIDANYA?

DOS may be nearing the end of its useful life. It is no longer supported in the current set of Microsoft development tools, and our development tools are beginning to show their age. Once they're incompatible with the future versions of Windows, our tools will be useless.

Having workstations dedicated to support tools for software development is out of the question. And even documentation is getting harder to find.

As you know, Windows isn't an RTOS. Although your needs may be soft real time, events and responses to those events typically occur on the order of tens of milliseconds. Windows isn't deterministic enough.

Windows also continues to grow in bulkiness, so it's less suited to embedded applications. It consumes valuable resources (e.g., RAM), and its size makes it difficult to mount on devices such as flash ROM.

ADVANTAGE WINDOWS CE

Choosing an OS is like any other major design decision. Table 1 summarizes the major factors Burleigh considered when evaluating OSs.

The advantages of using a standard development environment and API were considered in general. Proprietary OSs like QNX and Wind River's VxWorks would have required new development systems and tools. Phar Lap's OS, while Win32 compatible, would have needed different development tools and a specific compiler.

Burleigh relies heavily on SBC and data-acquisition vendors to supply drivers for their products. The only commonly supported OSs are DOS and Windows, which provide few or no RTOS drivers.

Burleigh also looked at third-party development tools. After talking with several vendors, it became clear that Windows CE would be supported not only by them but also by others in their respective industries. For example, Annasoft adapted Windows CE to '486 and Pentium PC-compatible systems. Its CE Launcher enables Windows CE to load without DOS, and Jump Start provides driver support not provided by Microsoft.

There are several aspects to consider when estimating the cost of ownership:

	DOS	Windows	WinCE	QNX	Wind River	Phar Lap
Standard API (Win32)	No	Yes	Yes	No	No	Yes
Standard development environment	No	Yes	Yes	No	No	Yes
Third-party support						
SBC support	Yes	Yes	Yes	Yes	No	Yes
I/O drivers	Yes	Yes	Yes	No	No	Yes
GPIB drivers	Yes	Yes	Yes	Yes	No	No
Cost of ownership						
Development tool cost	N/A	N/A	$1500	<$2000	>$10,000	~$4995
Run-time license cost	~$19	~$119	~$35	~$250	~$50	~$30
Technical factors						
Real-time capability	No	No	Yes	Yes	Yes	Yes
Embedded capabilities	Yes	No	Yes	Yes	Yes	Yes
GUI support	No	Yes	Yes	Yes	Yes	No

Table 1—Here's the comparison chart that Burleigh Instruments used to determine which OS would provide the best solution for their needs.

- entry cost—the cost of development and support tools
- maintenance—the yearly cost of updating and maintaining tools
- customization—does the standard toolset permit scaling the OS to meet your application or does scaling to the desired configuration require additional fees?
- use (i.e., licensing and royalties)—run-time fees
- training—available locally or via third parties who can provide education and support?

For Burleigh, the entry cost of VxWorks was high, as was QNX's recurring run-time license cost. With its low cost of entry, high degree of customization, relatively low run-time licensing, and training available from multiple sources, Windows CE had the best cost of ownership.

Historically, Burleigh's application software has been targeted to DOS and Windows environments. So, in selecting an RTOS, Burleigh gave high consideration to OSs that support the Win32 API (e.g., Windows CE and Phar Lap).

Technical factors (e.g., real-time capabilities and size) were considered, too. All of the OSs met our basic requirements. But, the superior real-time capabilities and smaller footprint of RTOSs weren't important for our applications. A millisecond real-time response is adequate.

Size wasn't a huge factor because RAM is relatively inexpensive and not a critical consideration in Burleigh's highly specialized low-volume products.

A GUI, however, was critical. One drawback of the Phar Lap OS was its lack of a GUI. Windows CE's ability to provide a full Windows interface was an advantage.

SINGLE WAVELENGTH METER

One project in which Windows CE is likely to be applied is Burleigh's WA-1600 wavelength meter (see Photo 1), which currently runs under DOS. Designed for manufacturing environments, the WA-1600 precisely characterizes and optimizes wave division multiplexing (WDM) components like transmission lasers, fiber Bragg gratings, and erbium-doped fiber amplifiers (EDFAs).

WDM enables telecommunications carriers to increase bandwidth by sending multiple optical signals over a single fiber. As WDM equipment continues to increase the number of channels supported, the ability to more accurately measure optical wavelength is necessary to properly characterize and optimize the WDM components in the equipment. The WA-1600 measures the wavelength of a single-mode continuous laser to within 0.2 ppm (the highest-accuracy wavelength measurement available).

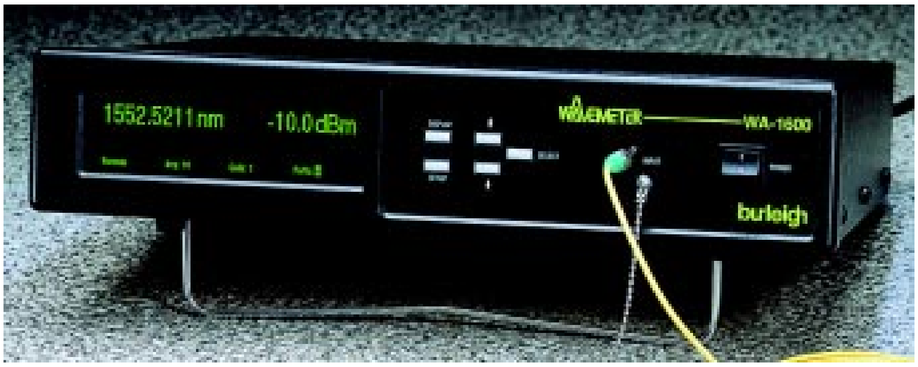


Photo 1—The WA-6000 wavelength meter's software will be ported from DOS to Windows CE.

A stand-alone benchtop and rack-mountable instrument, the WA-1600 uses a Teknor Viper806 SBC with a '486 processor, 8-MB RAM, 4-MB flash memory, and bidirectional communication interfaces like RS-232 and Ethernet 10BaseT (using TCP/IP).

The Viper806 has a PC/104 interface onto which Burleigh adds proprietary hardware and a Computer Boards IEEE-488 (GPIB) and analog and digital I/O modules. The display is a Noritake vacuum fluorescent display, which interfaces to the Viper-806 via a custom keyboard interface.

WA-1600 OS REQUIREMENTS

To implement Windows CE on this device, we need full driver support for all third-party boards. We chose Teknor and Annasoft to provide the drivers for all the interfaces onboard (video, keyboard, floppy, parallel and serial ports, dual IDE, and flash file system). Jump Start provides most of the drivers off-the-shelf. Teknor provides drivers for their flash-memory and dual IDE systems.

We need utilities equivalent to those we used in the past. During configuration and assembly, we use the keyboard and display interface that is standard on the Viper806 to run Teknor's various configuration utilities.

So, Teknor has to supply the Windows CE equivalent of the utilities used to set up the system BIOS extensions supporting the onboard flash memory and the video BIOS extensions supporting flat-panel displays. Configuration utilities for the Computer Boards modules have to be ported to the OS. Then, we have to port our own drivers.

Additionally, the OS must conform to some of the constraints of the platform and application. The Viper806 supports up to 4 MB of flash memory. The OS, drivers, application, and associated initialization files have to fit into this space.

The full version of Windows CE occupies ~5 MB. But because the WA-1600 doesn't have a GUI or networking interface, we can use the limited version, reducing both the size and the recurring license cost (by a third).

The analysis-and-optimization tools in Integration Expert help identify components that can be eliminated. Since our application is only ~400 KB and doesn't need many features, we're confident that Windows CE can be configured to occupy well under 4 MB.

Because of the accuracy and precision our instruments must deliver, Windows CE has to support a true double-precision floating-point library. We also have to worry about errors like floating-point and hardware exceptions.

For example, if the result of a calculation causes a divide-by-zero, a DOS or Windows application might terminate and return the user to the desktop or, worse yet, the `C:` prompt. Similarly, a user who removes the floppy during a write operation can't be asked to respond to an error message through a user interface equipped with five keys. We need to embed the SBC as deeply as possible, so we need to intercept and gracefully handle such exceptions.

Real-time performance isn't an issue. The data-sampling rate is less than 10 times per second, and user-interface response to keyboard input happens in less than 250 ms.

PULSED WAVEMETER

Another project likely to use Windows CE is the WA-5500 pulsed wavemeter optical wavelength meter. Its current configuration requires the coupling of the optical head to a PC running application software under Windows 95. We want to completely embed the PC in the instrument while enhancing instrument performance.

The desire to reduce IC feature size places great demands on the optical lithography process, and the optical source has to operate at shorter wavelengths. We addressed this need by employing KrF excimer lasers that operate at 248 nm. For optimum performance and efficiency of the optical system, the absolute wavelength of the excimer laser must be known and controlled to a high accuracy.

The new WA-5500 measures the absolute wavelength of excimer lasers to ±0.002 nm at 248 nm. A special design can be used to achieve an absolute accuracy of ±0.0005 nm at 248 nm.

Architecturally, the WA-5500 is like the WA-1600, except it uses the Viper821, which is Pentium-based and has up to 16-MB SanDisk–based flash memory.

This instrument (along with Windows CE) needs to support a Sharp TFT full-color VGA flat-panel display and a 2-D CCD camera. The result: more graphical information must be transferred to the graphic display and there is a larger data set to be collected from the CCD camera. Because the source of the data is a pulsed laser, the information must be collected, reduced, and analyzed as rapidly as possible.

We anticipate Windows CE's real-time capabilities to be pushed as we gauge the limits of system bandwidth. We'll have to work around the absence of nested interrupts through the use of semaphores.

Using the available priority levels and bounded task response in the 150-µs range, we expect to achieve the necessary performance and measurement accuracy. Windows CE V.3.0's nested interrupts and 32 levels of prioritization should let us improve the instrument's performance.

READY TO GO

There's little question that Windows CE is being widely adopted and supported, not only by SBC and data-acquisition vendors but also by instrument manufacturers. Once again, the early adopters have proven that something new can be something good. EPC

Thomas Wall is the software manager for the electro/optics product group at Burleigh Instruments.

Mike Bauer is the director of marketing and business development at Annasoft Systems. You may reach Mike at bauer@annasoft.com.

SOURCES

Windows CE, Jump Start
Annasoft Systems
(800) 590-3870
(619) 674-6155
Fax: (619) 673-1432
www.annasoft.com

Operating Systems
QNX
(800) 676-0566
(613) 591-0931
Fax: (613) 591-3579
www.qnx.com

Wind River
(800) 545-9463
(510) 748-4100
Fax: (510) 749-2010
www.windriver.com

Phar Lap Software, Inc.
(617) 661-1510
Fax: (617)876-2972
www.pharlap.com

Integration Expert
Intrinsyc Software, Inc.
(604) 801-6461
Fax: (604) 801-6417
www.intrinsyc.com

Ingo Cyliax

Embedded RT-Linux

Part 4: Developing Under Linux gcc/gdb

Want to work with Linux? Well, then, you need to know about the development tools. Fortunately, all of Linux's development tools run under Linux—that includes everything from compilers and languages to I/O debugging.

Over the past few months, I've been looking at Linux and discussing some of the issues involved with embedding Linux. But to embed Linux, you need development tools. That's what I'm focusing on this month.

Linux is a self-hosted development system. In other words, all of the development tools for Linux run under Linux. They include the standard C/C++ high-level language support with compilers, linkers, libraries, and source-level debuggers.

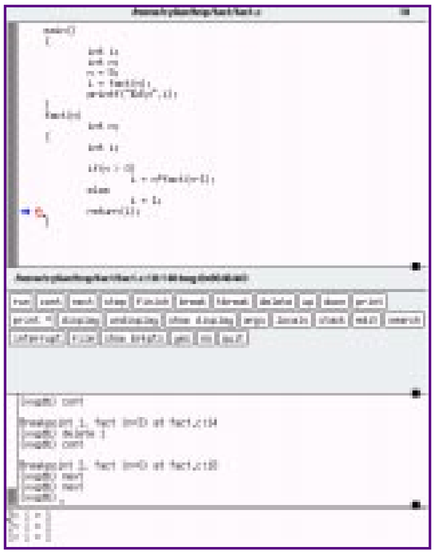
Along with the standard C/C++ tool chain, Linux provides support for other languages. There's a just-in-time (JIT) virtual machine for Java Kaffe that enables you to run Sun's JDK, including their Java compiler and support tools, under Linux.

Languages like Pascal and FORTRAN 77 are supported via code converters that convert Pascal and FORTRAN into C. Then, you just use the normal C/C++ tool chain.

For you computer science types, there's a scheme interpreter (UMB-Scheme) and a commercial scheme compiler (Chez Scheme) available. There are even scripting languages like Tcl, Perl, and Python.

Because Linux's development is self-hosted and freely distributable, it's possible to ship the development system with the embedded system if you have resources available. If you're limited in memory and disk footprint, you need to host the development in a large host or desktop system and download the application to your target.

The same system interface is used in the development host and target, so you can test and debug your application on the development system. As a final step, embed the code into your embedded-Linux system.



```
main()
{
        int i;
        int n;
        n = 5;
        i = fact(n);
        printf("%d\n",i);
}
fact(n)
        int n;
{
        int i;

        if(n > 0)
                i = n*fact(n-1);
        else
                i = 1;
        return(i);
}
```

```
(xxgdb) cont

Breakpoint 1, fact (n=1) at fact.c:14
(xxgdb) delete 1
(xxgdb) cont

Breakpoint 2, fact (n=0) at fact.c:18
(xxgdb) next
(xxgdb) next
(xxgdb) _
```

Photo 1—*The X Windows version of the GNU debugger (*xxgdb*) has a source window, a command-line window, and an extra window to display variables. These are features you'd expect from a source-level debugger.*

LET'S C

Linux supports many programming languages, but I want to take a detailed look at C. I'll show you how to build a simple hello-world–type application and what you need to do to arrive at an executable.

Using C is pretty straightforward. The source-code modules are in files with a `.c` extension. These modules are compiled and assembled into object files by the compiler. The object files use the `.o` extension and, along with static libraries, are linked into an executable or dynamic-load module by the linker.

A static library manager enables you to combine object modules into a library. Static libraries use the `.a` extension, and dynamic link modules (i.e., shared libraries or objects) use the `.so` extension.

The C++ compiler works similarly but uses `.cc` file extensions as source file input. In general, a developer uses the gcc program as the front end to the compiler and linker phase.

gcc knows where in the system to find standard `include` files, the `.h` extension, and libraries. It also knows about extensions and figures out the sequence to apply the compiler, assembler, and linker to arrive at an output file.

To compile a single module into an executable file, invoke `gcc hello.c`. It looks at the file specified on the command line and recognizes the `.c` extension as a C source file. After running the source file through the C macro/`include` file preprocessor, it calls up the C compiler and compiles the program into an assembly-language program.

Assembler files use the `.s` extension. Before linking with the standard start-up object module and standard libraries, `gcc hello.c` calls up the assembler to assemble the file into an object file and generates `a.out`, which is the executable.

