CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#104 MARCH 1999

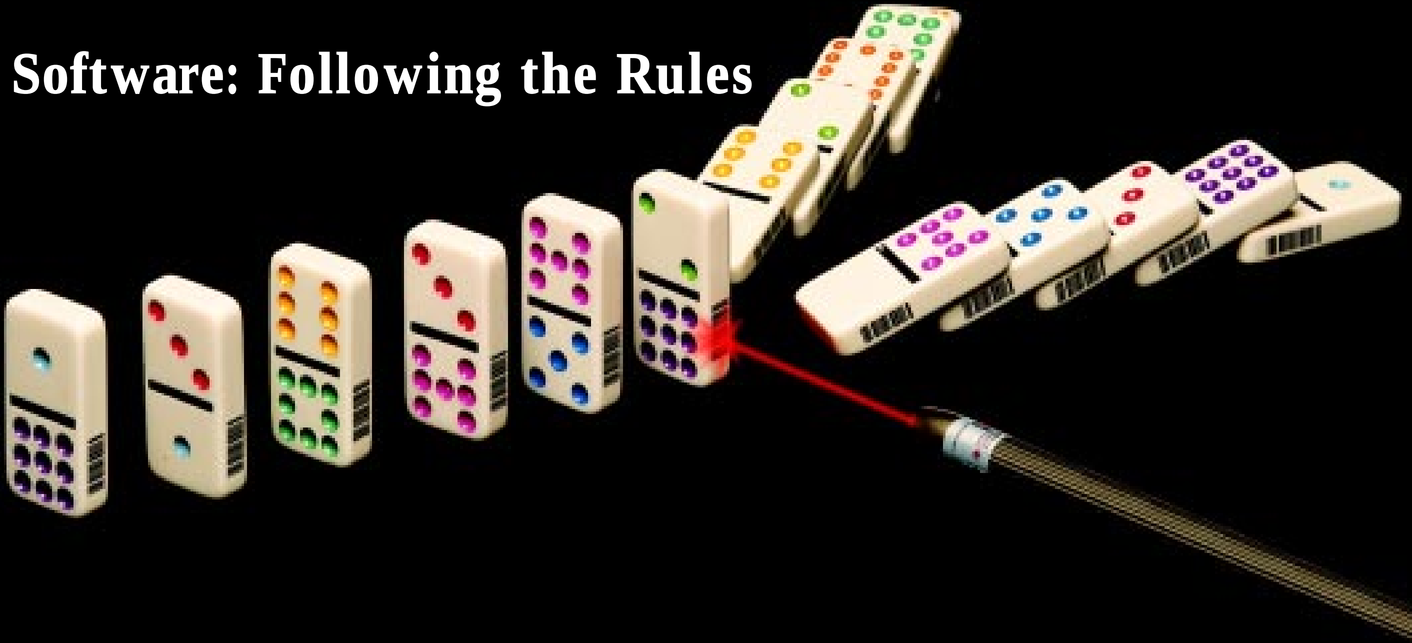
AUTOMATION AND CONTROL

Build a CEBus Plug-and-Play Relay

A Chipset Solution for Vehicle Control

Barcode Decoding

Software: Following the Rules



TASK MANAGER

Archival Alternatives

I'm not a terrible packrat, but there are some things I'll just never throw away: my fourth-grade report on "Venus: The Veiled Planet," an English project illustrating Booth Tarkington's quote "It's the land of nowadays that we never discover," a university paper detailing the vocabulary similarities of Russian and Norwegian....

Sure, I know that none of these papers will serve a great purpose for all humankind. It's just for fun. Something to show my daughter someday.

I keep a lot of *useful* information in my bookshelves, too. But, when it comes to keeping useful information, well, that's where the trouble starts.

What am I going to need tomorrow, next month, next year? It's just impossible to tell. I've got to keep it all. But then, how am I going to find what I'm looking for when the situation does arise? Sorting through all those files and books for one piece of crucial information is not my idea of a swell time.

Steve, I have to agree with you. Paper isn't dead. It's alive, well, and taking over my attic!

Seems that a lot of *Circuit Cellar* readers are in a similar bind. Whether you read the magazine in the office or at home, when you're finished with it, you put it on your shelf. According to our latest survey, nearly all *Circuit Cellar* readers keep every issue. We've passed issue 100, so you must be getting buried by now, right?

It's the curse of useful information. You want to keep it, refer to it. You want it to be there when you need it. Of course, when you do need it, you have to search through all those back issues for that one article or even that one schematic that has the answer you're looking for.

And then there's the sheer weight of it all. I can't tell you how many requests we've had over the years for back issues on CD-ROM simply because CDs are so much easier to move than boxes of paper.

So, it's happened at last—Circuit Cellar back issues on CD-ROM! It took us some time, some investigating, some investment, but I think you'll appreciate the result. This spring, two CD-ROMs will be available: one of the first five years (issues 1–30), and one of years six through ten (issues 31–89). These versions will have black and white pdf files of the scanned magazine, and most importantly, you'll be able to search them digitally.

Starting this June with *Circuit Cellar* 1998 (issues 90–101), annual CD-ROMs will be made as well. The annual versions will have full-color pdf files created directly from the layout files (not scanned pages).

As well, on every CD-ROM, we'll include the complete *Circuit Cellar* index and the relevant software for the issues on that disk.

We believe these CD-ROMs will be a boon, even if you're partial to paper. After all, it's a lot easier to stick a CD in the drive and look up that project from who-knows-how-many years ago than it is to go hunting through stacks of magazines. Once you find what you're looking for, you can then read the information off the pdf file or go pull that issue off the shelf.

Of course, CD-ROMs aren't always the answer. If you want to take a walk down memory lane and show all those projects to your kids, isn't it a bit more fun to sift through all the back issues?

Eli

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

MANAGING EDITOR
Elizabeth Laurençot

TECHNICAL EDITORS
Michael Palumbo
Rob Walker

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Ingo Cyliax
Ken Davidson
Fred Eady

NEW PRODUCTS EDITOR
Harv Weiner

ASSOCIATE PUBLISHER
Sue Skolnick

CIRCULATION MANAGER
Rose Mansella

CHIEF FINANCIAL OFFICER
Jeannette Ciarcia

ART DIRECTOR
KC Zienka

ENGINEERING STAFF
Jeff Bachiochi

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 104

12 **A CEBus Plug-and-Play Relay**
Peter House

18 **In Black and White**
A Barcode Decoder
Craig Pataky

22 **Driving in New Directions**
A Chipset for Vehicle Control Systems
Ross Bannatyne

28 **Web-Based Circuit Engineering**
Bob Perrin

62 **MicroSeries**
TPU
Part 3: Programming in Microcode
Joe DiBartolomeo

70 **From the Bench**
JTAG
Working with CoolPLD
Jeff Bachiochi

76 **Silicon Update**
Flash Forward
The 68HC908GP20
Tom Cantrell

Task Manager 2
Elizabeth Laurençot
Archival Alternatives

Reader I/O/ 6
Circuit Cellar On-line

New Product News 8
edited by Harv Weiner

Advertiser's Index/ 95
April Preview

Priority Interrupt 96
Steve Ciarcia
Paper is Dead?