You can use the `-v` switch to get a trace of all the subprograms called and the switches and arguments used (see Figure 1a). As you see, the program does a lot and the options are complex. That's probably why most people use the gcc front end to compile programs.

gcc has many switches that control the operation of the preprocessor, compile, assembly, and linking phases. There are options to override the standard search paths for the `include` file and libraries. Also, the compiler can be controlled with options to compile functions in-line for a particular processor or if you want to use hardware floating point or emulation.

a)

```
gcc -v hello.c
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/2.7.2.3/specs
gcc version 2.7.2.3
  /usr/lib/gcc-lib/i386-redhat-linux/2.7.2.3/cpp -lang-c -v -undef
  -D__GNUC__=2 -D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux
 -D__ELF__ -D__unix__ -D__i386__ -D__linux__ -D__unix -D__i386 -D__linux
  -Asystem(unix) -Asystem(posix) -Acpu(i386) -Amachine(i386) hello.c
  /tmp/cca16646.i
GNU CPP version 2.7.2.3 (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
  /usr/local/include
  /usr/i386-redhat-linux/include
  /usr/lib/gcc-lib/i386-redhat-linux/2.7.2.3/include
  /usr/include
End of search list.
  /usr/lib/gcc-lib/i386-redhat-linux/2.7.2.3/cc1 /tmp/cca16646.i -quiet
  -dumpbase hello.c -version -o /tmp/cca16646.s
GNU C version 2.7.2.3 (i386 Linux/ELF) compiled by GNU C version 2.7.2.3.
  as -V -Qy -o /tmp/cca166461.o /tmp/cca16646.s
GNU assembler version 2.9.1 (i686-pc-linux-gnu), using BFD version
  2.9.1.0.4
ld -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 /usr/lib/crt1.o /usr/
  lib/crti.o /usr/lib/gcc-lib/i386-redhat-linux/2.7.2.3/crtbegin.o -L/
  usr/lib/gcc-lib/i386-redhat-linux/2.7.2.3 -L/usr/i386-redhat-linux/lib
  /tmp/cca166461.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-redhat-linux/
  2.7.2.3/crtend.o /usr/lib/crtn.o
```

b)

```
5 -rwxr-xr-x   1 cyliax   wheel     4149 Nov  7 20:05 hello.dyn
431 -rwxr-xr-x   1 cyliax   wheel   437473 Nov  7 20:13 hello.sta
```

Figure 1a—These are the steps that gcc takes to compile a program into an executable. gcc acts as a front end and directs what tools need to be executed with their options to compile a program or module. b—Here's what you see when you look at the file sizes for a statically and dynamically linked programs. Although the dynamically linked program seems to be much smaller, remember that you also need the run-time libraries that it was linked against to run it.

The linker can generate several different output file formats such as ELF, COFF, or Unix `a.out` format. More controls make it possible to link the program statically or dynamically.

Dynamic linking permits commonly used libraries to be loaded at runtime rather than including them in the executable. Because libraries are shared, using dynamic libraries makes executables smaller. But, you need to make sure the dynamic library is available at runtime.

For embedded systems where you don't want dynamic libraries, you may want to compile your program using the `-static` switch. Using `gcc -static hello.c` makes a big difference for a simple program (see Figure 1b).

Although the static version is almost 0.5 MB, the dynamic version is only 4 KB. The dynamic version shares the library with other programs, resulting in a smaller disk image. But if the system has just one program, then a statically linked image contains all that's needed and is smaller than all the dynamic libraries combined. Note that these sizes are the file sizes and have little to do with the run-time size of the program.

A more complex program may consist of several modules, which you might want to link separately. The `-c` switch tells gcc to stop after compiling and assembling the program into an object module. All modules are linked into an executable `prog1`, as in Figure 2. The same concepts apply for C++ modules. Just substitute g++ for `gcc` and use `.cc` for the source modules.

Along with linking modules to programs, you can put the modules into a library with the `ar` utility. This archive program takes individual modules and builds one file containing the modules. To build a simple library, execute `ar qv mylib.a mod1.o mod2.o`.

It's good to build a symbol table in the library archive. Although it's not necessary,

```
gcc -c main.c         # generate main.o
gcc -c mod1.c         # generate mod1.o
gcc -c mod2.c         # generate mod2.o
gcc main.o mod1.o     # link modules
  mod2.o -o prog1
```

Figure 2—When you're building a program with two source modules, you first compile them into object files and then link them into an executable.

the linker can use the symbol table to quickly find references in the archive rather than sequentially searching it for symbol references.

To index the library, use `ranlib`. After typing `ranlib mylib.a`, you can link your program with the library by using `gcc main.o mylib.a -o prog1`. The linker extracts all the modules from the library archive and satisfies all of the symbol references that need to be resolved to link the program.

```
a) hugo 87% make
   cc    -c main.c -o main.o
   cc    -c mod1.c -o mod1.o
   cc    -c mod2.c -o mod2.o
   ar qv mylib.a mod1.o mod2.o
   a - mod1.o
   a - mod2.o
   ranlib mylib.a
   gcc main.o mylib.a -o prog1

b) hugo 86% make
   cc    -c mod1.c -o mod1.o
   ar qv mylib.a mod1.o mod2.o
   a - mod1.o
   a - mod2.o
   ranlib mylib.a
   gcc main.o mylib.a -o prog1
```

Figure 4a—You can use `make` to build a program that depends on several modules in a library. b—Here's what happens when you change one module (`mod1.c`). `make` figures out which module changed and only performs the steps necessary to update the project.

Libraries are often used for large projects with many modules where several programs use the modules. This way, you don't have to keep track of all the module names. You just stick them into a library and let the linker sort it out.

Keeping track of individual modules in a project can be tricky and cumbersome, especially if you want to figure out which modules need to be recompiled to update the program. Luckily, there's a utility that handles this task: `make`.

Different versions of `make` have been around for a while under Unix and before Linux. The version that comes with Linux is the GNU `make` facility. It's quite powerful and has many features, so I'll only be able to scratch the surface here.

`make` is a facility for maintaining objects that are made up of dependent components. You describe the project by building a `makefile` like the one in Listing 1.

`make` works by having a dependency line for a target. In my example, the top-level target is `prog1`. `make` uses the first target in the `makefile` as the top-level target. Tell `make` that `prog1` is made up of `main.o` and `mylib.a`. In turn, `mylib.a` is made up of `mod1.o` and `mod2.o`. After the dependency line, list the commands needed to generate that target.

Because the object modules—`main.o`, `mod1.o`, and `mod2.o`—are derived from C source modules, you don't need to tell `make` that `main.o` is made from `main.c` by compiling it. `make` already has built-in rules on how to make `.o` from many different kinds of source modules.

`make` knows that you can also make `.o` files by compiling FORTRAN from files that have the `.f` extension and C++ from files with `.cc` extensions. Listing 2 shows some of the built-in defaults for `make`.

You can see that with the `makefile` and its built-in rules, you are building a dependency tree that looks like Figure 3. Now, when I execute `make`, it generates all the commands necessary to build my top-level target (see Figure 4a).

This, in itself, is a good deal, but `make` is also smart enough to check the modification times of all the components necessary to build the target. If I edit `mod1.c` and change the module, I only need to run `make` again and it rebuilds my program by executing only the steps necessary to update the project, as in Figure 4b.

For large projects or a project where many programmers are working on different parts, this feature is handy. For example, `make` is used to maintain the Linux kernel and all the packages available under Linux.

Besides `make`, there are tools that generate `makefiles` for a project or component based on some higher level rules. One tool is `Imake`, which uses configuration files and prototype `makefiles` to build custom `makefiles` for specific installation. It even builds the X Windows source tree. But in the end, `makefiles` control how the project components are compiled or built.

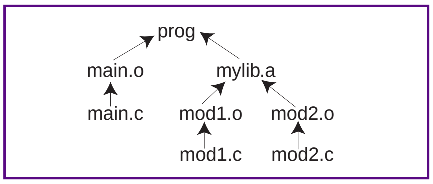


Figure 3—Here's the dependency tree for the sample `makefile` in Listing 1. The leaf nodes, `main.c`, `mod1.c`, and `mod2.c` are the source modules needed to build the program. `make` knows how to build common things, like compiling `mod1.c` to get `mod1.o`.

To further aid in software maintenance, software revision tools are available. One of these is the revision control system (RCS). RCS lets you checkpoint individual source modules and check them into an archive.

RCS also keeps track of the differences between module revisions and lets you go back or consolidate different versions. Because Linux is multiuser, it keeps track of which user checked out a module and is making changes to it.

In its simplest form, you use `ci` for check-in. Checking in a module compares it to the last version of the module and saves the most current version but notes the differences. If I check in `mod1.c`, I get the dialog shown in Figure 5a.

Signing my name to the change isn't necessary because the change is noted in the RCS log. To view the revision log for a file, use the program `rlog`. Figure 5b shows that user `cyliax` checked in V.1.2.

Perhaps one of the best features of RCS is that `make` knows about it and will check out a module with RCS if it is currently checked in. So, going back to rebuilding `prog1` after I have checked in `mod1.c`...

Figure 5c shows that `make` used the program `co` to check out the source module `mod1.c` and then proceeded to compile, build the library, and relink the program. Finally, it deleted the module because it wasn't needed anymore.

`co` is used to check out a module from the archive. By default, `co` checks out the

Listing 1—The target `prog1` consists of a `main.o` module and a library `mylib.a`, which in turn is made up of `mod1.o` and `mod2.o`. `make` reads this file and uses a system-wide `make` file to construct a dependency tree.

```
prog1: main.o mylib.a
  gcc main.o mylib.a

mylib.a: mod1.o mod2.o
  ar qv mylib.a mod1.o mod2.o
  ranlib mylib.a
```

Listing 2—The system rule file contains all the standard rules that `make` uses to transform files from one extension to another.

```
#       @(#)sys.mk      5.11 (Berkeley) 3/13/91
unix=           We run UNIX.

.SUFFIXES: .out .a .ln .o .c .F .f .e .r .y .l .s .cl .p .h

.LIBS:          .a
AR=             ar
ARFLAGS=        rl
RANLIB=         ranlib
AS=             as
AFLAGS=
CC=             cc
CFLAGS=         -O
CPP=            cpp
FC=             f77
FFLAGS=         -O
EFLAGS=
LEX=            lex
LFLAGS=
LD=             ld
LDFLAGS=
LINT=           lint
LINTFLAGS=      -chapbx
MAKE=           make
PC=             pc
PFLAGS=
RC=             f77
RFLAGS=
SHELL=          sh
YACC=           yacc
YFLAGS=-d
.c.o:
        ${CC} ${CFLAGS} -c ${.IMPSRC}
.p.o:
        ${PC} ${PFLAGS} -c ${.IMPSRC}
.e.o .r.o .F.o .f.o:
        ${FC} ${RFLAGS} ${EFLAGS} ${FFLAGS} -c ${.IMPSRC}
.s.o:
        ${AS} ${AFLAGS} -o ${.TARGET} ${.IMPSRC}
.y.o:
        ${YACC} ${YFLAGS} ${.IMPSRC}
        ${CC} ${CFLAGS} -c y.tab.c -o ${.TARGET}
        rm -f y.tab.c
.l.o:
        ${LEX} ${LFLAGS} ${.IMPSRC}
        ${CC} ${CFLAGS} -c lex.yy.c -o ${.TARGET}
        rm -f lex.yy.c
.y.c:
        ${YACC} ${YFLAGS} ${.IMPSRC}
        mv y.tab.c ${.TARGET}
.l.c:
        ${LEX} ${LFLAGS} ${.IMPSRC}
        mv lex.yy.c ${.TARGET}
.s.out .c.out .o.out:
        ${CC} ${CFLAGS} ${.IMPSRC} ${LDLIBS} -o ${.TARGET}
.f.out .F.out .r.out .e.out:
        ${FC} ${EFLAGS} ${RFLAGS} ${FFLAGS} ${.IMPSRC} \
          ${LDLIBS} -o ${.TARGET}
        rm -f ${.PREFIX}.o
.y.out:
        ${YACC} ${YFLAGS} ${.IMPSRC}
        ${CC} ${CFLAGS} y.tab.c ${LDLIBS} -ly -o ${.TARGET}
        rm -f y.tab.c
.l.out:
        ${LEX} ${LFLAGS} ${.IMPSRC}
        ${CC} ${CFLAGS} lex.yy.c ${LDLIBS} -ll -o ${.TARGET}
        rm -f lex.yy.c
```

Listing 3—This program has to be compiled with the optimization turned on so the `outb()` function is inline. It also needs to be run at system-level privileges to get permission to access the ports.

```
#include <asm/io.h>
main()
{
  unsigned char i;
  ioperm(0x378,8,1);
  while(1){
    outb(i,0x378);
    i++;
  }
}
```

module in an unlocked mode. In this mode, a read-only copy is created. To change the module, you need to check out and lock the module with the `-l` switch.

`co -l mod1.c` checks out the locked module and creates a writeable version. In this state, no one else can check out this version of the module until I check it back in. But, they can create their own version of the module or even a new revision branch. RCS works on individual files and there's a higher-level version control system—the concurrent version system (CVS)—that extends RCS to a project-level version-control system.

That just about covers compiling programs. Remember, even though I've only shown how to use `make` and RCS with C modules, they'll work with different kinds of source-code modules.

In fact, I use RCS for all of my writing to make sure I don't lose anything, even if I go back and delete some thoughts or edit the article. I also use `make` to maintain and build hardware components like PCB netlists that make up a design, components, and libraries for FPGA large designs.

I/O

Although it's a high-level OS, Linux enables application programs to access 'x86 I/O ports if the program has appropriate permissions. The `include` file `asm/io.h` contains macros to generate inline functions for I/O instructions such as `outb` and `inb`. But, there are some rules.

First, the program that wants to use the I/O instructions needs to turn on the range of addresses these instructions will work for. This action is done via `ioperm()`. To

a)
```
hugo 82% ci mod1.c
mod1.c,v  <--  mod1.c
new revision: 1.2; previous
  revision: 1.1
enter log message,
  terminated with single '.'
  or end of file:
>> I made a trivial change.
  -ingo >> .
done
```

c)
```
hugo 82% make
co  mod1.c,v mod1.c
mod1.c,v  -->  mod1.c
revision 1.2
done
cc    -c mod1.c -o mod1.o
ar qv mylib.a mod1.o mod2.o
a - mod1.o
a - mod2.o
ranlib mylib.a
gcc main.o mylib.a -o prog1
rm mod1.c
```

b)
```
hugo 82% rlog mod1.c
RCS file: mod1.c,v
Working file: mod1.c
head: 1.2
branch:
locks: strict
access list:
symbolic names:
keyword substitution: kv
total revisions: 2;
selected revisions: 2
description:
----------------------------
revision 1.2
date: 1998/11/09 02:29:18;
author: cyliax;
state: Exp;  lines: +1 -0
I made a trivial change. -ingo
----------------------------
revision 1.1
date: 1998/11/09 02:28:56;
author: cyliax;  state: Exp;
Initial revision
```

Figure 5a—Here's the dialog between `ci`, the RCS check-in program, and the author. `ci` records the comments in the log it keeps with each module it manages. b—You can use `rlog` to look at the revision log for a module. c—When `make` needs a module that has been checked in and is maintained by RCS, it checks out the module with the `co` command before using it.

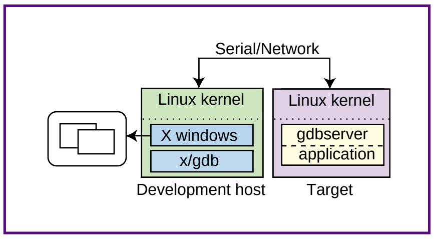


Figure 6 – Here's the setup to use if you want to use `gdbserver` to remotely debug an application on a target system. `gdbserver` can use a serial port or any network connection to communicate with the debugger running on the development system.

execute `ioperm()`, the program needs to run at the supervisory or system-privilege level (sometimes called root level).

For the I/O instructions to be inline, the program needs to be compiled with the optimizer on. Listing 3 opens an eight-byte range, starting at the parallel port (0x378), and simply counts on the parallel port.

With the I/O instructions, you can write C programs that easily access I/O cards. But if your I/O card generates interrupts, you need to write in a device driver to communicate with it.

There's currently no way to deliver interrupts to application-level programs. Writing device drivers for Linux is beyond the scope of this article, but you can find books on the topic. Because the sources are available, you can also use an existing device driver as a template.

DEBUGGING

If you use the GNU software development tools for Linux to compile and link programs, you can use the GNU debugger (gdb) as well. gdb comes in two flavors: command-line and X Windows.

Photo 1 shows a screenshot of the X version (`xxgdb`). You'll need a display that's running X Windows to display it on. The command version of gdb, of course, runs on any command-line window or a terminal over a serial port.

If your application is in a target that has no graphics-capable display, you have three choices. You can run the command version over a serial port on the target to a host or a terminal. Or, if you have a network connection, you can run the X version of the debugger on the target and use a remote display for its output.

This feature is an artifact of the X Windows system in which all applications talk to the Windows server via a network connection, even if they're running on the same host. In my example, the X client is running on a remote machine (i.e., the target) and displaying on an X server, which can be on a notebook or desktop system.

If you're pressed for memory and disk space on your target, use `gdbserver` to attach to your embedded applications. `gdbserver` is a small program that implements remote debugging stubs for gdb.

You can use either the network or a serial port to attach a remotely running program via `gdbserver` and run gdb (the command-line or the X version), on a different host with more resources (see Figure 6). `gdbserver` comes with the gdb source distribution and needs to be built, but it's freely available.

With `gdb` and `xxgdb`, whether running locally or remotely, with or without the `gdbserver` debugger, you have access to all the standard source-level debugging features such as setting breakpoints, tracing program flow or access to variables, and starting or stopping programs.

GOT ALL THAT?

Well, that wraps up this installment, but you can find more information on developing software under Linux in your nearest bookstore. There are plenty of books on software development, system administration, and even Linux internals. Although I didn't cover languages like Perl, Tcl, and Python, they're available under Linux, too.

Next month, I'll look at real-time applications using RT-Linux. Remember, RT-Linux is a real-time extender that sits underneath Linux and enables real-time threads to run independently of Linux processes.

As I've mentioned before, Linux isn't the answer to all your embedded-system needs. It's just another tool in your toolbox. RPC.EPC

Ingo Cyliax has written for INK *on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego–based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. Before joining DSI, Ingo worked for over 12 years as a system and research engineer for several universities and as an independent consultant. You may reach him at cyliax@derivation.com.*

Fred Eady

In the Face of Medusa

Part 2: A Whole New Solution

Fred had it all figured out. His tried-and-true design was set to go, but then a conversation with a friend turned him on to something new. Using an Octagon 6040 and an industrial-strength BASIC made all the difference in the world.

I really enjoyed doing the PicStic-4Q stuff last time. You know, it's great to be able to drop the inside bit-banging coding, enjoy some powerful prewritten routines, and just plain have fun with it. I never thought that it would be possible, but you can have the same good times with an 80386 embedded platform, too.

Before I talked to Rick Applegate at Octagon, I envisioned using my tried-and-true 4010 embedded PC to tame the Medusa. Because the 4010 doesn't have any dedicated I/O pins, I would have used standard PC ports to fake the implementation of digital and analog I/O operations.

And, although the complexity isn't there, C would have been my language of choice. Along with C, I would have needed a development system to pull the application off. Well, as you might have guessed, that ain't the way it's gonna be.

You can think of the Octagon 6040 as the mother of all PicStics. Even though they come from differing schools, they're similar in many ways.

The Octagon 6000 series of embedded PC cards was designed to ease the process of building an embedded-control application. Like PicStic-4Qs, the 6040 has a gaggle of I/O lines that includes ADC (eight lines) and DAC (two lines). Kinda like having a PicStic-4Q with an Intel processor.

Photo 1—I/O...anytime...anywhere...any weather.

For example, the elimination of a development system makes the 6040 very mobile when it comes to maintaining a library of routines or applications. The 6040 can be programmed in any of the academic ways, but the CAMBASIC language makes application maintenance a bit less hairy. Datalight's ROM-DOS V.6.22 is standard with the 6000 series, and if you want to blow off DOS and load QNX, that's OK, too.

The 6040 was designed to be rugged as well. The only moving part is the board as you plug it into a passive backplane. Phoenix supplies the 6040 with the Pico FA flash file system that eliminates spin. Two solid-state disks are standard, with one SSD consisting of 1 MB of flash memory and the second SSD being 128 KB of battery-backed SRAM.

The 6040, in minimum program configuration, leaves about 512 KB of flash memory for your application. The SSDs are augmented with 2 or 4 MB of on-card DRAM. What would you do with all that?

But, if you must go in circles, the 6040 has a novel way of hooking up an external diskette drive—a multifunction parallel port. You simply tell the 6040 what is where on the parallel port pins in the setup.

The 6040 BIOS, also from the Phoenix stable, is equipped with industrial extensions. If you like writing device drivers, buy another board. This one comes with a driver library. If you don't see what you want, turn to CAMBASIC for your custom driver code.

BASICALLY EMBEDDED

CAMBASIC is the industrial-strength version of what we all know as QBASIC or QuickBASIC. Like the traditional BASICs, CAMBASIC is easy to use and easy to learn. If you can already program in any other BASIC language, you can do CAMBASIC.

The idea behind CAMBASIC was to relieve the industrial-control designer from having to wear the programmer's hat. Unlike regular BASIC, CAMBASIC wasn't written for your Model 4. It was designed to be fast in a real-time sense.

The code may look the same, but where it counts, CAMBASIC routines are faster and tighter than their commercial BASIC counterparts. Time-sensitive parts of the code are compiled without the user being aware that it's being done. CAMBASIC is a real-time multitasking version of BASIC aimed at data control and acquisition applications.

One of the features that distinguishes CAMBASIC is individual bit manipulation. I've seen this on other BASIC compilers such as PowerBASIC, but it's not standard in the world of BASIC. CAMBASIC has the ability to access every I/O line on the 6040 individually.

Also, 80386 assembly code can be put inline with the CAMBASIC statements. I don't have to tell you what that means. As you'll see later, CAMBASIC also has built-in functions that mimic the functionality found in the PicStic-4Q. Functions like debounce routines, keypad routines, serial I/O, and analog control can all be found in its syntax.

There are even interrupt-on bit, comm port, and keypad routines. The CAMBASIC language and the hardware architecture of the 6040 work well with the PicStic-4Q.

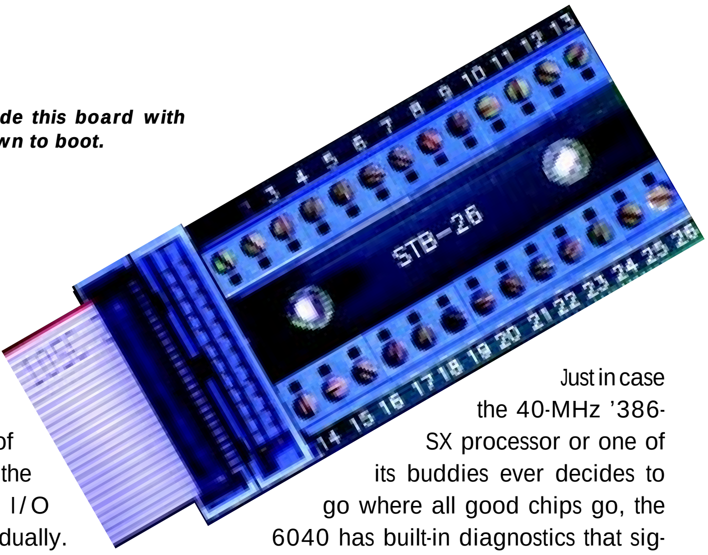


Photo 2—You can cascade this board with another and screw it down to boot.

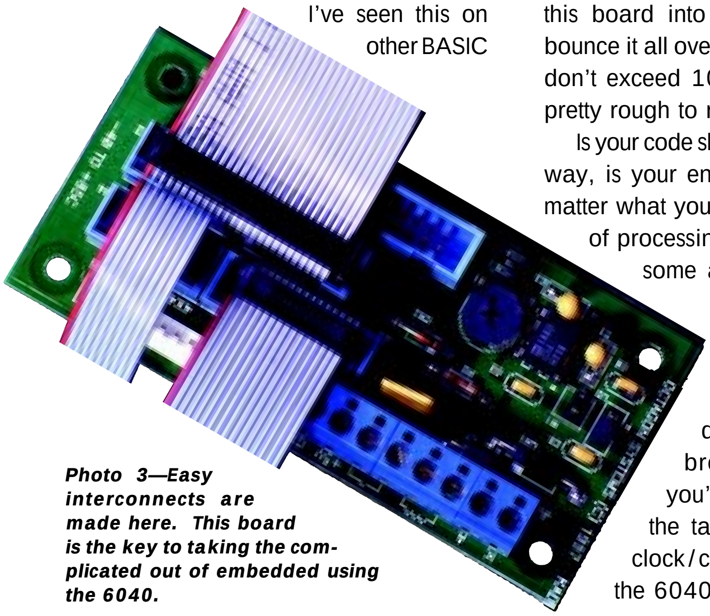


Photo 3—Easy interconnects are made here. This board is the key to taking the complicated out of embedded using the 6040.

ONBOARD WITH THE 6040

Being in Florida, I don't experience the problem as much, but I hear that in colder climes, static discharge is a real problem. The 6040's solution is 8 kV of ESD protection on the two serial ports and backdrive protection on the parallel port. Added protection is provided by optoisolation on some of the interrupt inputs.

With the exception of the SRAM battery, nothing is socketed. So, if you want to put this board into an M1-series tank and bounce it all over creation, make sure you don't exceed 10 Gs. You'd have to get pretty rough to rip the ICs off this one.

Is your code slippery, or, to put it another way, is your environment iffy? Well, no matter what you blame it on, interruption of processing can be catastrophic in some applications.

The 6040 helps keep mission-critical code running with the inclusion of an onboard watchdog timer. And, if you break your Timex while you're bouncing around in the tank, an integral real-time clock/calendar is standard on the 6040.

Just in case the 40-MHz '386-SX processor or one of its buddies ever decides to go where all good chips go, the 6040 has built-in diagnostics that signal trouble via sound and LED activity. No special diagnostic tools are necessary to troubleshoot the 6040 while it's in service. Just power it up, and it runs internal diagnostic routines automatically.

Breaking the 6040 by hooking up the power incorrectly is possible but unlikely. First of all, the 6040 uses a single 5-V supply. Over-voltage and reverse-voltage circuitry protects against that "oops" (usually followed by "uh-oh").

Now that you know what the 6040 and PicStic-4Q combination can do, let's bring the Medusa application to life. If all this talk has got you wondering what the 6040 looks like, look no further than Photo 1.

A TOUCHY-FEELY INTERFACE

In the brief description of the 6040's inner workings, it was mentioned that the parallel port could be used to drive a standard 1.44-MB floppy disk drive. Well, that's just one thing the 6040's multifunctional parallel port can do.

Seventeen of the 41 general-purpose I/O lines can be had here, too. A 4 ´ 4 matrix keypad or four-line alphanumeric display can also be attached. Of course, you can use it for a printer port, too.

The data lines are hefty and can sink up to 24 mA. The PicStic-4Q can do 25 mA. The parallel port is also referenced in the manual as the AUX I/O port. The LPT mode is determined during setup.

That's nice, but I haven't told you much about the I/O structure, other than to say there's bunches of it. The 6040 is one of three 6000-series devices that contains something called EZ I/O. EZ I/O is a chip that supplies 24 I/O lines (three sets of eight) that can be individually programmed as input or output.

The lines are automatically configured for input mode on powerup. Hmmm…

Listing 1—The gotcha compensation code is missing here, but the ballot casting logic is what we're interested in anyway.

```
symbol ALIVE = b3
output 2                 'RS-232 out
input 1                  'RS-232 in
input 7                  'make pin PB7 input

ALIVE = 0                'clear keep alive register
for b2 = 1 to 15         'read pulse 15 times
pause 1                  'wait 1 ms
ALIVE = pin7             'store read
if pin7 = 1 then TALLY
ALIVE = ALIVE & 2        'clear the pulse low bit
goto NOTALLY
TALLY:
ALIVE = ALIVE | 2        'set the pulse high bit
NOTALLY:
next b2

if b3 = 0 then VOTE

VOTE:
serout 2,6,("vote")      'instruct other PicStic to vote
serin 1,6,("myvote"),b4
if b4 = 0 then VOTEOUT
```

sounds like I/O on another device in the article. Of course, the lines are TTL compatible and are designed to work in the 5-V range. These lines aren't as meaty as the parallel port lines and can only sink or source 15 mA each.

The EZ I/O pins are brought out on a header (J1) that interfaces via ribbon cable to a screw-down terminal block like the one shown in Photo 2. These lines are just like any other I/O port lines and are all supported in the syntax of CAMBASIC.

Additionally, all 24 lines can be tied to +5 V or ground through a 10-kW resistor by simply placing jumpers to do so. With the exception of the pulling resistors, the EZ I/O chip looks and feels like an 8255.

I went to all that trouble to come to this point—we need an interface. You know me, it's got to be simple but powerful. Checking the PicStic-4Q doc, you can see how the module is designed to drive displays and keypads. Looking at the 6040 doc, well, you know.

You can easily add a 4 ´ 4 keypad and LCD to the 6040 by attaching a keypad and display board or KAD like the one shown in Photo 3. This is too easy. We can attach the KAD to the AUX I/O port or the EZ I/O port. The only difference is the cable used and the number of pins we tie up.

Let's keep the EZ I/O open and attach the KAD to the AUX I/O port. I attached a 4 ´ 20 LCD display and a 4 ´ 4 keypad to the KAD, as you can see in Photo 4. To make the KAD hardware program-ready, the only thing left to be done is to make sure the AUX I/O port is configured as LPT1, a bidirectional printer port.

GETTING OUT OF THE BOX

OK, I have a simple but informative and easy to use interface coupled to the 6040 embedded stamp via the KAD board. I also have all of the EZ I/O at my disposal to do with as I please. The 6040 is ready to service Medusa…almost.

I am assured of program control and data in, but what about data out? Yep, the serial ports are still available. I'll need one later, so only one serial port is available.

I hear you mumbling, "Fred always uses the serial port. I'll bet he was born with a

Listing 2—The `.01` parms set a 10-ms pulse width and the `AUTO` keyword makes for a continuous pulse train.

```
10 Config PIO &100,0,0,0,0,0     ;configure the 8255
20 Clear Timer 0 :               ;clear timer before starting
30 Config Timer 0,&100,1,1,.01,.01,AUTO ;set 10-ms pulse width
40 Start Timer 0 :
```

nine-pin connector….” In case you didn’t read Part 1, I’m going with Ethernet here, so hold on.

I thought it would be simple. The Octagon 5500 Ethernet adapter board fits right on the passive backplane with the 6040. Unfortunately, this misconception was the initial downer in the whole project.

It’s not that support was bad or the hardware was flaky. There were a few gotchas, and I got by most of them.

Days later, after trying almost every trick I could think of, I was talking to one of the programmers I do my day thing with, and we started talking about the new Microsoft Lan Client that replaced the MS Workgroup Add-on.

I thought I saw somewhere that the latest Lan Client code was available from the Windows NT V.4 CD. Wrong. Seems that this version doesn’t permit full sharing. So, I went back to the bench and ripped out the new and put in the old. I then picked up all those bad cables and reboxed that bad hub.

During all of this, I’d at least ironed out one problem—the lack of SSD space to hold the network files. Remember, there’s about 512 KB of flash memory posing as a hard drive available after ROM-DOS and BIOS get through playing, but there’s plenty of DRAM to turn into a virtual disk. I can cram the other program-related stuff into battery-backed SRAM if necessary.

Here’s how I squeezed the network code in. I first ran the set-up program that identifies the network card and also asks other pertinent networking questions on a standard PC running DOS.

Once the network directory was completed, I zipped the directory and loaded the zip file into the flash SSD along with the unzip program.