EMBEDDED PC

40 **Nouveau PC**
edited by Harv Weiner

43 **Embedded Framework and Foundations**
Richard Russell

49 RPC **Real-Time PC**
Embedded RT-Linux
Part 5: Real-Time Programming
Ingo Cyliax

57 APC **Applied PCs**
ICE on Tap
Part 1: SuperTAP Setup
Fred Eady

READER I/O

MISSING PARTS

In "Mobile Environmental Control" (*INK* 100), Dan Leland described a great project that enabled his quadriplegic friend to remotely control television, VCR, and other home-control settings.

I was interested in using the technology he mentioned so I contacted Universal Electronic for the remote control. However, I was told that they no longer make the OFA8 remote. Is there a substitute model? A friend asked me to build a similar project and it's only possible if a serial remote is available.

Raymond Lagacé
rlagace@cyanic.org

I recently found out myself that Universal Electronics, who manufactures the One-For-All line of infrared remotes, decided to give up the business. I couldn't believe it! After all, they were the leader in that area.

I've had a few people contact me about this matter, and there are some options. Some OFA8s may still be available from either www.homecontrols.com or www.smarthome.com. These mail-order companies offer all kinds of home-automation technology.

Apparently, Universal Electronics licensed their technology to a company called Impact Merchandise. Impact is going to produce OFA models 8200, 8080, and 8090 only. These units have the necessary serial port but, I've never mapped the command structure so you'll have to figure out the necessary serial commands.

Although the OFA models are not included on their website, you can reach Impact Merchandise at (800) 323-9688 or (925) 373-9700 for more information on the product line.

Dan Leland

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 March Design Forum password is:

Tools

New!

- If the "Web-Based Circuit Design" article in this issue has you interested in online simulation, check out the online hardware simulators hosted in Design Forum.
- Wouldn't it be great if you could remember exactly which issue of *Circuit Cellar* contained certain projects or applications? And wouldn't it be helpful if there was an easier way to search past articles for certain topics (besides flipping through the pages of 104 back issues)? Stop wishing and start clicking! Head over to our homepage to find out how to get your searchable CD-ROM of the *Circuit Cellar* back issues. Don't forget to order before April 1 for additional savings.
- 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 contest guidelines.

Design Forum

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

Silicon Update Online: Yet Another Architecture?—Tom Cantrell
Lessons from the Trenches: Getting a Head Start on Software Development—In the Real World—George Martin
Building an Automatic Guitar Tuner: Massimo Porzio
Embedded Development Tools: Richard Russell

NEW PRODUCT NEWS

Edited by Harv Weiner

UNIVERSAL SIGNAL-CONDITIONING MODULES

The **ADAM-3000** signal-conditioning modules can process a wide range of sensor and transducer signals and provide excellent protection against the harmful effects of ground loops, motor noise, and other types of electrical interference. The modules are ideal for industrial equipment, data acquisition, or control devices that require external signal conditioning.

The ADAM-3000 modules are designed to operate in a wide range of conditions at minimal power consumption. They accept voltage, current, or thermocouple signals as input and provide either voltage or current output to the PLC or controller.

The thermocouple module features built-in linearization circuitry and cold junction compensation to ensure accurate temperature measurement. The modules use optical isolation technology to provide full three-way (input/output/power) isolation to 1000 VDC.

The modules are factory precalibrated for all standard input and output ranges. They can isolate and convert various combinations of user-customized analog and industry-standard signals. DIP-switch range selection eliminates the need to order different modules for different input and output signals.

Modules can be easily mounted on a standard DIN rail. Signal wires are connected through a front-access screw terminal, as is the +24-VDC required power. Two-wire input/output cables help simplify the wiring.

Prices range from **$70** to **$210** in single quantities.

American Advantech Corp.
(408) 245-6678
Fax: (408) 245-8268
www.advantech-usa.com

PROTOTYPING TOOL

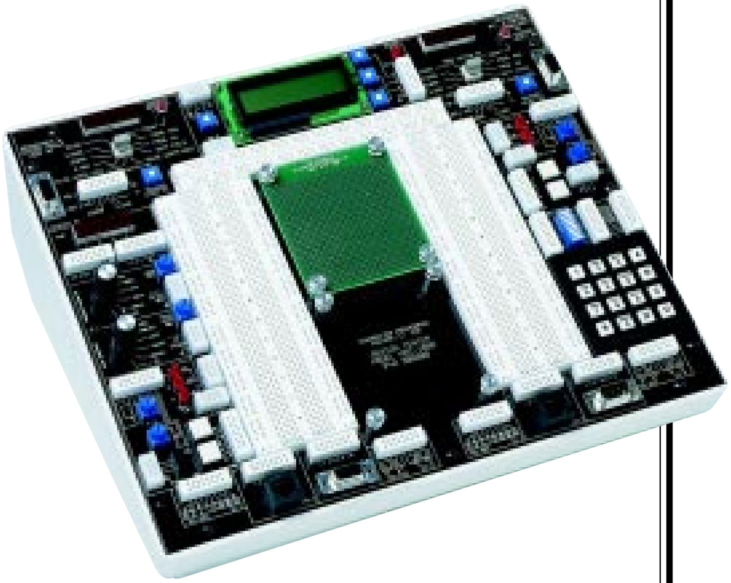
Design Center is a stand-alone self-contained design development/training tool that supports students and engineers involved in the prototyping of microcontroller-based electronic circuits. It enables easy testing of software and hardware components in designs that use embedded microprocessors.

Design Center features three adjustable DC voltage supplies, a 60-Hz reference, a frequency generator (to 1 MHz), and a pulse generator. Also included are common peripheral components and test equipment for building and debugging microcontroller-based circuits, such as two RS-485 interfaces, two RS-232 interfaces, a 4 × 4 matrix keypad, eight DIP switches, four pushbuttons, six LEDs, and a four-digit alphanumeric LED display. Not to be forgotten are the two SPDT slide switches, five potentiometers, two DE-9 connectors (one male, one female), and a 2 × 16 LCD module.

A unique adapter cradle permits current and future components in just about any package to be used. Design Center also has two identical measurement and control modules (MCM), which are intelligent built-in meters that can measure voltage, current, frequency, period, duty cycle, or pulse count of an input signal. Each MCM has an RS-232 port (with female DE-9 connector) which, when connected to a PC, enables monitoring or control of the circuits under test.

Pricing for the Design Center starts at **$997**.

Diversified Engineering
(203) 876-7408
www.diversifiedengineering.net

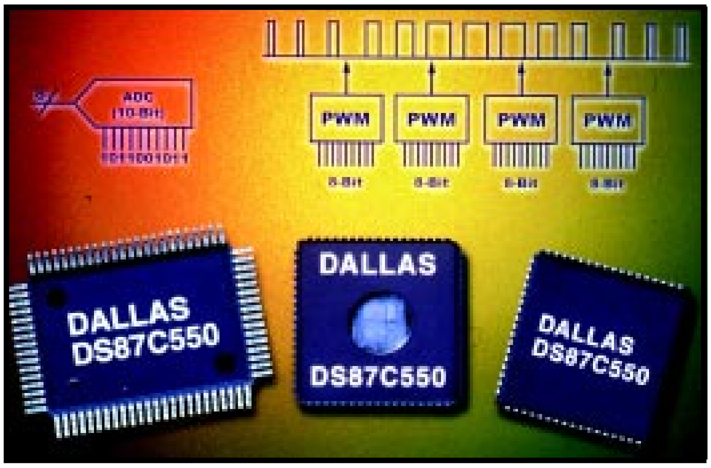


NEW PRODUCT NEWS

HIGH-SPEED MICROCONTROLLER

The **DS87C550** is a high-speed microcontroller with an on-chip ADC and PWM. It is 100% code- and pin-compatible with the industry-standard 8051. The microcontroller is ideal for battery chargers, industrial and research test or diagnostic instruments, household appliances, TVs, VCRs, stereos, computers and printers, automobile engines, and climate or environmental controls.