The 6040 comes with a terminal program (PC SmartLink) that has the capability of making the flash memory look like a local drive on the host PC. This program makes copying files between the 6040 and a host PC easy. With a little batch file magic, I created a VDISK in the remaining DRAM space and unzipped my network directory into it.

Once the network files were placed into their drive, I pointed to them and executed the proper NET command to load the SMC9000 Ethernet chipset on the Octagon 5500 Ethernet Adapter card. Of course, I shared my 6040 resources and the SSD became accessible on the LAN.

Finally, the host embedded hardware is complete and ready to be programmed and configured. Photo 5 shows the 5500 card that gave me so much grief.

Photo 4—Mark should be an astronaut. Most astronauts would fly just to watch the lights, push the buttons, and read the LCD on this interface panel.

GETTING ON THE STIC

The idea here is to integrate a couple PicStic-4Qs with the 6040 using minimal external hardware and a somewhat common programming language. The end result is a powerful embedded host system and an equally powerful subsystem. So, let's map it out, and I'll show you snippets of the more significant code.

Figure 1 shows the pinouts of the two PicStic-4Qs. PS-A controls the stepper motor that drives the Medusa syringes. PS-A RB3 is the output pulse applied to the stepper. PS-A RB4 determines the step direction. The module measures the relative humidity and temperature using a HyCal sensor via the PS-A AIN pins.

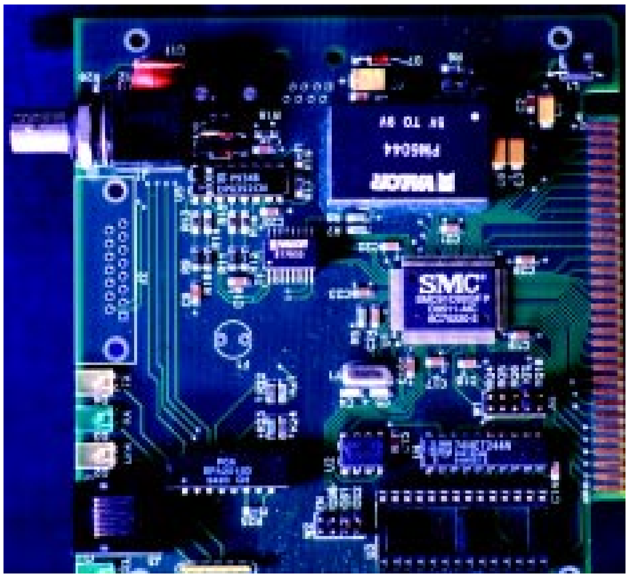


Photo 5—It's all here. You just need to know what to do with it when you find it.

The sensor is factory-calibrated and emits a voltage for a percentage of humidity. Temperature readings are provided by a thermistor that we read the voltage across. A small constant current is applied to the thermistor to provide a voltage that corresponds to temperature.

Eight bits of resolution is fine for our needs. The RS-232 port, defined by PS4 BASIC at PS-A RB1 and RB2, relays data to the 6040 host. In case of emergency, the RS-232 port determines the health of the computing hardware.

Two pins at PS-A, DIO3 and DIO4, are used to communicate between PS-A and PS-B. This communications channel is used as an "are you there" between the two devices. The protocol is clocked (i.e., one pin clocks the data presented on the other pin).

Using the same "are you there" principle, PS-A RB7 is configured as an input that expects to see a continuous square wave from the 6040 host. If it isn't detected, the 6040 is presumed to be out of service.

Both PicStics monitor this line and vote on the health of the host. If both don't see a signal, it's a landslide that the 6040 is dead. At this point, each module reverts to its internal clock and continues until its memory was exhausted. This way, at least some data is gathered.

If one PicStic votes yes and the other votes no, it's possible that one of them is wrong and possibly broken. At this point, PS-A uses the serial port to attempt to communicate with the 6040. If all is well, operations resume as normal but keep-alive monitoring is disabled on the incorrect voting module.

If both PicStics vote "dead" but continue to be given commands, then hopefully they can at least see each other and signal that they both may be broken. This situation would be classified as fatal and all processing would be terminated. If all seems to be in the dumper, PS-A pin DIO2 is poised to reset the 6040.

Normally, I'd note these events in a log, but for this experiment, that would take up precious space and time. The last thing I want is a logging loop. Listing 1 shows how the PicStic-4Qs cast their ballots.

The "I am here" square wave from the 6040 is generated in the background and places little, if any, burden on the main thread. CAMBASIC has the versatility to run other asynchronous activities in the same fashion.

Another good example of background execution is the serial I/O function of CAMBASIC. The main thread can be executing at full speed while the background serial I/O routine buffers incoming data. Listing 2 shows the "I am here" code.

Of course, the main purpose of assembling all this computing power is to germinate some vegetation. Photo 6 shows the test fixture that drives the Medusa stepper and the HyCal sensor.

Pushing fluids to exact locations is rather difficult in orbit, which is why the Medusa system was designed. The lack of gravity tends to mess up things that were intended to stay on earth. So, Medusa has to pump liquid in a precise manner to wherever it needs to end up.

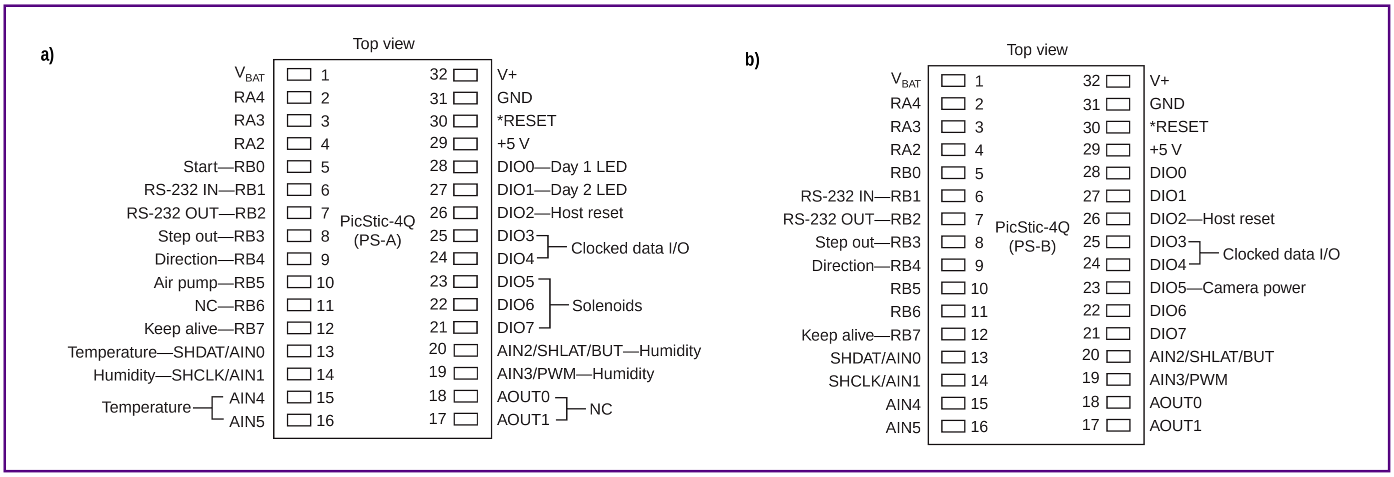


Figure 1—PS-A has pins dedicated to temperature, humidity, and more (a), whereas PS-B is monitoring camera power, among other things (b).

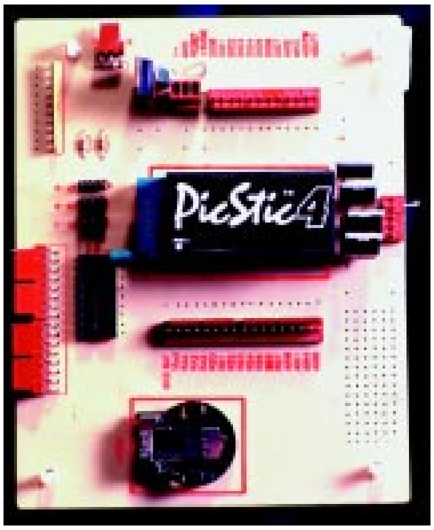


Photo 6—Here's what a full-blown machine shop with a intuitive operater/owner can do with a PicStic-4Q and some perfboard. (I supplied the electronics.)

Once the veggies come alive and all the pictures are taken, the plants are fixed or killed so their states are preserved. This technique enables the ground scientists to further study the experiment.

The PS-A RB5 is responsible for fixation. It controls a pump that forces a fixation liquid from an IV bag into the plant chambers. PS-A pins DIO 5, 6, and 7 control the solenoids that determine which chamber gets fixed.

STEPPING OUT

PS-B's code is like the PS-A's stepper code. The camera arm is calibrated by moving a flag into an optical sensor. The number of steps between dishes is determined, and steps happen until something says stop. The camera attached to the arm feeds NTSC video to a frame-grabber board plugged into the passive backplane.

There's plenty of EZ I/O left. Some of it will be allocated to running the front-panel LEDs that signal experiment status.

The experiment is still in the preflight stages, and I'm sure almost all of the I/O lines will be used before it's over. If we run out of I/O on the 6040, there's lots of PicStic-4Q I/O left, too. These three microcomputers make up a formidable experiment control system with system diagnostic capability built in.

The combined power of PicBasic, CAMBASIC, PicStic-4Q, and the 6040 has once again proven that it doesn't have to be complicated to be embedded. APC.EPC

Fred Eady has over 20 years' 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

80386 6040
Octagon Systems
(303) 430-1500
Fax: (303) 426-8126
www.octagonsystems.com

IH-3602
Honeywell/HyCal
(800) 932-2702
www.honeywell.com/sensing/prodinfo/temperature/

Pico FA
Phoenix Technologies
(408) 654-9000
Fax: (408) 452-1985
www.ptld.compico

PicStic-4Q
Micromint, Inc.
(800) 635-3355
(860) 871-6170
Fax: (860) 872-2204
www.micromint.com

ROM-DOS V.6.22
Datalight
(800) 221-6630
(360) 435-8086
Fax: (360) 435-0253
www.datalight.com

MICRO SERIES

Joe DiBartolomeo

TPU
Freeing the CPU

Part 2 of 4

Now that we know what the TPU is, we can begin to put it to work for us. In this installment, Joe applies the TPU to some processor-intensive applications. The big advantage of TPU is the increase in throughput.

When I began this series last month, I focused on the most common timer/counters found on microprocessors and introduced the time processor unit. In this installment, I want to apply the TPU to some common and normally processor-intensive timing functions.

In these applications, the TPU handles all the timing and counting details. When the TPU completes the timing function, it passes the results to the CPU, increasing system throughput. In fact, several timer/counter applications are so processor-intensive that, without the TPU, a multiprocessor solution would be required.

As you may recall, the TPU is a coprocessor to the CPU and runs semi-autonomously. The CPU sets up the TPU for the required timing function and is then free to perform other tasks.

The CPU can monitor the TPU's progress by polling it. But, a far more common and useful method is to have the TPU generate an interrupt when it completes the timing function.

A timing function is a set of TPU microcode instructions that directs the TPU in performing a specific task. TPU microcode is run by the execution unit in the TPU. This architecture is completely different from the standard timer/counter that's found on most microprocessors.

The TPU can perform prepackaged timing functions (i.e., microcode stored in microprocessor ROM). Or, it can run user-written microcode stored in onboard RAM or flash memory. This month, I discuss the preprogrammed "canned" functions that were listed in Table 1 of Part 1 (p. 70, *INK* 102).

Once again I'll be working with the 68332 microprocessor from Motorola. The TPU on the 68332 has 16 channels that can be programmed to perform any combination of functions. Here, I use the TPU in two applications: motor speed control as well as distance, speed, direction, and position mapping.

From a programmer's point of view, running the canned functions is straightforward. The CPU globally initializes the TPU and sets up the individual channel for the desired timing function. Global initialization was discussed in Part 1, so now it's time to talk about programming the individual channels for particular timing functions.

SPEED CONTROL OF DC MOTOR

The speed of a DC motor is proportional to the voltage applied to the motor—the larger the applied voltage, the faster the motor rotates. In low-power systems, the motor drive can be a DC voltage. But as power levels increase, heat-dissipation problems occur in the output driver stage.

For this reason, at higher power levels, DC motors are normally driven using PWM signals. Figure 1a shows a motor-speed control scheme using analog blocks.

To control the speed of a DC motor using a microprocessor, we have to measure the speed of motor rotation and produce a PWM drive signal. Although imaginative programmers can use just about any timer/counter to perform this task, the TPU's built-in functions make it ideal for this application. Figure 1b shows how the TPU can control the speed of a DC motor.

An optical encoder (see sidebar, "Encoders") connected to the shaft of the DC motor produces a pulse train with a frequency that represents the motor speed. The function measuring the frequency of an incoming signal on a channel's external pin is the frequency measurement function (FQM)—function $C of mask set G.

The FQM measures the number of pulses that occur during a time window, and the value is passed to the CPU. The CPU then calculates the frequency by multiplying or dividing to normalize to cycles per second. The '332 has a multiply/divide instruction that takes 64 clock cycles.

Because the time window is user defined, selecting a time window of base 2 enables the multiply/divide instruction to be a shift/rotate to the right or left. If the actual frequency of motor rotation isn't needed, the number of accumulated pulses can be used to represent motor speed.

Once the CPU obtains a representation of the motor speed, it can derive or look up the required PWM characteristics. The CPU then uses another TPU function to produce a PWM motor-drive signal.

Mask set G has a multichannel PWM function that can generate the required PWM signal. But, it's intended for complex multichannel PWM applications.

When you're using mask set G and a single PWM channel is required, the queued output match function (QOM) is a better choice. If you use mask set A, select functions PPWA and PWM to perform the FQM and PWM functions.

Now, let's set up the TPU channels for the FQM and PWM functions. Recall from Part 1 that each TPU channel has a number of bit fields used to set up and control timing functions. Each channel also has six words (channels 14 and 15 have eight words) of dual-port parameter RAM for transfering data to or from the CPU.

Figure 2a is a programmer's map of a TPU channel. Although setting up a TPU channel is straightforward, because the channel bit fields are in TPU registers shared by other channels, you have to be careful.

For example, as you see in Figure 2b, there are two host service request

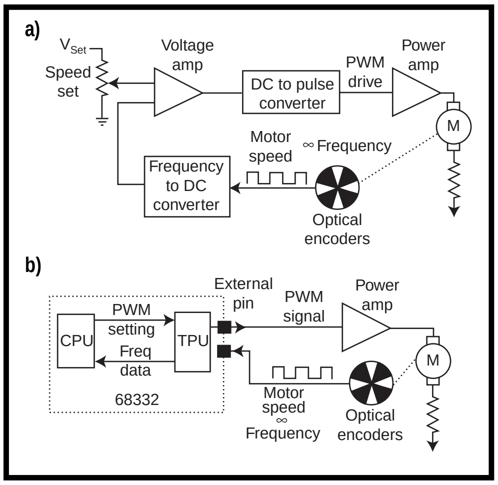


Figure 1a—*This DC motor-speed control scheme uses a PWM drive signal.* ***b***—*This one, by contrast, uses a TPU-equipped 68332. Note that only two microprocessor pins are required to control the motor speed.*

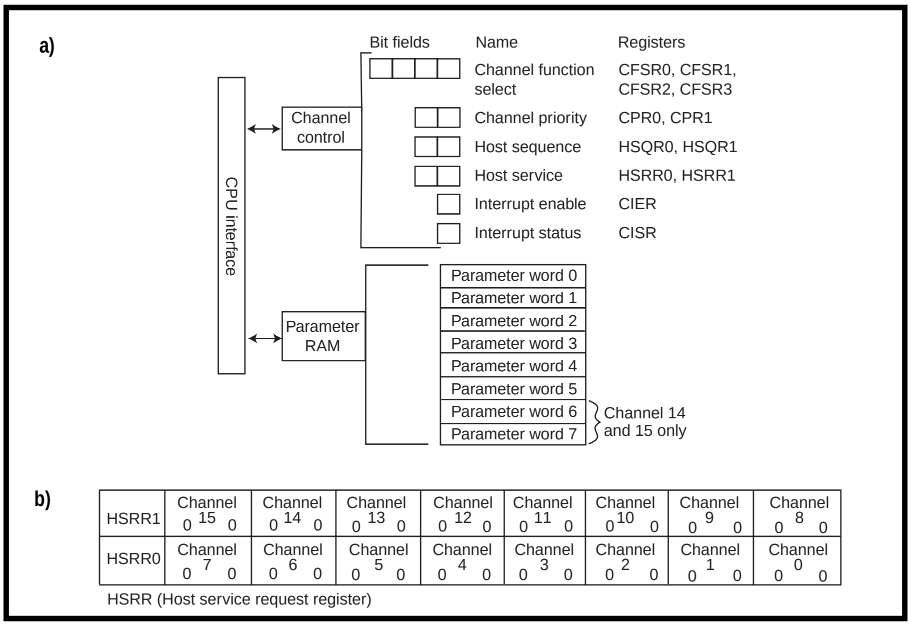


Figure 2a—*Here's the programmer's map of a single TPU channel.* ***b***—*In the host service request register layout, there are eight channels sharing each register. In changing any channel's HSRR bit field, you must ensure that the other channels are not affected.*

registers, HSRR0 and HSRR1. HSRR0 contains the host service request bit fields for channels 0–7, and HSRR1 contains the host service request bit fields for channels 8–15.

When you update the host service request bits for any given channel, be sure that you don't inadvertently alter the host service request bits of other channels.

MEASURING FREQUENCY

The FQM function has two modes of operation—single shot and continuous. In single-shot mode, incoming pulses are accumulated for one time window. But, in continuous mode, the pulses of repetitive time windows are accumulated.

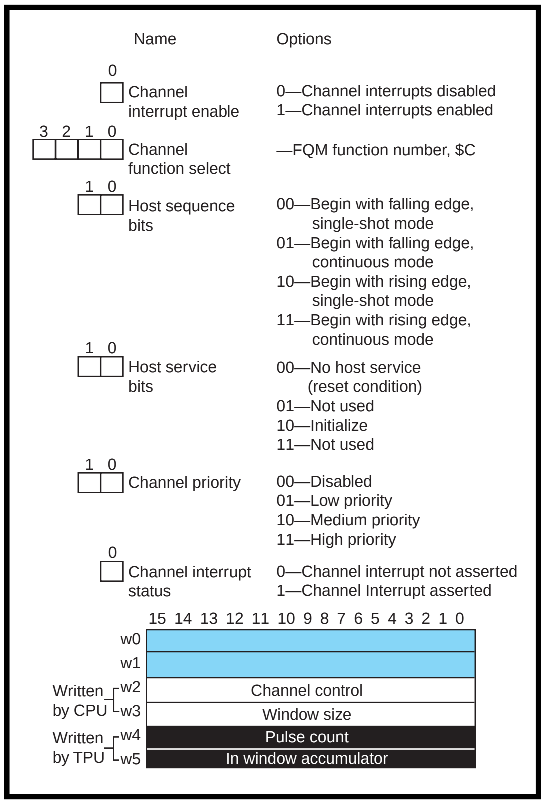
The CPU must perform the following steps to start the FQM function, assuming global initialization was already performed. Actually, the first step is to select a channel. This determines the bit fields in the shared TPU registers that must be programmed.

For example, if you are using TPU channel 5, then bits 8 and 9 in the HSRR0 form the bit field for the host service request, as you can see in Figure 2b. Figure 3 shows all the bit fields that need to be programmed, along with the parameter RAM.

The first step in setting up a TPU channel to run the FQM function is to clear the appropriate channel priority bits that disable the channel. Next, the FQM function number ($C) is written into the appropriate bits of the channel function-select register, selecting the FQM function for that channel.

As the third step, the CPU writes CHANNEL_CONTROL and WINDOW_SIZE values into channel parameter RAM. The CHANNEL_CONTROL parameter sets the rising or falling edge to start pulse detection on, timer TCR1 or TCR, and continuous or single-shot mode.

Figure 3—*The CPU sets up the TPU via the bit fields and parameter RAM. The TPU returns results to the CPU via parameter RAM.*



WINDOW_SIZE is a 16-bit number representing the number of timer ticks in the time window. Multiplying this parameter by the timer period gives the time window in seconds.

After this, the host sequence bits are written to define the function's action. This setup must match the one established by CHANNEL_CONTROL.

Next, the CPU writes the host service request bits—in this case, 10—to have the TPU perform an initialization once the channel is enabled. As their name implies, the host service request bits indicate to the TPU channel what type of service the CPU is requesting.

Once the channel interrupt bits are set up, there's only one more step to go. Set the channel priority bits to any one of the three possible nonzero values—01, 10, 11. This enables the function and assigns priority as low, medium, or high. The TPU then executes a channel initialization.

After the TPU channel is initialized, it waits for the first appropriate edge transition to appear on its external pin. The transition signifies the start of an incoming pulse, and the time window begins.

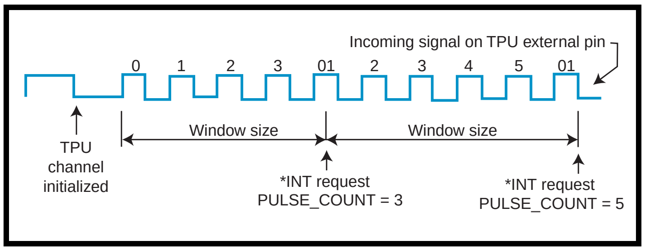
When the opposite edge is detected, the transition signifies the end of the incoming pulse, and the value in IN_WINDOW_ACCUMULATOR is incremented. The value in IN_WINDOW_ACCUMULATOR is incremented every time a pulse is detected until the time window (WINDOW_SIZE) expires.

When the time window expires, the parameter RAM value in IN_WINDOW_ACCUMULATOR is transferred to the parameter RAM location PULSE_COUNT and an interrupt is issued. If the TPU channel is set for single-shot mode, it stops and waits for the CPU to issue another service request.

In continuous mode, after the value in IN_WINDOW_ACCUMULATOR is transferred to PULSE_COUNT and an interrupt is requested, a new time window is started and the channel begins accumulating pulses as illustrated in Figure 4.

Note that the new timing window begins immediately and any pulses that cross over timing windows are automatically taken care of by the TPU, ensuring that they are only counted once.

In single-shot mode, the CPU reads the PULSE_COUNT at any convenient time, as long as the interrupt is handled or disabled. In continuous mode, the CPU has a time equal to the WINDOW_SIZE to read the PULSE_COUNT. Otherwise, the value is written over at the end of the next time window.



Incoming signal on TPU external pin
0 1 2 3 01 2 3 4 5 01
TPU channel initialized
Window size
Window size
*INT request PULSE_COUNT = 3
*INT request PULSE_COUNT = 5

Figure 4—*This diagram shows the frequency measurement function (FQM) in continuous mode. At the end of each time window, Int is requested and Pulse_Count is updated.*

GENERATING A PWM SIGNAL

Based on a sequence of matches, the QOM function generates a pulse train without CPU intervention. You can define a series of match values and store them in sequence in parameter RAM. When the value of a free-running counter (TCR1 or TCR2) is equal to the first match value, a match occurs.

When a match occurs, a pin action (i.e., pin is taken high, is taken low, or remains the same) takes place on the channel's external pin. The next match value is obtained, and the sequence repeats. The pin action is programmable.

The QOM function enables the TPU to output a PWM signal on any TPU channel. Setting up the QOM function for PWM output is similar to setting up the FQM function.

The CPU has direct control over the low and high times of the PWM signal via parameter RAM values `OFFSET_1` and `OFFSET_2`. These values can be changed or updated at any time by the CPU. The duty cycle ranges from 0 to 100%. Figure 5 shows the output of the QOM function used to produce a PWM signal.

To sum up how to use the TPU to control the speed of a DC motor, first the CPU sets up one TPU channel to run the FQM function and another TPU channel to run the QOM. Then the CPU starts the TPU channels.

The CPU then goes on to other tasks, and when the FQM function generates an interrupt request, the CPU services this request in a time no greater than `WINDOW_SIZE`. Next, the CPU reads the `PULSE_COUNT` and uses it as a vector to a look-up table that contains the PWM parameters. After that, the CPU then goes on to load the new PWM parameters into the TPU channel running the QOM function.

From a programmer's viewpoint, once the initialization is done, only a few lines of CPU code are required to control the motor speed. From a hardware viewpoint, only two external pins are needed.

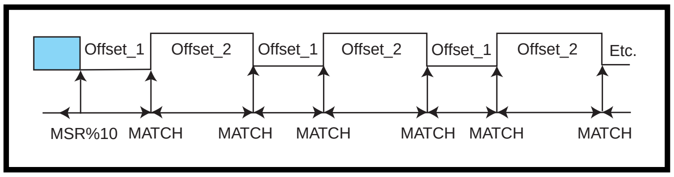


Figure 5—*Here's the signal produced by QOM function. Both the high time `Offset_2` and the low time `Offset_1` are user-programmable. The TPU produces this waveform with no CPU intervention, and the channel is set up.*

PARAMETER MEASURMENT

There are countless applications that require the measurement of distance traveled, speed of travel, direction of motion, and position mapping. The TPU can be very useful in measuring these parameters.

If you take the encoder used in the previous example and attach it to a wheel, you can use the FQM function or the PPWA function in mask set A to count the number of pulses generated as the wheel rotates.

Knowing the number of pulses per revolution and the circumference of the wheel, you can calculate distance and speed, giving you an odometer and speedometer. The TPU makes these measurement simple, presenting the CPU with the data it needs to easily and quickly calculate distance and or speed.

But, neither an odometer nor speedometer gives directional information. To get direction you need a more sophisticated encoder—a quadrature encoder.

A quadrature encoder is basically a dual encoder generating two pulse trains 90° out of phase as in Figure 6. If signal A leads signal B, this signifies travel in one direction, but if signal A lags signal B, travel is in the opposite direction.

The TPU function fast quadrature decode (FQD), on both mask sets A and G, decodes the signal from a quadrature encoder. Two adjacent TPU channels are required. Once they are selected, one channel is designated as primary and the other as secondary. Encoder output A connects to the external pin of the primary channel, and output B connects to the secondary channel's external pin.

By attaching a quadrature encoder to a wheel of known circumference and knowing the number of encoder pulses per wheel revolution, we have an odometer that also gives directional information.

The operation of the FQD is quite simple. The function maintains a 16-bit register in parameter RAM that is incremented or decremented, depending on direction of travel, whenever there's a transition on the primary channel. This register is `POSITION_COUNT` and is stored in the primary channel's parameter RAM.

Like the setting up of the FQM function, the setup for the FQD requires several steps. The first step for setting up the TPU to read a quadrature encoder is to disable the two channels by clearing their associated priority bits.

Next, write the FQD function number ($6) to both channels' function select bits. You then set up parameter RAM registers in both channels.

After that, initialize `POSITION_COUNT` to the desired start value. `POSITION_COUNT` is the primary output of the function residing in parameter RAM of the primary channel. This parameter can be written or read at any time by the CPU. A secondary output, `EDGE_TIME`, provides a time stamp by capturing the value of TCR1 at the time of transition.

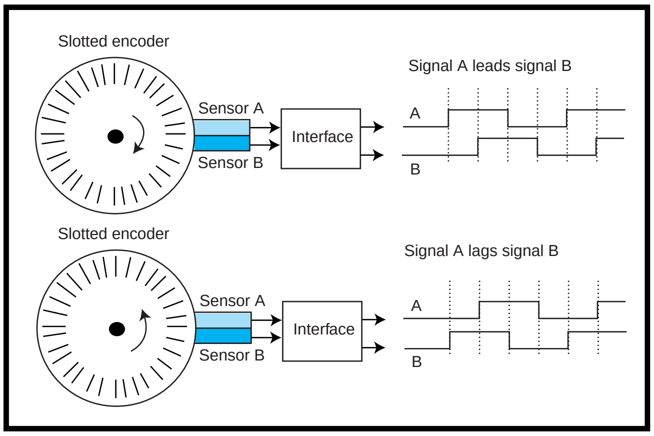


Figure 6—*A quadrature encoder is basically a dual encoder. Directional information is related to the phase relationship between signals A and B.*

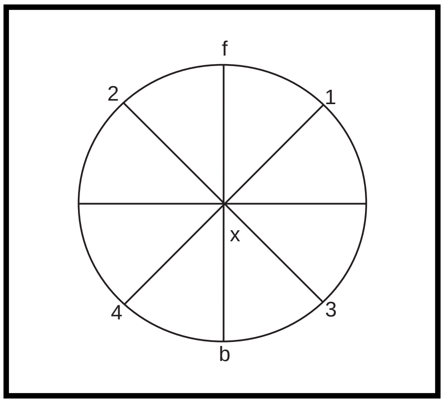


Figure 7—*A quadrature encoder can give you distance travelled and direction. But with regards to exact position, the best you can say is that you are on the top or bottom half of the circle.*

Once POSITION_COUNT is initialized, use the host sequence bits to define function operation. Select one channel as primary and the other as secondary. Then, you should choose and set the mode.

Normal mode increments or decrements the POSITION_COUNT on every valid transition. Fast mode increments or decrements the POSITION_COUNT on every fourth valid transition. Fast mode is used when incoming transitions occur rapidly and no direction change is immanent.

A host service request is issued to initialize the function, and finally, the two channels' priority bits are set to any one of the three possible nonzero values—11, 10, 01. This action enables the function and assigns priority as either low, medium, or high.

The TPU executes the initialization of the channels and starts decoding. It then updates POSITION_COUNT every time there's a valid transition on the primary channel's input pin. All the CPU has to do is read the POSITION_COUNT value in parameter RAM to obtain position information.

Note that I used the term "position information," not "position." That's because the term "POSITION_COUNT" is a bit of a misnomer. It's not really the position. It's distance with a direction. Let me explain.

Let's start at point *x* in Figure 7 and move out 1 m, setting movement towards the top of the page as forward and movement towards the bottom of the page as backwards, and always moving at right angles to the horizontal axis of the page. Doing this, you end up at either point *f* or point *b*.

The TPU would indicate in its POSITION_COUNT that you are 1 m forward or back of position *x*. Of course, the CPU would have to work out the distance based on counts per revolution and wheel circumference.

Now, if you move out from point *x* at 45° angles to the horizontal of the page, you either end up at points 1 or 2 if you travel forward, or at points 3 or 4 if you travel backwards. But, the value in POSITION_COUNT is exactly the same depending on which half of the circle you are on, regardless of whether you are at points 1, 2, or *f*, or at points 3, 4, or *b*.

Therefore, by using a quadrature encoder you can determine the distance traveled and the direction. With regards to position, all you can say for sure is that you're on either the top or the bottom semicircle in Figure 7.

To get position, then, you need another wheel and another quadrature encoder. Look at Figure 8 and you'll see two wheels with a quadrature encoder on each wheel, meaning that you need four TPU channels.

The wheels are connected together by a 1-m axial but rotate independently. In Figure 8a, the connecting axial extends 1 m to the left and is fixed at point *x*, whereas in Figure 8b, the connecting axial extends 1 m to the right and is fixed to point *y*.

In Figure 8a, by rotating the wheels counterclockwise around point *x*, you see that in one full rotation, wheel 1 travels 6.28 m and wheel 2 travels 12.58 m (i.e., 2p*r*). As you can see in Figure 8b, rotating the wheels clockwise around point *y* means that in one full rotation, wheel 2 travels 6.28 m and wheel 1 travels 12.58 m (i.e., 2p*r*).