The DS87C550 executes 8051 instructions up to three times faster than the original 8051 architecture at the same crystal speed, thanks to an efficient instruction-execution method that eliminates wasted clock and memory cycles. As a result, an instruction typically uses four clock cycles instead of 12, thereby tripling the speed of a standard 8051. A maximum crystal speed of 33 MHz gives the DS87C550 apparent execution speeds of up to 99 MHz, and a single-cycle instruction executes in just 121 ns.



This device includes a 10-bit ADC, four channels of 8-bit PWM, 8-KB EPROM, two full-duplex serial ports, three 16-bit timer/counters, as well as four capture and three compare registers. Also featured are a watchdog timer, brownout detection, power-fail reset, and 16 interrupt sources.

Pricing for this device starts at **$12.60** each in 1000-piece quantities. It's also available in 68-pin PLCC, 68-pin windowed CLCC, and 80-pin PQFP packages.

Dallas Semiconductor
(972) 371-4448
Fax: (972) 371-3715
www.dalsemi.com

NEW PRODUCT NEWS

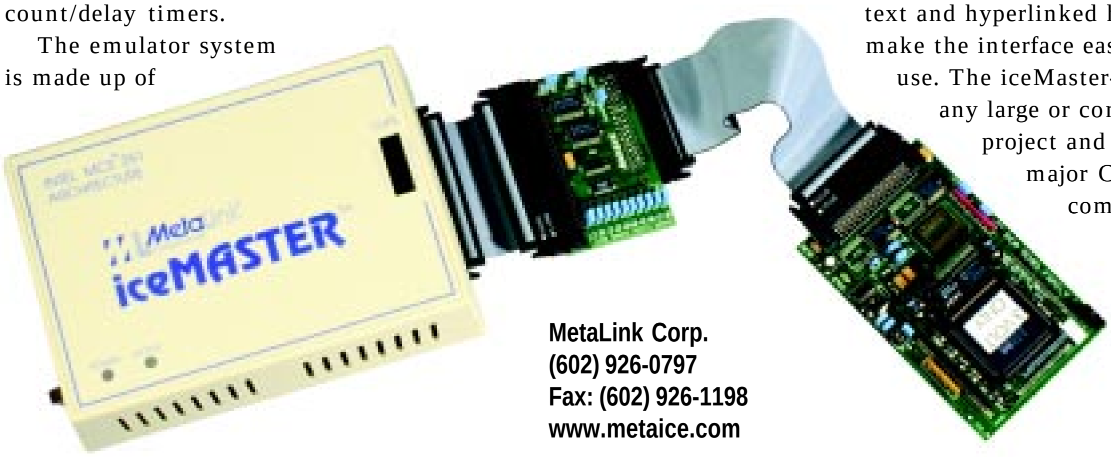
8051 IN-CIRCUIT EMULATOR

The **iceMaster-8051-SF** ICE provides full-featured real-time transparent emulation at frequencies up to 40 MHz. The unit supports bank switching and time-stamp and also includes 512 KB of code memory and 512 KB of XDATA memory. A 64-KB trace buffer reduces debug time.

The iceMaster-SF has a command window that lets the user create macro-like routines for repetitive testing. Other features include an elapsed timer and two pass count/delay timers.

The emulator system is made up of interchangeable probe cards and a PC adapter connected to an emulator base. Full portability is achieved via a 57.6-kbps serial interface between the emulator base and PC. The probe cards plug directly into a target application or can operate in stand-alone mode.

A Windows-based host interface enables high productivity. The iceMaster-SF supports most third-party assemblers and compilers. It also features full symbolic and source-level debugging. The context-sensitive hypertext and hyperlinked help system make the interface easy to learn and use. The iceMaster-SF can handle any large or complex 8051 project and supports all major C compilers. The complete system costs less than **$5000**.



MetaLink Corp.
(602) 926-0797
Fax: (602) 926-1198
www.metaice.com

NEW PRODUCT NEWS

INDUSTRIAL SMART CAMERA

The **VC Series** of smart cameras are complete machine vision systems that include up to 8 MB of RAM, 2 MB of flash memory, and a powerful DSP. The 120 × 50 × 35 mm camera features optically isolated industrial I/O (four 12–24-V inputs and four 150-mA outputs), an RS-232 serial interface, RS-170/CCIR video output, and pixel-identical sensor readout.

They feature shutter speeds from $^1/_{100,000}$ s to 20 s. CCD sensors with up to 1280 × 1024 pixels provide high resolution and sensitivity, allowing accurate measurements down to 1 µm. Progressive scanning, a color sensor, and flicker-free SVGA video output are offered as options on some models.

A normalized grayscale correlation (NGSC) software library is available for use in addressing precision registration, alignment, and robot-guidance tasks. The software can be used to create correlation-based OCR, OCV, logo/mark inspection, IC pin-one indicator detection, and golden template comparison inspection solutions.

VC Series-compatible software decodes the two-dimensional ECC200 data matrix codes used in various industries.

An integrated Windows-based development environment, which includes a C cross-compiler, real-time OS, and a large optimized image processing library is available.

Pricing ranges from **$1150** to **$3953**.

Vision Components
(617) 492-1252
Fax: (617) 492-1252
www.vision-components.de/

FEATURES

12 A CEBus Plug-and-Play Relay

18 In Black and White

22 Driving in New Directions

28 Web-Based Circuit Engineering

FEATURE ARTICLE

Peter House

A CEBus Plug-and-Play Relay

Now that prices are falling for a complete receive-only device, it's getting even easier to create a low-cost consumer electronics device. Listen up to see how Peter built this CEBus relay. The secret for its success is in the power line.

My idea of a low-cost device is one that sells for around $20. This price requires the bill of materials to run between $5 and $10, depending on the means of distribution.

Before January 1998, a complete chipset for implementing consumer electronics bus (CEBus) communications cost about $15 in 100,000-piece quantities and took two or three ICs. That price didn't include the power supply, microcontroller, or application circuit (relay or triac). Today, however, a complete receive-only device can be purchased and imported (in quantity) for about $12.

STARTING THE CEBus

The CEBus standard, originally published in 1992, was updated as ANSI/EIA-600 in 1998. It describes communication between devices in the home over five media—infrared (IR), coax (CX), twisted pair (TP), radio frequency (RF), and power line (PL).

Because we all have power lines in our houses, the power line is the most practical medium for low-cost implementation and universal usage. The CEBus standard for PL describes everything from low-level data transmission to the common application language (CAL), which is the high-level language used to send control information.