Note that the wheels travel different distances only when a change of direction occurs, which is something that occurs continuously when traveling in a circle. If no change in direction occurs, the wheels travel the same distance.

One method for determining exact position is shown in Figure 8c. Now, let's move the wheels from position

Encoders

Encoders provide a simple and cost-effective means of obtaining position and/or speed data. There are many types of encoders, but the four most popular types are capactive, magnetic, contact, and optical. Here, I only want to discuss optical encoders—specifically, rotary optical encoders.

Rotary optical encoders translate a rotation into a pulse train. The frequency of that pulse train is representative of the rotation.

Optical encoders are made up of a light source (normally a collimated LED), a light detector (usually a photodiode), conditioning electronics, and a wheel or disk with a pattern of alternating translucent and opaque sections (see Figure i).

The photodiode output is fed into the conditioning electronics for shaping and sharpening. The patterned wheel is placed between the light source and the light detector. The encoder is mounted onto any shaft so that when the shaft rotates, so does the encoder wheel. The rotation of the encoder wheel causes the pulse train, which is representative of the shaft's rotation, as you can see below.

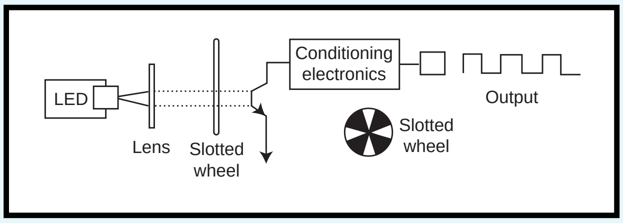


Figure i—*Rotary optical encoders, like the one diagrammed here, translate a rotation into a pulse train.*

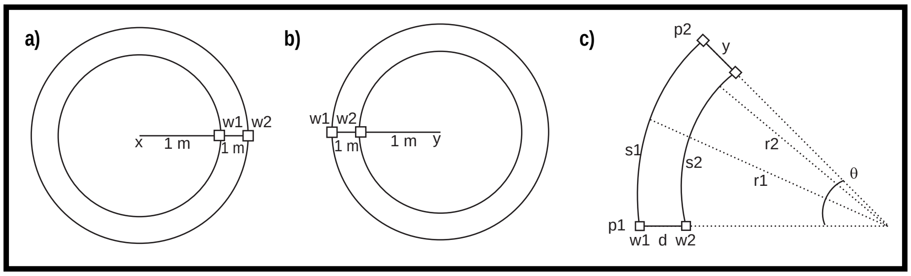


Figure 8a—*Wheel 1 (*w1*) will not travel as far as wheel 2 (*w2*) as they are rotated around* x*.* ***b****—But,* w1 *will travel farther than* w2 *as they are rotated around* y*.* ***c****—Therefore,* w1 *and* w2 *travel different distances from* p1 *and* p2*. The values shared by the TPU are* s1 *and* s2*. Because the angle is common and you know* d*, it's a matter of simple algebra to determine* r1 *and* r2 *and, therefore, the exact position.*

*p*1 to position *p*2. *s*1 and *s*2 are obtained from the POSITION_COUNT parameter values. *s*1 and *s*2 are equal to the radius in radians multiplied by the angle q (see equations 1 and 2). The relationship between *r*1 and *r*2 is given in equation 3.

s1 = r1 ´ q [1]

s2 = r2 ´ q [2]

r1 = r2 + d [3]

In equation 1 and 2, the angle is the same (q), therefore:

$$\frac{s1}{r1} = \frac{s2}{r2} \qquad [4]$$

Because we know *d* (i.e., the distance between the two wheels), we have equations 3 and 4 and unknowns *r*1 and *r*2. The determination of exact position is only a matter of algebra.

You now have a means of obtaining several parameters of travel. Because there are so many applications that require speed, distance, direction, or the mapping of position, I'll have to stop here.

The important thing that I want you to keep in mind is that these travel parameters were obtained using only six TPU channels and very little CPU time. Now you see how using the TPU's canned timer functions can increase the functionality and performance of microprocessor-based systems.

Motorola provides a wide range of timer/counter functions for the 68332 in mask sets A and G. But, even a company the size of Motorola couldn't provide canned functions for every application. So, they did the next best thing.

Using microcode, the TPU can be programmed by the user, giving the user complete control over it. This greatly expands the application field for the TPU.

In fact, the TPU doesn't need to be used strictly for timing/counter functions. Its 16 channels are completely under programmer control, and many nontiming applications can benefit from this flexibility.

Programming the TPU using microcode will be the topic of the next two articles in the series. Beginning next month, I'll look at the register structure of the TPU, the link between the TPU, the TPU microcode, and programming the TPU.

Joe DiBartolomeo has over 15 years of engineering experience. He currently works for a radar company and also runs a consulting company, Northern Engineering Associates. You may reach him at jdb.nea@sympatico.ca.

REFERENCES

T. Harman, *The Motorola MC68332 Microcontroller*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Motorola, *Time Processor Unit Reference Manual*, TPURM/AD, 1993.

Motorola, *Central Processor Unit*, 1990.

SOURCE

Microcontroller
Motorola
(512) 328-2268
Fax: (512) 891-4465
freeware.aus.sps.mot.com

FROM THE BENCH

Jeff Bachiochi

JTAG Testability

Seems that some standards don't do what they were designed for. They turn out to do more! This month, Jeff explains why JTAG is being so widely adopted by manufacturers of everything from chips to systems.

Move over Justice League, Fantastic Four, and X-Men. Here comes the Joint Team Action Group!

These guys, also known as JTAG, aren't your run-of-the-mill Saturday morning TV supergroup. They can't change the course of mighty rivers or bend steel with their bare hands. But, they did establish a solution for board testing and made it an industry standard.

Is JTAG a term you're not comfortable with? Well, it beats the official specification, IEEE Standard 1149.1-1990, or "Test Access Port and Boundary-Scan Architecture."

Testing is defined as the observation of output produced by carefully controlled input. When you're testing the components on a circuit board, you observe output behavior based on certain input stimuli. Observing incorrect behavior is as important as observing correct behavior. After all, we want to identify both good and bad products.

Identifying the cause of a bad board helps in two ways. Fault identification enables reclamation to be performed at a less technical level. And, lower reclamation expenditures keep the cost of finished goods down.

BED-OF-NAILS TESTERS

To test for manufacturing defects like missing, damaged, or misaligned devices and open or shorted circuits, testing houses were forced into using a bed-of-nails in-circuit tester.

Similar in appearance to the Indian fakir's torturous bed of nails, these testers are strategically placed spring-loaded nails that make contact with traces (or component leads protruding through the circuit board). By contacting an individual net of the circuitry, each nail enables the tester to control or monitor the boundaries between parts.

When you apply controlled stimuli to inputs, you can monitor each net for proper response. With bed-of-nails testing, input stimuli cascades throughout the board. An improper signal output by a device to a boundary (net) is passed on to the following devices, possibly looking like additional errors.

What's needed here is the ability to remove the output drive to a net and give the tester complete control of that net. But, that's not possible with bed-of-nails testing because the tester does not have a way to break the signal path.

BOUNDARY-SCAN ARCHITECTURE

JTAG proposed an architecture that provides complete control of each net by defining basic functional specifications without becoming bogged down in the specifics of design. The group didn't need to consider the future possibilities of today's devices because they left adequate breadth in the specs.

For a device to be JTAG compliant, it needs a few basic elements. Four I/O pins must be dedicated as TDI (test data in), TDO (test data out), TCL (test clock), and TMS (test mode select). They are internally connected to the TAP (test access port). The TAP controller needs to function like a predefined state machine.

One boundary-scan cell must be located between each device and its I/O pin. As well, all cells have to be internally connected to form a serial boundary-scan shift register. There needs to be an IR (instruction register) of at least two bits, and there must be a bypass register as well.

It makes sense that the first parts to implement a JTAG port were the bus latches and drivers. Because these common parts tie larger elements together, having control of the latches

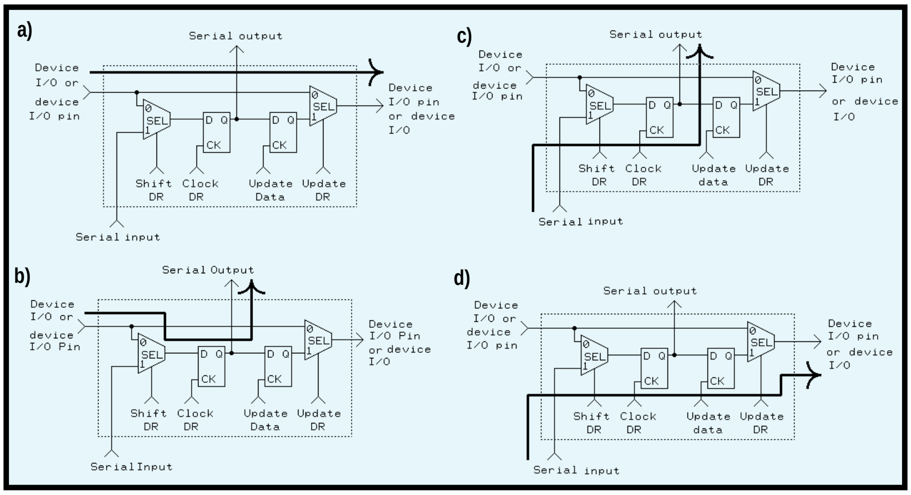


Figure 1—*These diagrams show the makeup of a simple boundary-scan cell and the four possible data paths through it—(**a**) normal operation, (**b**) sample cell input data, (**c**) shift through serial data, and (**d**) set serial data at the cell's output.*

and drivers on a circuit board isolates most circuitry. Let's look at how this arrangement is implemented in the standard 74*xx*374-latched driver.

To comply with JTAG specifications, two elements are added to the latched driver's circuitry. The first is the boundary-scan cell, which may be as simple as two 2-to-1 multiplexors (mux) and two D-type register latches. These cells are placed between a device's I/O pin and the device.

The data signal passes through the boundary-scan cell unencumbered in the default power-up mode, as you see in Figure 1a. The output mux handles this operation but enables the path through the cell to be broken under control conditions. Under these conditions, the input mux can choose the original data (see Figure 1b) or serial scan data from the JTAG TDI or previous cell (see Figure 1c).

The chosen signal is applied to the D input of the first latch. This output of this register becomes the serial scan data to TDO (or the next cell) and is applied to the second D-type register. When clocked, this register holds the original data or the serial-scan data (see Figure 1d) and applies it to the other input of the output mux.

What's the effect of the boundary-scan cell? The cell should look invisible under normal operating conditions, but once enabled, it can disconnect the flow of circuit data, sample data coming in and shift it out TDO, shift external data in from TXI, and apply the new data to the output. These actions give complete control of the data coming into and going out of a device's I/O pin, and the data coming into and going out of the device.

By controlling the data on each pin, full testing of all the interconnections between pins can be handled independently and tested in parallel with other independent nets. Because all connections between the device pin and the device are disconnected internally by the boundary-scan cell, the devices can be tested for functionality as well. Some newer devices have built-in self-testing.

The second element added to the standard driver circuitry is the TAP controller. The four required JTAG signals—TDI, TDO, TMS, and TCK—are integral parts of the TAP controller and add four pins to the standard device (see Figure 2).

The TCK input is the signal man for the TAP, sampling the TDI and TMS inputs on the rising edge and latching TDO outputs on the falling edge. TMS data determines the activity of the TAP controller.

Figure 3 shows how the TMS affects the TAP outputs. There are three main states. In the Reset state, the device is unimpeded. In the Select DR-Scan state, the data register is placed between TDI and TDO. The instruction

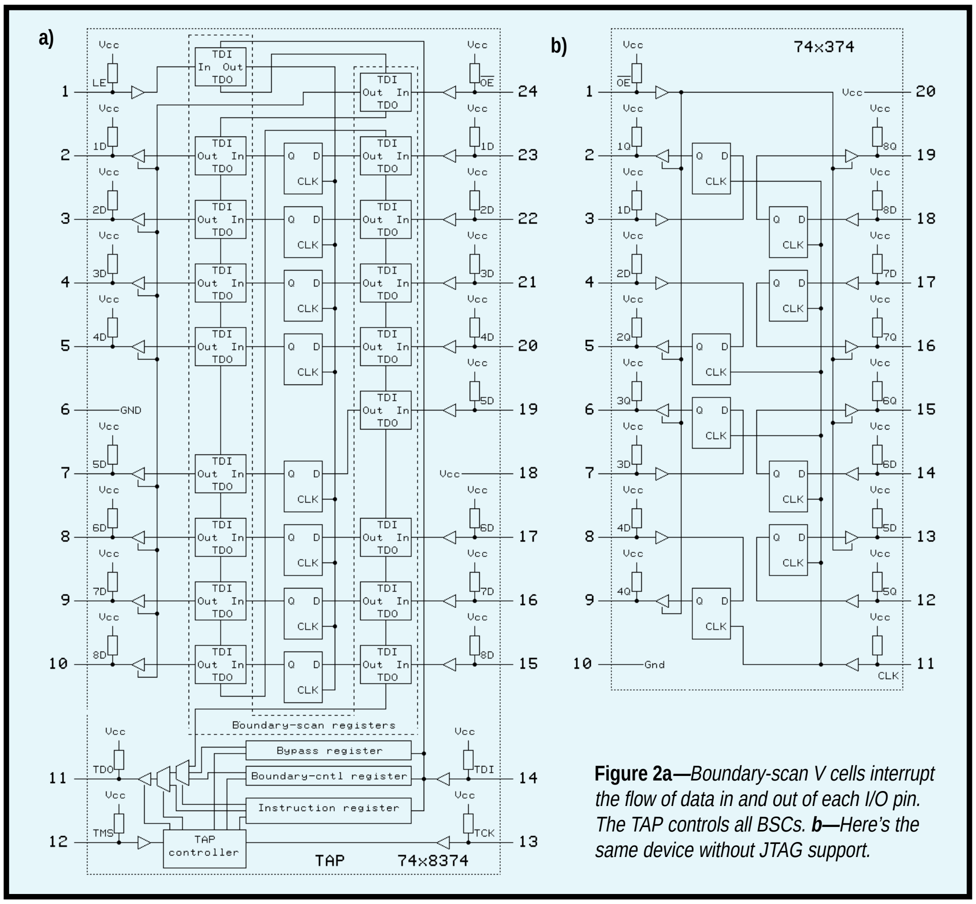


Figure 2a—*Boundary-scan V cells interrupt the flow of data in and out of each I/O pin. The TAP controls all BSCs.* ***b—****Here's the same device without JTAG support.*

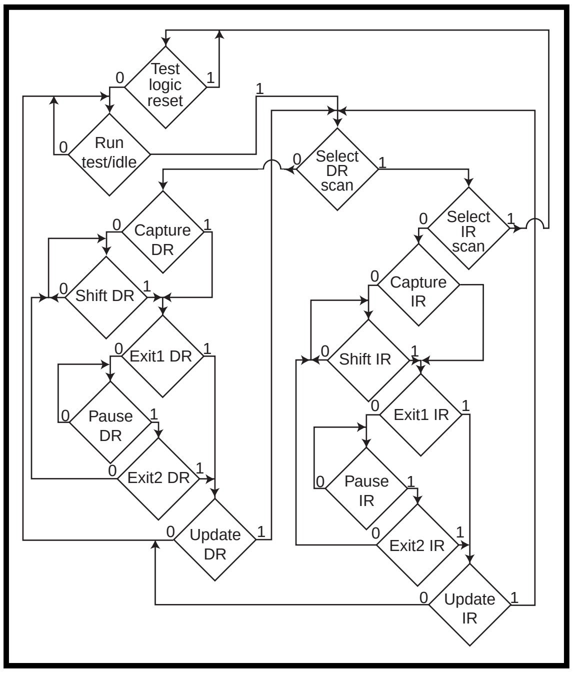


Figure 3—*This JTAG state diagram demonstrates how the TAP operates based on the TMS input clocked by TCK.*

register is placed between TDI and TDO in the Select IR-Scan state.