Using the CEBus standard as a guide, you can build a device that sends CAL messages to a device made by someone else. And, you have pretty good assurance the message will be understood.

Unfortunately, the CEBus standard doesn't give a clue about which messages to send at any given time. Some messages are easy to figure out, but others aren't so simple.

Well, if you want a light turned on, you send a light control message with the command to turn on the light. Easy.

But, let's say you plug in a new device from one of those home improvement places. What message(s) does it send and how does it link up with other devices? The home plug-and-play (HomePnP) standard attempts to answer these questions.

WHAT IS HOMEPnP?

Three companies with an interest in CEBus and consumer electronics started the HomePnP specification. Microsoft, Honeywell, and Intel wanted to know how a device interacts with other similar devices on the network as far as addressing and binding to other devices to share control information.

If you look at the CEBus specification, you might conclude that it tells the devices how to communicate with each other. Unfortunately, it doesn't tell devices what to say. That's where the HomePnP specification takes over.

HomePnP is an industry specification managed by the CEBus Industry Council to ensure interoperability at a system level. It's the system-level specification used to tie together individual CEBus-compatible devices.

ONE-WAY SOLUTION

When you design a device for sales in the consumer electronics market, there are three reasons to do anything—cost, cost, and cost. The choice between a receive-only device and a device that can transmit and receive goes directly to the BOM.

A device that can also transmit requires more code and more electronics. You need the code to implement

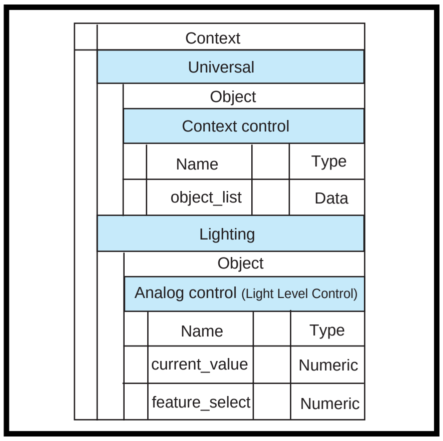


Figure 1—*Here's the CEBus context specification for the receive-only relay. Most of a normal context specification is not necessary.*

the device transmission and the added electronics to drive the communication signal on the power line. The transmit amplifier and coupling drive the cost up.

To a transmitter, a power line looks like an impedance between 10 and 40 Ω in a 100–400-kHz transmit band. The complexity surfaces when you look at the power-line environment.

Because this device will occasionally be plugged in next to a computer with a switching power supply, your output amplifier must be able to drive a short circuit without failure. These switchers have a filter on the input to prevent noise from the switcher from getting coupled to the power line. The filter has a capacitor directly across the power line, making for a low-impedance load.

If that isn't bad enough, a simple linear power supply has diodes that turn on and off 120 times per second. And, when they turn on, they connect another capacitor to the power line. Your amplifier has to drive this wildly changing impedance with very little distortion or your signal won't get through.

Did I mention the power supply for the amplifier? It has to provide current for an amplifier trying to drive a low impedance load with a signal of –5 V. How much current does it take to drive a 1-Ω load with a 5-Vp-p signal?

Fortunately, the device rarely has to transmit. So, you can use an energy-storage device like a capacitor to store current for transmit requirements.

But, adding transmit circuitry and beefing up the power supply costs about $3 on the BOM, or an additional $12 to the end user. These costs are on top of a BOM that's in the $5–7 range. By removing the transmit portion, you can sell the device for under $20.

Keep in mind that there are two ways this relay module will be used—control by a computer and control by a person. So, there are subtle implications regarding the message received and how the device handles certain message situations.

Also, a two-way device needs additional firmware to hail for and track address resources and context instance resources (I'll explain these later). This requirement complicates the user interface and requires a more detailed explanation than I can provide here.

DEMONSTRATION PLATFORM

I based this project on the EKP300 evaluation kit from Intellon. Intellon uses this kit to provide customer evaluation of the P300 or P200 series of CEBus interface ICs (see Photo 1).

The eval board connects to the power line using a special wall wart providing 10 VAC and signal coupling for the spread-spectrum signal. The board has two switches, two LEDs, a socket for a 68HC05C9A microcontroller, a P300 or P200 CEBus communication IC, a P111 power amplifier/line drive IC, and the necessary power-supply circuitry.

The output of my device is a single LED. Because I did this project for a customer demo, I didn't connect it to a relay.

USER INTERFACE

Because you save money on the transmit circuitry, you can splurge on the user interface and have a single-character keyboard (one button) and a one-pixel video display (LED).

The HomePnP specification suggests some general functionality for this interface. The button has three actions defined as press, hold, and long hold.

Press is any key press released within 0.75 s. Hold is any key press released after 0.75 s but before 3 s, and long hold is any key press lasting longer than 3 s. The long hold is ignored after 8 s in the case of a stuck switch.

A long hold places the device in setup mode. In setup mode, the device does not respond to control messages but does receive binding messages. The

Packet type (bits 0, 1, 2)	
000	IACK
001	ACK_DATA
010	UNACK_DATA
011	*
100	FAILURE
101	ADDR_ACK_DATA
110	ADDR_IACK
111	ADDR_UNACK_DATA
Packet priority (bits 3, 4)	
00	High
01	Standard
10	Deferred
11	*
Reserved (bit 5)	
Service class (bit 6)	
0	Basic
1	Extended (undefined at this time)
Sequence number (bit 7)	
Alternates each time a new packet is sent to a destination address	

Table 1—*This data link layer control field shows the bit values used to control how the packet is sent and what happens at the receiver.*

user interface LED also blinks while in setup mode. As soon as a binding message is received, the device exits setup mode and the LED stops blinking.

If another long hold is detected during setup mode, the device clears all binding memories, exits setup mode, and stops the LED from blinking. When the device isn't in setup mode, pressing the button toggles the state of the load, which gives the user local control over the controlled output.

ADDRESSING REFRESH

A CEBus device has 32 bits of address that's divided into two parts—system address and the MAC address. I'll refer to these two parts by more common names that may be familiar if you have X-10 experience—house code (system address) and unit code (MAC address).

However, unlike with X-10, these are 16-bit values and are stored in EEPROM. Rotary or DIP switches would be unwieldy and costly.

MESSAGE BASICS

There are four CEBus data link layer (DLL) services—ACK_DATA, UNACK_DATA, ADR_ACK and ADR_UNACK. I wrote this code to work with either a P300 or P200 CEBus interface IC.

The P200 only supports the UNACK_DATA service, but the P300 supports all DLL types. Even though the device cannot acknowledge the acknowledged service types without transmit circuitry, I couldn't think of a good reason to ignore these messages simply because they used the wrong DLL service.

So, if the sender of the message sent everything else correctly, the code accepts the message and controls the device. Remember this.

The CEBus protocol includes a network layer that's used for segmentation, routing, and brouting control. Segmentation breaks up long messages into many packets, with each packet being a maximum of 32 bytes long.

Routing control enables each packet to control how it is routed (i.e., moved from one network segment to another) around the network. Brouting is like routing except that it takes into account unwired media such as IR.

Because IR is line-of-sight, you'd need an IR router in each room of your house. The problem occurs if you stand in a doorway and press a key on your remote. Two routers receive the IR command and pass it on to the power line. This process causes redundant and potentially harmful results.

Brouters prevent such problems, but my device doesn't accept brouted packets. If you want to control the device using an IR remote, use an intermediary device (gateway) to translate IR commands into appropriate PL commands. The device does not permit the use of extended network services and only responds to devices on PL.

The application layer in the CEBus protocol includes facilities for rejecting duplicate messages and specifying responses from the receiving device. Because they cause no harm in my application, duplicate messages aren't rejected. If my device receives two On messages, it turns on twice. The only command where this is a problem is the toggle command.

But the primary use of the toggle command is for personal control, so I left duplicate rejection to the user. Because this is the receiving device and there are no transmit facilities, you can ignore this part of the application layer.

CAL METHODS

CAL is object oriented (with certain exceptions) and has methods that operate on objects and pass parameters (called instance variables [IVs]) based on the method and the object. My device only responds to a subset of CAL methods: SETON (0x42), SETOFF (0x41), and SETVALUE.

SETON and SETOFF turn the device on or off, and SETVALUE sets the relay to something between on and off. If the relay were a light dimmer, being set between on and off would work fine. Instead, SETVALUE sets the value of the relay between 0 and 100 where any nonzero value is On.

A CAL IV is a single variable that models the value of a particular behavior in a CEBus device. I only used the current-value and feature-select IVs. CAL methods are usually directed to an IV. For example, the SETON method, when directed to the current-value IV, sets it to a value of 100s.

My device has two objects to receive CAL messages. The context-control object handles the address assignment, and the light-control object controls the relay. There are three IVs associated with the light-control object (see Figure 1).

Each CAL object is a list of CAL IVs and represents a functional group of IVs in the CEBus device. As you'll see, you can have objects without associated IVs in a receive-only device.

A CAL context is a group of CAL objects that represents an externally identifiable group of control objects in a CEBus device. Almost every CEBus device has a universal context defining its presence on the network.

The definition takes the form of house and unit codes, group IDs, and other network management information regarding operation on the CEBus network. My device is CEBus compatible but not CEBus compliant, so it doesn't have a universal context or a real address on the CEBus network.

CONTROL FIELD

Also known as the LPDU header, the control byte instructs the DLL how to handle the packet. The control-byte bit definitions are listed in Table 1.

Because the device is limited to the UNACK_DATA DLL service, only the sequence-number bit isn't used. The service class is basic, and the reserved bit is 0.

The packet priority can be any of three two-bit combinations, and the packet type should be UNACK_DATA. If the type is ACK_DATA, ADR_ACK, or ADR_UNACK, the packet is accepted even though the P200 doesn't support these services. If the packet type is an ADR service, the sequence number is examined to reject duplicate packets.

The code doesn't compare sequence numbers. The design decisions I covered are based on the fact that duplicate messages cause no harm to the application and are indistinguishable to the user.

ADDRESS FIELDS

The destination address is the current house code of the sending device, and the unit code is the HomePnP broadcast address (0xFFFF). Any HomePnP device can receive the message, but because you can't send a response, the source address is ignored.

The network-layer header specifies unprivileged, flood or directory route, only packet (for single nonsegmented messages), no extended services, this media only, and no brouter address. Bit 5 is not affected, so the network header byte must equal 0x70 or 0x50.

The application-layer header specifies basic one byte fixed, explicit or implicit invoke, and a don't-care invoke ID. This means bits 0–3 are don't care and the upper nibble must equal 0xE.

RECEIVED MESSAGES

My device can receive messages that fall into either setup or operation messages. The setup message takes the form of a macro sent to the context-control object of the universal context and includes the message DE 21 01 93 31 F5 EC 08.

Here's how to interpret the message: all instances (0xDE) of the lighting-control context (0x21) the context-control object (0x01) macro 20 (0x93), with parameters instance 1 (0x31) and (0xF5 0xEC) type 8 (0x08). The 0xDE byte is a CAL token and uses all instances of the following object.

Macros are defined using the value of 0x80 to mean macro 1. Incrementing to 20 leaves 0x93 as macro 20. The macro system is documented by the CEBus specification and the meaning of macro 20 is specified by HomePnP.

The instance is any instantiation of the lighting-control object that is controlled and represented by the ASCII value of the desired instance. The 0xF5 is a delimiter, and the EC is a CAL token indicating that a literal expression will follow. Because my device is receive only, the message type 8 is defined in the HomePnP specification and is not used.

When the relay control is in the setup mode and this message is received, the relay control stores the destination house code and the lighting-context instance in a look-up table.

When future control messages are received, their destination house code and instance value are compared to the stored value. If the new message matches the stored value, the control message is accepted. So, it takes three bytes to make a control association.

CONTROL MESSAGES

A control message can control the relay using either the current value or feature select. The current value can be set to any value between 0 and 100, where 0 represents off and any other value is on. This range is necessary for the current value to be compatible with a dimmable load control.

A1 21 02 45 43 F5 30 turns off the relay using the current-value IV. This message is read: instance 1 (0xA1) of the lighting-control context (0x21) light level control object (0x02) SETVALUE (0x45) current value (C or 0x43) to (0xF5) 0 (0x30).

The instance value is added to 0xA0 to create the instance specifier for the context object to follow. SETVALUE sets the value of a CAL IV. C or 0x43 represents the current-value IV, and 0xF5 is a delimiter character followed by the new value for the IV specified by the ASCII character for 0.

To toggle the relay using the feature-select IV, use A1 21 02 45 66 F5 31 30 30. It reads: instance 1 (0xA1) of the lighting-control context (0x21) light level control object (0x02) SETVALUE (0x45) feature select (f or 0x66) to (0xF5) 0 (0x30).

The feature-select IV is represented by f or 0x66. 0xF5 is a delimiter character followed by the new value for the IV specified by the ASCII characters for 100. The feature-select command changes the state of the relay and is the only command I know of (for my device) that is affected by duplicate messages.

A1 21 02 42 43 also turns on the relay using the current-value IV. It reads: instance 1 (0xA1) of the lighting-control context (0x21) light level control object (0x02) SETON (0x42) current value (C or 0x43). SETON sets a CAL IV to an implicit value determined by the application. For the lighting context, the current value is set to 100.

Note that in the macro 20 message, the instance is specified as an ASCII value, and in the control message, it is added to 0xA0. The instance is specified that way (although it doesn't seem to make sense) because the specifications had many contributors and evolved over more than 10 years. Thanks to all those contributors, the standard covers almost everything, but not without minor inconsistencies in the implementation.

FEATURE SELECT

Feature-select messages are sent to the feature-select IV of the light level control object of the lighting-control context. These messages are like macros because they have defined functionality and affect the current value. Table 2 lists feature-select values.

A1 21 02 45 66 F5 31 30 30 changes the state of the relay using the feature-select IV and is read: instance 1 (0xA1) of the lighting-control context (0x21) light level control object (0x02) SETVALUE (0x45) feature select (f or 0x66). SETVALUE sets the value of a CAL IV. A value of 100 for feature select should toggle the state of the relay.