The IR/DR-Scan states have identical subloops. The only difference is which register the operation affects (IR or DR). The subloop captures data from the register to a shadow register and exits or shifts the data in the shadow register.

If the data shifts, it can shift again, exit, or update the register from the shadow register. Remember that external data comes from TDI shifting through a shadow register and exiting through TDO. Thus, a register can be read from and/or written to in one access.

DATA REGISTERS

A JTAG-compliant device has at least two data registers, with the simplest being the Bypass register. When it is enabled between TDI and TDO, the Bypass register clocks the data through without modification. It enables the tester to skip over this device, thereby shortening the required serial datastream.

When multiple devices are connected in a daisy chain, the resulting serial word can be quite long. Bypassing devices keeps word length to a minimum.

The second required data register is the boundary-scan cell (BSC) register. The device has an equal number of BSC registers and I/O pins, so the tester can supply data into TDI and the data is clocked through each BSC register.

The instruction register can use the data to force the BSC register's outputs into a particular state. Other instructions load the BSC registers with the data present on the BSC inputs. The tester can then shift the data out the TDO to acquire that data.

From the PCB point of view, each device on the board has its TDI daisy-chained from a previous device (or the board's TDI input pin) and its TDO daisy-chained to the next device (or the board's TDO output pin).

Optional data registers may include a 32-bit device or user identification register. Only one data register can be active at a time, and it is controlled by the instruction in the IR.

INSTRUCTION REGISTER

Because the IR is two bits wide (the absolute minimum), you'd expect four possible instructions. But in fact, only three are needed.

The `11` instruction calls for a Bypass Data register to be connected between TDI and TDO, leaving the device functioning normally. The device can then be totally bypassed, and fewer shifts (TCKs) are required to get data through to other devices.

`00 Extest` places the BSC registers between TDI and TDO, and between the device's pins and the device. This arrangement permits TDI data to be placed on the cell outputs connecting to device pin outputs. It also permits the cell input data from device input pins to be monitored via TDO.

The third mandatory instruction, `Sample/Preload`, uses a bit code left up to the vendor (but which has to be 01 or 10 in a two bit-wide instruction register). It places the BSC registers between TDI and TDO while enabling the device to function normally.

With this code, samples of the data coming into the device pins or out of the device core can be taken while the device is operating normally. Also, setup data can be shifted into the boundary-scan cells prior to issuing `Extest`.

Because JTAG requires a minimum instruction register size of two bits, the protocol leaves an open door for optional instructions using a larger resistor size. Optional instructions may include `Intest` for testing the device itself and `Runblist` for running a device self-test.

Another optional instruction, `Clamp`, sets the output pins via `Sample/Preload` and places the TDI and TDO into bypass mode. `Highz` places all output pins into a high-impedance state. `Idcode` dumps a 32-bit manufacturer device type and version code out TDO, and `Usercode` dumps 32 bits of user-defined information out TDO while the device operates normally.

Obviously, these optional instructions can't all be contained within the minimum two-bit IR, so each device has additional bits in the IR. If these bits are implemented, it's up to the manufacturer to define them.

That's where the IEEE Standard 1149.1-1990 gets its true power. It offers strict implementation of basic functions, yet it's open to potential expansion through a nonstandard instruction set. Compatible, yet expandable—that's JTAG.

JTAG PORT

Connecting to the JTAG design requires only four signals and ground. Most JTAG ports consist of a 2 ´ 5 square-pin header. Although a maximum of five wires is required, a 10-pin connector permits the interleaving of grounds within a flat ribbon cable. The intersignal interference is improved, but the pinout of the JTAG connector may be nonstandard.

By designing with JTAG-compliant devices, the complete bed-of-nails hardware test becomes unnecessary. Eliminating the need for major fixtures is an immediate cost savings. So, even though connectors on the PCB still require some level of connection for testing, the JTAG design simplifies them.

The level of testing can now go beyond simple monitoring. The test routines can pinpoint where problems occur and even suggest specific steps to correct the problem.

It's no wonder JTAG ports are popping up in many of today's designs. As you might suspect, it's not just about boundary-scan testing anymore. Indeed, JTAG might empower you as the next great superhero of testability.

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

REFERENCES

IEEE Standard 1149.1 (JTAG) Testability, Primer, SSYA002C, 1997.

Texas Instruments, *Boundary-Scan Logic*, SCTD002, 1994.

www.ti.com/sc/docs/jtag/jtaghome.htm

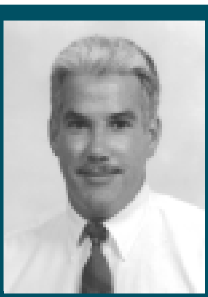
www.asset-intertech.com

Car 1451, Where are You?

SILICON UPDATE

Tom Cantrell

A Look at the IEEE 1451 Standard



Smart sensors aren't new, but the IEEE 1451 standard is. As Tom investigates this new development, he asks whether we're entering the digital age of sensors. Should you scrap all of your op-amp databooks?

Although the concept of smart sensors isn't new, as of today, the traditional signal chain, composed of sensor, signal conditioning, and A/D conversion, is proving stubbornly resistant to change. This old soldier not only isn't dying but doesn't seem to be fading away, either.

Not that there haven't been some inroads. For instance, various devices have emerged that combine a number of links in a single device, such as micromachined pressure sensors and accelerometers that incorporate signal conditioning to present a high-level (e.g., 0–5 V) output. Even brainier sensors (common examples are temperature and optical) throw in the ADC as well, delivering a digitized output (parallel I/O or pulse train) for direct connection to your favorite micro or DSP.

That's all well and good, but problems remain. For instance, there's little agreement on the digital I/O format and even less on the software machinations required to get at our beloved 1s and 0s. The result is that a seemingly simple change from brand *x* to brand *y* is likely to require a wholesale redesign of both hardware and software.

And it gets worse. So far, we're only talking about basic point-to-point lashups, which work fine for a few channels. But, what if the goal is to minimize wiring by connecting multiple devices on a single bus? Of course, the analog world has long relied on the good old 4–20-mA current loop for just such a purpose.

Ah, but the digital world is another story. Talk about putting multiple digital sensors on a single bus, and you open a closet door behind which rattle a bewildering array of skeletons. In fact, the so-called fieldbus concept is so confusing that, as far as I can tell, the whole idea threatens to collapse under its own weight.

Consider the sidebar "Sensor Control Networks," which shows you the latest (but likely not the last) version of the line card that confronts designers. Can anybody out there make sense of this, much less pick the likely winners and losers?

THE GREAT BYTE HOPE

Lurking at the edge of the ring is a new contender, the IEEE 1451 standard,

Line	Logic	Driven By	Function
DIN	Positive logic	NCAP	Address and data from transport from NCAP to STIM
DOUT	Positive logic	STIM	Data transport from STIM to NCAP
DCLK	Positive logic	NCAP	Positive-going edge latches data on DIN and DOUT
NIOE	Active low	NCAP	Signals that the data transport is active and delimits data transport framing
NTRIG	Negative logic	NCAP	Performs triggering function
NACK	Negative logic	STIM	Serves two functions: trigger acknowledge and data transport acknowledge
NINT	Negative logic	STIM	Used by the STIM to request service from the NCAP
NSDET	Active low	STIM	Used by the NCAP to detect the presence of a STIM
POWER	N/A	NCAP	Nominal 5-V power supply
COMMON	N/A	NCAP	Signal common or ground

Table 1—*The 1451.2 standard defines what is known as the TII, a 10-pin clocked serial transducer-independent interface.*

that purports to bring sensors into the digital age and then some.

I'm usually rather skeptical whenever I hear about the latest and greatest ultimate solution that's going to sweep all problems and uncertainties into a tidy little package for disposal on the ash heap of computer history. Heard it before and will hear it again, although the next big thing almost never lives up to its advanced billing.

Thus, for now at least, I'm not recommending that anyone chuck their op-amp databooks. Judgment certainly needs to be withheld until you understand what's under the hood and, even further, satisfy yourselves that the wind is blowing in a favorable direction.

However, I will say this for 1451. First, as an IEEE standard, you're given some assurance that it's both open and credible. Yes, I say "some" rather than "total" because we're all familiar with the way special interests sometimes jockey for position behind the standards veneer, attempting to gain proprietary advantage.

As for credibility, heck, even the long-buried S-100 bus got blessed by IEEE. But, I don't think 1451 has either been hijacked or is undeserving.

Another positive factor for 1451 is the relative absence of hype that often surrounds proposed standards. In fact, the effort could use a shot of PR. Few beyond those who specialize in sensors have even heard of it, much less know what it is. Compared to the puffery that often accompanies standards efforts, it appears 1451 proponents are adhering to a "walk softly but carry a big spec" strategy.

Enough background. Let's take a look at some of the details and you can judge for yourself. It's important to be informed since, à la democracy, you're going to get the smart sensors you deserve.

DOTS NOT ALL

The 1451 standard consists of four parts, "dot-1" to "dot-4" (i.e., 1451.1–1451.4), respectively. However, the history of 1451 is a little more convoluted than the nice sequential numbering scheme might indicate.

Development started a few years ago within IEEE and NIST on "A Smart Transducer Interface for Sensors and Actuators," now delivered under the auspices of IEEE 1451.2. Essentially, the standard defines what's known as a smart transducer interface module (STIM).

The key definitions incorporated into 1451.2 are a standard digital interface known as the transducer-independent interface (TII) and the format of a built-in transducer electronic datasheet (TEDS).

As you can see from Table 1, the electrical connection (i.e., TII) is relatively straightforward. It's a clocked serial interface (à la SPI) with most of the action revolving around a data clock (DCLK) and unidirectional data lines (DIN and DOUT).

NIOE is an output enable driven by the host (in 1451-speak, a network-capable application processor [NCAP]) that frames activity on the data lines. NACK is driven by the STIM to indicate that a byte transfer can proceed—that is, the host can drive DCLK.

Using NACK as a byte handshake reduces the timing burden on the STIM. The timing spec, which is shown in Figure 1, calls for all devices involved to support a minimum 6-kHz DCLK, but they're allowed to mutually agree on a higher rate.

Now, 6 kHz is well within the means of most chips' clock serial ports, so that's not a problem during a single-byte transfer. But, allowing only 166 ms between bytes might be problematic, depending on the amount of work the STIM needs to do between one byte transfer and the next. Requiring the host to hold off DCLK until the STIM gives the OK with NACK eliminates such concerns.

NTRIG can be asserted by the host to initiate a particular operation in the STIM. This action lets you establish precise timing when necessary, independent of any latency or uncertainty associated with communication and setup.

For the most part, all activities are carried out under direction of the host, except that NINT can be driven by the STIM to allow it to request service asynchronously. However, even in this case, the host response timing is not restricted. Thus, the host is in charge of its own destiny, which means that practically any device, fast or slow, can fill the role.

NSDET is also driven by the STIM as a way to signal its presence or absence. In simple configurations, it might be connected to ground on the STIM and pulled up on the host. But in other cases, the STIM may exercise explicit control to simulate a disconnect/reconnect (i.e., hot swap) cycle.

Finally, the host is responsible for providing power (up to 75 mA at 5 V) to the STIM. The STIM is certainly allowed to use an independent power source (e.g., if a high-power transducer or actuator is incorporated), but power for the interface itself should come

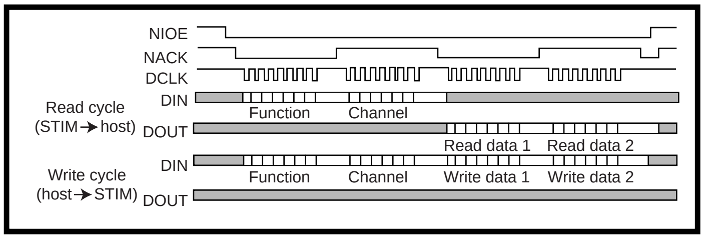


Figure 1—*The TII features a data clock (DCLK), separate data lines (DIN and DOUT), various control signals, and power and ground. NACK is used to throttle transfers at the byte level, eliminating the need for especially fast hardware to prevent overrun.*

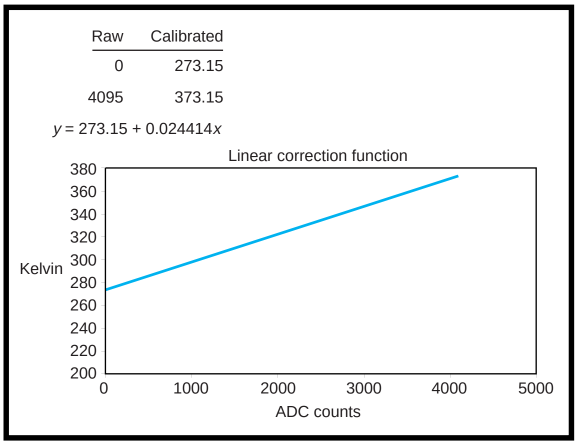


Figure 2—*1451.2 features an optional correction engine that handles calibration and standard units conversion. Here, it uses a polynomial equation to transform raw sensor output into meaningful information.*

from the host. This setup minimizes noise and ground loops.

One interesting note is the connector standard—to wit, there isn't one. Many of the early units incorporate a 10-pin (2 ´ 5) header, whereas others use a DA-15. In essence, the parties involved decided that, given the breadth of possible applications, there's more to be gained leaving users some flexibility than blessing a particular connector.

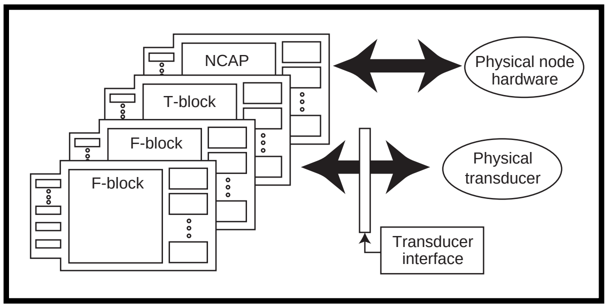


Figure 3—*The 1451.1 standard defines a software backplane for high-level objects, including NCAPs (Network Capable Application Processors), transducers (T-blocks), and functions (F-blocks).*

WHO'S ON FIRST?

The other major aspect of 1451.2 is its requirement that the STIM be the repository for the TEDS. This datasheet permits the host to find out what's on the other end of the wire—a sensors version of plug and play, if you will.

The TEDS comprises eight fields—two mandatory and six machine readable, each consisting of a length (byte count), data, and checksum. The two mandatory fields (Meta-TEDS and Channel TEDS) are machine-readable descriptions that uniquely identify the STIM and characterize the function of each channel.

The first optional field is the machine-readable Calibration TEDS. It relates to another major 1451.2 capability—the correction engine. In short, the idea is that the STIM should handle all adjustments and conversions needed to convert sensor input into something usable by the host.

Typically, correcting a sensor output involves using an equation to adjust a raw A/D reading. The 1451.2 approach supports a variety of transformations including linear, polynomial (see Figure 2), and multisegment polynomial.

The correction engine also accommodates conversion to and from IEEE 754 floating-point representations of SI (Standard International) units. STIMs from different vendors and based on completely different underlying technology can deliver exactly the same information to your program. Neat!

The next three fields are optional human-readable equivalents of the first three. Because humans come in a variety of flavors, a wide variety of languages and character sets is allowed, not just English and ASCII.

Another optional field is the Application-Specific TEDS, which is any (human readable) stuff you care to add. Finally, the spec makes provision for future growth with an arbitrary number of Extension TEDS.