THE SOFTWARE

The software is written in 6805 assembler and the complete code is 1454 bytes long, plus interrupt vectors. The evaluation board is designed around a 68HC05C9A, so the code takes ~10% of the possible program memory.

I made one modification to the board for this project. I originally intended to control a triac and made a circuit to get the zero crossings from the power line. I used a simple RC network to connect the 60-Hz signal to the input-capture input.

Timing in the module is based on the 120 interrupts/s from that source. In retrospect, I should have used the timer to generate 100-ms interrupts. After all, this project is a demo and the triac might have been a distraction.

Seven source-code files make up the project: main.asm, light.asm, timers.asm, CENode.asm, subs.asm, e2subs.asm, and GLEquates.asm. main.asm is the main program loop including RAM definitions, general initialization, and interrupt vectors.

light.asm contains the message parser and the code to control the relay. timers.asm contains the code for the '05 onboard timers which also includes the input-capture code. CENode.asm is the initialization, messaging, and interrupt routines for communicating with the CEBus Node or Px00 IC.

subs.asm has a couple of routines that didn't fit anywhere else—a timer delay loop (used during power-on) and an ASCII numeric number parser for getting the ASCII numbers out of the HomePnP messages. e2subs.asm has all of the interface code for the EEPROM memory used to store the house codes and lighting context instance numbers. GLEquates.asm contains all of the bit, byte, and register definitions.

Because of the project size and the compiler's speed, I used the assembler to do the linking and bypassed a separate link step. The source files are included in the main.asm file so no linking takes place outside of the assembler program. Unfortunately, this untested (but documented) mode of the assembler led to the following.

ASSEMBLER ERRORS

Every now and then, I made a simple change to the code and nothing would work. The change could have been as simple as adding a NOP!

I isolated the problem to an assembler bug that caused the first byte in a subroutine to be swapped with the last byte in the code immediately prior to the subroutine. I finally identified this bug by tediously comparing the listing and .s19 files.

Fortunately, only the object file was incorrect. Placing NOPs in strategic places was a good workaround.

Unfortunately, Motorola no longer supports this assembler, which is only two years old. Motorola tech support said they'd never heard of this bug when using the separate linker program after assembling to object code.

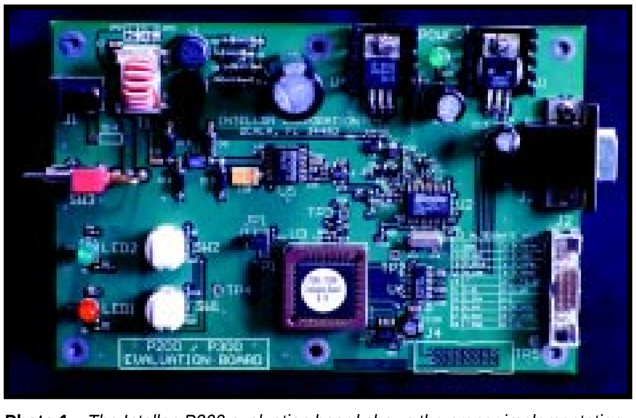


Photo 1—*The Intellon P300 evaluation board shows the proper implementation of the power-line communication ICs in an easy-to-probe package.*

Feature-select ID	Description	Action for relay module
0	No feature selected/stop feature	None
1	Ramp brightness to maximum	Set C to 100
2	Ramp brightness to minimum	Set C to 0
3	current_value->saved_and turn off	Set C to 0
4	current_value->saved_value and ramp off	Set C to 0
5	Turn on to saved_value	Set C to 100
6	Ramp on to saved_value	Set C to 100
7	Turn on to local brightness level	Set C to 100
8	Ramp on to local brightness level	Set C to 100
9	Set current brightness to minimum	Set C to 0
10	current_value->saved_value and flash	Flash at 1 Hz
11–99	Reserved for future use. Values >99 are available for manufacturer use.	
100	Toggle—if 0, then set to 100. If nonzero, then Toggle set to 0. Note: De facto standard; used by some manufacturers and not defined in standard.	

Table 2—*The feature-select instance variable represents built-in functionality for this lighting-related device.*

THE PX00 INTERFACE

The interfaces for the P300 and P200 ICs are identical so I'll refer to them as the Px00. The hardware interface to the microprocessor uses six lines—reset, chip select, serial clock, serial data in, serial data out, and interrupt.

The host controls all transactions and each byte transferred generates an interrupt by the Px00. The serial lines and the chip select make up an SPI-like interface that can be clocked up to 2 MHz. The maximum rate that this '05 can run is 500 kHz, and a byte takes only 16 µs to transfer.

It may seem like 2 MHz would be four times faster, but because there's an interrupt for every byte transferred, the interface can't run much faster. When the code initiates an SPI read, the read is finished in a couple of spins around a spin loop while waiting for the transfer complete flag to be set. This can be done with a couple `NOP`s because the transfer is completed in hardware and always finishes in the same amount of time.

Each message is made up of a command byte, length byte, and one or more data bytes. I use these messages to communicate with the Px00: layer management write (`LW`), interface read (`IR`), read receive header (`RRH`), control write (`CW`), and packet receive (`PR`).

`LW` initializes the Px00. Because this application uses the simplest mode of operation for the Px00, it has one parameter to set the mode. Other parameters are passed as nulls. `IR` interrogates the Px00 to determine its status.

`RRH` reads the message header so the microprocessor can determine if the destination is correct. `CW` follows `RRH` to let the Px00 know what to do after the complete message is received. And, `PR` retrieves the complete message from the Px00 after it's received.

The Px00 is always host driven so you don't have to poll it, which would waste valuable communications bandwidth over the serial data link. When you have the chip-select line active, each interrupt is part of the current communication session in progress.

If the chip-select line isn't active and the microprocessor receives an interrupt, it's treated as a request for service and the microprocessor responds with an `IR` to find out what's going on. This is called an attention sequence. Because there's no need for polling with the Px00 parts, polling should be avoided.

20 MS IN THE LIFE...

When the device is powered on, the host sends an `LW` to initialize the Px00. The next step is to wait for an incoming message or user action. For now, let's focus on an incoming message.

A typical CEBus message of the proper length to control this relay module is ~20 ms long. The preamble and header last for ~10 ms, and the data makes up the rest. When the Px00 receives the last data in the header, it generates an interrupt to let the host know something is going on.

The host responds with an `IR` and determines if a header is available for transfer. It uses `RRH` to get the header and checks the header contents for the correct destination address.

If the destination address House Code was previously stored by receipt of a 20-ms message and the Unit Code is the HomePnP broadcast address, the host sends `CW` to tell the Px00 to interrupt again when the rest of the message is complete. If the address doesn't match, nothing is done and the Px00 does nothing when this message is completely received.

When the message is complete and if the Px00 was instructed by `CW`, an attention sequence is generated and the host responds with an `IR`. The result of the `IR` indicates a received packet, and the host uses `PR` to get its contents. The incoming message is then parsed and the relay controlled.

DEVELOPMENT TOOLS

Development was accomplished using Premia's Codewright editor, a Motorola EVS development system and assembler, and an HP 54645D scope.

I connected the six lines to the logic analyzer inputs and triggered on the chip select. With one of the analog inputs on the power line, I could see the incoming packet and all of the communications between the microprocessor and Px00 in one screen capture, which simplified debugging and made quick work of interface timing issues. I could also zoom in and see bytes transferred serially over the SPI-like interface, which was clocked at 500 kHz.

So, as you can see, designing the hardware and firmware is easier than ever for a high-performance appliance control using the power line for communications. ■

Peter House spent four years at Intellon as an applications engineer working with power-line communications. He is now a contract engineer and consultant specializing in applications where firmware meets hardware. You may reach him at peter.house@iname.com.

SOURCE

EKP300
Intellon
(352) 237-7416
Fax: (352) 237-7616
www.intellon.com

FEATURE ARTICLE

Craig Pataky

In Black and White

A Barcode Decoder

Decoders are plentiful and you can certainly buy one, but sometimes you need to build your own for a custom application. That's what happened to Craig. Listen up as he explains how to make sense of the world in black and white.

From the production line to the checkout line, the barcode reigns supreme.

Look around. There's one on every container, wrapper, and can. Open your computer and you see barcodes on every PCB, disk drive, and power supply. You probably even have one on your driver's license.

A barcode is present on any item that must be read quickly, accurately, and economically. Other technologies exist, but nothing is more cost effective than applying a barcode sticker. And the proliferation of inexpensive scanning equipment practically guarantees the barcode a long and glorious future.

Although there's no shortage of off-the-shelf scanners, I sometimes need to integrate a decoder into a custom application. For example, one client was using a PC-Lite in the field to log rainwater samples. He would go to the site, find the vials, type in the serial numbers, and enter the results of various quality tests. Naturally, typing in all the data was slow and prone to error.

You may encounter such problems, too, so enough with the introduction. Let's get to work and create a decoder of our own.

LINGO

Every industry has its own jargon, and barcode is no different. So, before I begin, a few general definitions are in order.

First, each different type of barcode is called a symbology, and there are numerous symbologies out there (e.g., Code 39, Code 128, Codabar, UPC, and I2of5). Each symbology has its own niche in one industry or another.

The black bars and white spaces that make up every barcode are called elements. It takes a certain number of elements (the exact number depends on the symbology type) to represent a single character.

A complete barcode of any symbology type is called a symbol. On the cover of this issue, you'll see a UPC symbol in the lower left corner. At the bottom of Figure 1 is a Code 39 symbol.

Every symbol has a leading white space area called a quiet zone. Like a pause between sentences, the quiet zone helps a decoder pick out the symbol from its surroundings.

An input device converts the black bars and white spaces into discrete logic levels called video and feeds it into a decoder. The decoder interprets the video and generates characters that you and I can understand. A decoder is always based around a microprocessor, be it a simple PIC or a Pentium.

CODE 39

Here, I'd like to focus on Code 39, which is the de facto standard for most industries around the world and is required in many government barcode specifications. Unless your business is strictly retail, I'm certain you'll run across Code 39.

Code 39 is popular because it represents both text and numbers (i.e., A–Z, 0–9, +, –, ., and <space>). Figure 1 shows all the encodation patterns for the Code 39 symbology.

Each Code 39 character is made up of five bars and four spaces, making a total of nine discrete elements. Of these nine elements, three are always about twice as wide as the others. The placement of the wide elements determines which character is represented.

Because Code 39 has only two element widths—wide and narrow—a binary translation comes naturally. Simply think of every narrow element as a 0 and every wide as a 1.

Using Figure 1, you can easily determine that an encoded letter A is represented by the pattern 100001001b or 109h. Similarly, the pattern for B is 001001001b or 49h.

A Code 39 symbol always begins and ends with an encoded asterisk. Referred to as the start/stop code, this character frames the encoded data. You can think of the asterisk as a preamble and closing that lets the decoder know where a Code 39 symbol begins and ends.

A	A	L	L	W	W	8	8
B	B	M	M	X	X	9	9
C	C	N	N	Y	Y	0	0
D	D	O	O	Z	Z	$	$<
E	E	P	P	1	1	/	/ >
F	F	Q	Q	2	2	+	+
G	G	R	R	3	3	–	-
H	H	S	S	4	4	.	.
I	I	T	T	5	5	<spc>	>
J	J	U	U	6	6	%	%
K	K	V	V	7	7	*	*

CCI

CCI

Figure 1—*Here's the complete Code 39 character set. You may want to enlarge it on a copier so you can better see the ratio of wide to narrow elements.*

MAGIC WAND

Now that you can decode a symbol with your own eyes, it's time to give some specialized sight to a decoding platform. There are different types of input devices, but for this project, I used a simple wand. Most wands look like a thick pen and have three signals—+5-V power, ground, and video out.

The internal operation of a wand is rather straightforward. Light from the tip illuminates the symbol, and an internal sensor converts the black bars and white spaces into logic levels.

You can probably build your own wand with an LED, Schmitt trigger, and phototransistor, but I recommend buying one. Hewlett Packard's HBCS-A500 operates at 5 V and draws a mere 5 mA. I bought mine direct for $110, which I consider to be a small price for a solid product.

INTO THE PC

The decoding platform I chose to develop for is a '486 running DOS V.6.2. It seemed to be the best middle ground between the embedded world and Windows NT.

My hope is that anyone can quickly adapt a DOS example to suit their needs. Besides, it's a safe bet that you have a PC at your disposal, so here we go.

Wands don't come ready to connect to a PC, so I constructed an interface. I first created an adapter for the parallel port by connecting the wand's ground to parallel port pin 25, video, to pin 15.

For power, I used a 5-V regulator connected to a 9-V power lump on the wall. However, an AC power converter eliminates any possibility for portable applications, and I wouldn't inflict a battery pack on anyone.

A while back, I read some documentation about the comm port and noticed that each pin can source 10 mA—more than enough to drive my 5-mA HP wand. Eureka! I cast aside my AC converter and made the interface depicted in Figure 2.

To apply power to the wand, I set DTR high (+12 V). The 100-Ω resistor, in series with DTR, works in conjunction with the wand's internal resistance to yield the required 5 V.

Here's something to keep in mind: When you're constructing your own interface, don't overlook the protection diode or you may damage the wand.

SOFTWARE SIDE

With the hardware complete, it's time to dig into software. For this application, I used Borland C V.3.1. You can use another compiler, but my examples and source code may not compile without modification.

Whatever compiler or language you choose, the source code is still your best resource for understanding the decoding process. Nevertheless, I want to cover some important areas that will make the source code more readable.

Any given comm port is controlled by 10 programmable one-byte registers, all of which are accessed through port addresses 3F8h–3FEh for COM1, or 2F8–2FE for COM2. Of these registers, the decoder is only interested in two.

Apply power to the wand by writing a 1 to 3FCh (COM1) or 2FCh (COM2). If you're probing DTR, you should see it jump to +12 V. Turn off the wand by writing a 0, and DTR drops to –12 V. With DTR set to +12 V, you can get the status of the video line by reading bit 7 of port address 3FEh (COM1) or 2FEh (COM2) (see Listing 1).

Do this several times in a loop as you move the wand over a printed page. A little experimentation shows that bit 7 is set when the wand is on a black region and clear when it's on white. Note that lifting the wand away from the white paper produces the same effect as scanning a black region.