The only definition associated with Extensions is an ID number that requires blessing by the IEEE. One ID number is reserved specifically to enable prototyping and experimentation prior to official assignment.

An interesting aside: Nothing says a STIM has to be a monolithic device. It's quite often the case that the sensor itself must work in an environment far too harsh for any chip.

Thus, you're free to connect the brains of the STIM and the sensor any way you please. However, although by no means enforceable, keep in mind that it's the intent of the spec that a sensor and its associated TEDS remain together until death do they part.

HANDS ON

As I mentioned, until now, 1451 action has largely been confined to sensor-industry gurus and insiders. Short of ordering the spec from the IEEE, there's been little the average embedded-system designer could do to check it out, much less get a head start on development.

I'm pleased to report that one outfit, Electronics Development Corporation (EDC), has stepped into the breach with a lineup of low-cost evaluation and development gear that puts the spec within reach of mere mortals.

In fact, EDC was an outsider until commissioned to help put together a 1451 demo at a tradeshow. They did this by tweaking their nifty CogniSense module, shown in Photo 1a, combining a PIC and signal-conditioning ASIC.

Subsequently, they jumped in with both feet and are certainly the best source I've found to help you get up to speed. Probably the best way to start is with their EDC 1451.2-KA Smart Transducer Interface kit (see Photo 1b).

Photo 1a—*The CogniSense module from Electronics Development Corporation (EDC) combining a PIC and signal-conditioning ASIC can be programmed to function as a 1451.2 STIM (smart transducer interface module).* ***b****—A good way to get started with 1451.2 is this development kit from EDC. It includes an NCAP and two STIMs as well as software.*

Sensor Control Networks

This list (see http://129.6.36.211/Home/P1451/IeeeSite/contnet.htm) highlights the dilemma designers face. So many networks, so little time.

ARCNet—Attached Resource Computer Network, developed in 1977 by Datapoint. Maximum data rate of 2.5 Mbps.

ASI—Actuator Sensor Interface, developed in Germany by a consortium of sensor suppliers. A low-cost, bit-level system designed to handle four bits per message for binary devices in a master/slave structure operating over distances up to 100 m.

BACnet—Building Automation Control Network, an American Association of Heating, Ventilation, Refrigeration, and Air Conditioning Engineers (ASHRAE) standard developed by HVAC system suppliers. It supports networking options: ARCNet, Ethernet, a master/slave token passing (MS/TP) network based on RS-485 protocol, and LonWorks.

Bitbus—developed in 1984 by Intel around the 8044 microprocessor. Features multitasking with a master/slave structure using RS-485 serial linking.

CAN bus—Control Area Network bus, developed in Germany by Robert Bosch GmbH with Intel and Philips in the early '80s for automotive in-vehicle networking. This peer-to-peer Carrier Sense Multiple Access (CSMA) system supports selectable data transfer rates up to 1 Mbps and twisted-pair, fiber, coax, and RF media. CAN is ISO Standard 11898, approved for passenger-vehicle applications. CAN-based systems were approved by SAE as Standard J1850 for American passenger cars and Standard J1939 for trucks and large vehicles. A CAN in Automation (CIA) Group has been formed in Germany to work on application issues.

CEBus—originally approved as Consumer Electronics Bus by the Electronic Industries Association. Today, CEBus goes far beyond consumer electronics. Primarily used in home automation, it supports coax, RF, power-line, twisted pair, and infrared and has provisions for fiber-optics media.

DeviceNet—version of CAN developed by Allen-Bradley. It features object-oriented software and is used in industrial control systems. It uses a four-wire (signal pair and power pair) shielded cable and supports up to 64 nodes per network at speeds up to 500 kbps at 100 m and 125 kbps at 500 m. An Open DeviceNet Vendors Association (ODVA) exists.

Foundation Fieldbus—formed from the merging of components of specifications by WorldFIP and Profibus supporters. It was formed to test and demonstrate fieldbus components to support an eventual single, universal fieldbus standard.

GPIB—General-Purpose Interface Bus, became the IEEE-488 Standard in 1978. It's more of a data-acquisition system with limited node capabilities and is used in laboratories and industrial instrument systems.

HART—Highway Addressable Remote Transducer, a network produced by Rosemount. It provides two-way digital communication atop traditional 4–20-mA loops. A HART organization has been formed.

Interbus S—open system developed by Phoenix Contact.This fast sensor/actuator data-ring–type bus uses RS-422 transceiver technology and handles analog via separate I/O modules. Up to 256 drops per network and up to 4096 digital I/Os can be supported. The network is deterministic with data throughput in the low milliseconds.

ISA SP50—organized in 1985 to develop a digital signal–based standard to complement the traditional 4–20-mA standard of the process industries. Beset by individual company interests as well as Profibus/WorldFIP polarizations, SP50 has had a long, tough trail in pursuing an acceptable fieldbus standard. But, recent progress is encouraging with the support

(continued)

For $450 you get an RS-485 network adapter (i.e., NCAP), RS-232–to–RS-485 adapter cable (connects the network adapter to a PC COM port), two CogniSense-based STIMs (one preconfigured with an accelerometer and one for experimenting with your own sensor) plus TEDS editing and STIM control/viewing software. Up to 255 of the network adapters can be daisy-chained on the '485 bus, and they, along with the STIMs, are also sold separately.

THE BIGGER PICTURE

Although you may be just getting your first good look at 1451.2, as an approved IEEE draft standard, it's well on its way to hitting the street. However, dot-2 isn't the whole story.

What about dot-1, dot-3, and dot-4? Is it the infamous spec creep in which every Johnny-come-lately starts loading stuff on to what might otherwise be a good (i.e., simple) idea?

A key point is that the numbering isn't tied to chronology. IEEE 1451.1, although it has a working group in place, is not as far along as 1451.2. And, IEEE 1451.3 and 1451.4 are only at the PAR (Project Authorization Request) stage.

Rather, the numbering is meant to reflect a hierarchical level of abstraction, from a 1s-and-0s network on the dot-1 end to a mostly analog gadget on the other side of dot-4. Here's the (by necessity, very brief at this point) story.

Hewlett-Packard has been a driving force behind dot-1, which can be considered both a theoretical and pragmatic response to the fieldbus chaos reflected in the sidebar.

On the theoretical side, the benefit of 1451.2 compatibility will be hobbled by network incompatibilities. What good is a compatible sensor if it is hidden behind an incompatible network? Software developers still face that prospect hacking away at their programs to support different networks.

As well, HP is of the belief that Ethernet, while never designed for this purpose, has a lot going for it as a fieldbus—namely, it's cheap, it works, and it's widely available. But, simply adding Ethernet to the laundry list of sensor networking schemes will do little to further their cause.

(continued)

of the Fieldbus Foundation. Profibus and WorldFIP offer eventual migration paths to any forthcoming IEC 1158-SP50 world standard.

J1850—an SAE Standard for passenger cars covering mid-speed data rates optimized at 10.4 and 41.6 kbps—rates used by GM and Ford.

LonWorks—Local Operating Network, a distributed control network developed by Echelon. It uses custom Neuron chips implementing ISO/OSI seven-layer stack protocol. It supports media like twisted pair, coax, fiber optic, RF, infrared, and power line with data rates up to 1.25 Mbps for distances up to 500 m and 78 kbps at 2000 m.

Profibus—Process Field Bus, developed in Germany and strongly supported by Siemens. It is German DIN Standard 19245. Parts 1 and 2 are designated Profibus-FMS and cover automation in general. Part 3, Profibus-DP, is a faster system for factory automation. A fourth Profibus-PA part is in preparation for process control. A number of installations are operating covering various industries. Chips and tools are available.

SDS—Smart Distributed System, developed by Honeywell MicroSwitch. This open CAN-based system uses a four-wire cable (two twisted pairs; signal and power). It supports up to 128 nodes at speeds up to 1.25 Mbps interfacing with PLCs and PCs for industrial control applications.

Sercos—a bus developed in Europe for motors and motion-control applications.

Seriplex—developed by Automation Process Control (APC) Company. This ASIC-based multiplexing system offers both peer-to-peer and master/slave communications.

WorldFIP—Factory Information Protocol, a French National Fieldbus Standard based on the three OSI control-related layers 1, 2, and 7. There are a number of installations primarily in France and Italy. Chips and products are available. FICOMP (Fieldbus Consortium), begun in late 1992, is developing board-level products and software in accordance with IEC, Fieldbus Foundation, and WorldFIP specs.

Thus, 1451.1 formalizes what any good programmer would do when faced with such a circumstance: add a level of abstraction like an object-oriented API (application program interface) between the network host and the NCAP(s) that insulates the highest-level software from the details of exactly which hardware is being used.

The concept is that a system comprises a Lego block–like collection of objects: NCAPs, transducers, and functions that can publish and subscribe data (see Figure 3).

Meanwhile, 1451.3 accommodates networking on a smaller scale, addressing applications that call for an array of transducers in close proximity to the STIM, where a big network would be overkill. Basically, dot-3 exists to overcome the technicality in dot-2 that tightly couples a TEDS with a single transducer.

Finally, IEEE 1451.4 recognizes that there's a lot of analog know-how and installed base that won't disappear overnight. By defining a mixed mode transducer interface, it plans to support transmission of basic-TEDS digital information over an analog link.

The good news is that each higher numbered spec is completely contained within the confines of a lower numbered one. For instance, 1451.1 only sees 1451.2 STIMs regardless of whether they're monolithic devices or composed of a combination of 1451.3 and/or 1451.4 devices (see Figure 4).

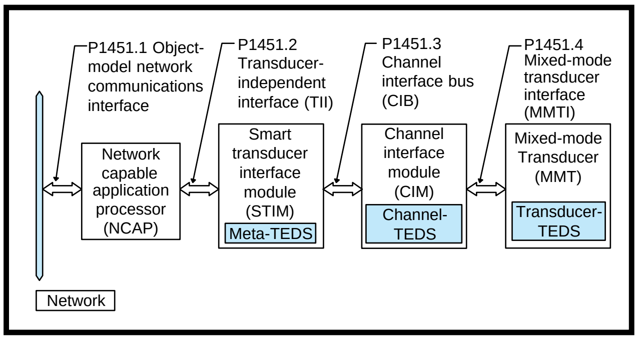

P1451.1 Object-model network communications interface
P1451.2 Transducer-independent interface (TII)
P1451.3 Channel interface bus (CIB)
P1451.4 Mixed-mode transducer interface (MMTI)
Network capable application processor (NCAP)
Smart transducer interface module (STIM)
Meta-TEDS
Channel interface module (CIM)
Channel-TEDS
Mixed-mode Transducer (MMT)
Transducer-TEDS
Network

Figure 4—*Putting it all together, IEEE 1451 defines the signal chain from analog sensor to digital network.*

HIGH-SPEED PURSUIT

For standards that purport to move sensors into the digital age, I must say that the backers haven't done enough to bring us digital types onboard. For instance, although a number of papers have been presented at Sensors Expo over the years, there's been nary a one at the Embedded Systems Conference.

Message to 1451 folks: Don't forget those who design the systems and write the software that ultimately uses, and pays for, smart sensors.

Admittedly, I've only mentioned 1451 a couple of times in previous columns myself. I hope this column serves as a worthy first step, but more investigation and hands-on scrutiny is called for. Success of the standard certainly isn't guaranteed at this point, but if it does take off, it's going to be a big deal.

Now that we know Car 1451 is out there somewhere, let's find it. ■

Tom Cantrell has been working on chip, board, and systems design and marketing in Silicon Valley for more than ten years. You may reach him by E-mail at tom.cantrell@circuitcellar.com, by telephone at (510) 657-0264, or by fax at (510) 657-5441.

REFERENCES

Electronics Development Corp., www.elecdev.com
Hewlett-Packard, www.hpie.com, www.hpl.hp.com/techreports/98/HPL-98-166.html
http://129.6.36.211/Home/P1451/ieee.htm
http://129.6.36.211/Home/P1451/docs/nist.htm
http://129.6.36.211/Home/P1451/IeeeSite/P1451.htm
www.ic.ornl.gov/p1451/p1451.html

Survey 101

Do you ever look in the mirror and ask yourself if you're doing things right? That might be the way to get answers for individual or personal matters, but I'm not egotistical enough to think that I have all the right answers when it comes to the magazine. Instead, I survey the readership every once in a while just to make sure. If you were one of the many respondents to our latest survey, I thank you for your prompt reply and valuable opinions.

What did I learn? The good news is that you still love the magazine—96% of you save every issue and our renewal rate remains among the highest in the industry. I wish there was a way to build this enthusiasm into a 100,000-circulation magazine, but I also know that the vertical specialization that instills such loyalty is the uniqueness that would be sacrificed to appeal to a larger audience. It's classic case of damned if you do…. Don't worry. I like what I'm doing and I intend to keep doing it.

So, who is the *Circuit Cellar* reader? Not surprisingly, our demographics look a lot like an electronic trade magazine. Our typical reader is a professionally trained male. Most list themselves as engineers or engineering managers, but I've noticed a considerable increase in the number of software professionals. Still, 72% say they're involved in hardware design and more than 50% claim project management responsibility. Over half are involved with software design or programming.

Their exceptionally high $85,000+ average salary is indicative of personal performance rather than inflationary spiral. Many readers have been with us since issue #1 and quite a few were around 10 years before that in *BYTE*. These engineers and entrepreneurs are now seasoned pros who still get involved in design management and design decisions. Our college reader program is lighting the way for tomorrow's designers.

I had my reasons why the survey asked what you do, what you use, and what you'd like to see. I've found that if a topic interests me then it will usually interest you. Unfortunately, interest has its limits. As a guy who identifies solder as his favorite programming language, I have to be especially sensitive to subjects that are technically significant yet boring as hell to me. I admit that I have biases, but usually if it interests me, it will interest you.

As you might have guessed, more people are working on 16- and 32-bit applications. What surprised me was that while '*x*86 and Pentiums were the dominant processors for "current use," Motorola MCU's took first place under the "future use" category. Heaven forbid that anyone out there has ever listened to my PC overkill speeches. I can only presume that competitive pressures and mandated price-performance has made engineers consider the manufacturing cost before burying the problem with massive computing power.

While I didn't specifically ask an 8-bit versus the world question, I suspect these applications still dominate. Everyone seems to agree that the most used processor in the past was the Z80, but PIC and Motorola 8-bit processors dominate present designs. That doesn't mean that the venerable 8051 has gone away. Certainly not. It's just been in stealth mode (perhaps because the market is segmented among many players). Selected slightly fewer times than Motorola and Microchip, the 8051 family has maintained a constant design-in factor from the past, into the present, and seemingly into the future. Our survey shows that variety is still the spice of life and engineers like having a lot of choices.

I also asked questions about editorial content and presentation. Over and over, the message was applications, hands-on projects, and tutorials. The topics you like are robotics, signal conditioning, analog and digital interfacing, control software and algorithms, and sensors.

Tutorials may be more important for other reasons, however. Most surveys came back with comments that addressed an issue of concern among new as well as older readers—fundamental electronics. The problem is that as microprocessors seem to be the prescribed solution for every electronic control problem (don't forget my PC overkill speech), schools are turning out BSEEs that are more like microcomputer programmers. Engineers write to us offering sophisticated program enhancements for published software and at the same time asking how to compute the gain of a single-stage op-amp in an adjacent schematic. Our hardware presentations appear to be at a lofty level because of a deficiency in the fundamentals.

The survey pointed out the need to add more articles on basic issues like analog interfacing and real-world components (for guys who think schematic diodes actually work like that). But, identifying the problem is only the first step. *Circuit Cellar* has always been a magazine written by our readers—not by advertisers or PR agencies. Here's an opportunity for you seasoned pros to pass something on that will be referenced again and again.

Surveys are wonderful tools for asking questions. The key to their benefit, however, is doing the right thing with the answers.

Steve

steve.ciarcia@circuitcellar.com