When scanning barcodes, most users place the tip of the wand on the quiet zone to the left or right side of the code and then move it across the symbol. When the scan is complete, the user lifts the wand and gets ready for the next scan. The initial placement of

Listing 1—*Sampling the video couldn't get any easier. Embedding the code to set RTS is a convenient way of ensuring that the wand is always enabled.*

```
#define COM1       0x3F8    // base address of COM1
#define WHITE      0        // bit 7 is low on white
#define BLACK      0x80     // bit 7 is high on black
BOOL SampleWandVideo(void)
{
  BOOL Data;
  outp(0x3FC,3);            // be sure to raise RTS!
  Data=inp(0x3FE);          // sample the data
  if ((Data&BLACK)==BLACK)  // if we're on black, return false
    Return (FALSE);
  else return(TRUE);        // otherwise on white, return true
}
```

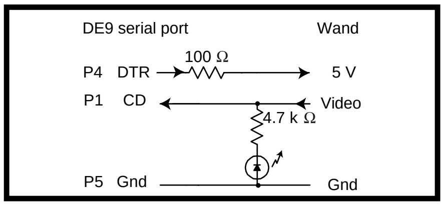


Figure 2—*Because the PC serial port is rated to source 10 mA per pin, I can tap more than enough power off DTR. As a bonus, this unorthodox interface leaves the Rx/Tx lines free for other applications.*

the wand on the quiet zone provides the decoder with an easily recognizable moment to begin digitizing input.

As the wand moves across the symbol, the decoder records the time spent on each black or white element. The timer that tracks the element widths is usually nothing more than a loop counter. As the decoder hits a black-to-white or white-to-black edge, it saves the count value, resets the counter, and waits to hit another edge.

The wider the element, the more counts it takes to cross it. If the counter overflows or exceeds a processor-dependent value, the wand has probably been lifted from the paper and you can begin decoding the buffer.

PATTERN MATCHING

Next, you convert the buffer of counts into discrete element widths. As I mentioned, all Code 39 characters are made up of nine elements—six narrow and three wide. To determine if an element is wide or narrow, compare its width (in counts) against the average of its eight closest neighbors.

If an element is greater than 1.4× the average, it is definitely wide. By substituting 1s and 0s for wide and narrow, you can build the pattern for a look-up table.

You may wonder, "Why compare against the average of the eight closest neighbors? Why not take the average of all sampled elements and compare against that?" I fell into this trap at first, and wound up with a decoder that didn't work at all.

Fact is, you tend to accelerate as you move the wand across the symbol. So, a narrow element at the start of the symbol may be 10,000 counts, but one at the end will be a miniscule 1000 or even 100 counts. By averaging the element widths on an as-you-go basis, you can factor out most of the error.

When the counts are decomposed into wide and narrow elements, you chunk through the buffer, comparing the binary patterns against a look-up table and appending the decoded characters to a string. That's it!

In the past, you might have resorted to clever look-up methods like hash tables. But, even the slowest processors today provide more than enough horsepower to justify a brute-force sequential-search approach.

CHECK-OUT TIME

For clarity, I focused on Code 39, but the same principles apply to any symbology type: collect buffer, convert to a binary pattern, and perform a brute-force match. The only thing that varies is the patterns to look for.

I built this decoder on a PC, but you can just as easily apply it to an embedded platform. In fact, an embedded implementation is easier because you have more direct control of pin I/O and timers. Good luck!

Craig Pataky is a systems engineer with over nine years of experience ranging from simple embedded programming to OS design. You may reach him at craig@logical-co.com or visit his website at www.logicfire.com.

SOFTWARE

Source code for the barcode decoder is available via the Circuit Cellar web site.

REFERENCE

R. Palmer, *The Bar Code Book*, Helmers Publishing, Peterborough, NH, 1995.

SOURCES

HBCS-A500
Hewlett-Packard
(800) 235-0312
(408) 654-8675
Fax: (408) 654-8575
www.hp.com

Borland C V.3.1
Inprise Corp.
(800) 457-9527
(831) 431-1000
Fax: (831) 431-9527
www.inprise.com

Driving in New Directions

FEATURE ARTICLE

Ross Bannatyne

A Chipset for Vehicle Control Systems

Chipsets cut down on development time because there's less of a need to evaluate interacting components. They're even more valuable if they can be applied to a number of related products (ABS, TCS, and ESP come to mind…).

Antilock braking systems, traction-control systems, and electronic-stability programs are all closely related and can often be standardized on one control platform.

In this article, I explain how a chipset optimizes this solution. I also discuss the electronic implementation and basic control strategy of each system.

A chipset's real value is illustrated when it can be applied across several products that are related, as in the case of ABS and vehicle control systems. Chassis-control systems are often based on a single platform but vary in features and functionality. Basic system requirements can be met with a chipset, and interchangeability with pin-for-pin–compatible variants permits upgradability to higher performance systems.

BRAKE TIME

Antilock braking systems (ABS) monitor four wheel-speed sensors to evaluate wheel slippage. Slip is determined by calculating the ratio of individual wheel speed to vehicle speed, which is continuously estimated from the four wheel speeds.

The control system has to maintain maximum possible wheel grip on the road (without the wheel locking) by manipulating the hydraulic fluid via electronically controlled solenoid valves. The relationship between wheel slip and the coefficient of friction on the road surface is shown in Figure 1. By manipulating these solenoid valves to maintain, reduce, or increase pressure at the wheel cylinder, the brake pressure is controlled at each wheel.

The limiting factor on the control cycle time is the actuation time of the hydraulic solenoid valves (~10 ms). Applying Nyquist's theorem determines that the control loop must execute in ~5 ms. In that time, each wheel-speed input must be processed, wheel slip determined, and the appropriate output actuation signals sent to the valves and hydraulic pump motor.

Figure 2 illustrates the relationships between wheel speed, solenoid valve pressure control, and hydraulic pressure at the wheel cylinder. Note that such characteristics exist for each wheel.

At $t = 0$, the driver applies the brakes. The speed reduces and the hydraulic pressure increases until $t = a$, at which time the vehicle speed falls below the control-reference speed.

The control-reference speed is determined by the microcontroller executing the control algorithm. When the speed falls below this reference speed, the slip ratio becomes excessive and the solenoid valve is fired into the pressure-hold position by an output of

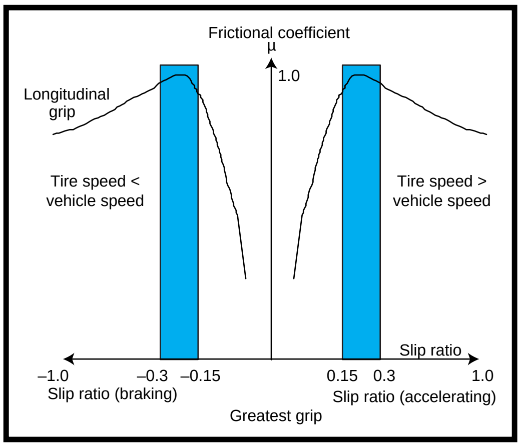


Figure 1—*This graph illustrates lateral and longitudinal grip as well as the relationship between slip and the frictional coefficient of the road surface.*

the control unit. This position puts a constant high pressure on the wheel so its speed continues to reduce.

At t = b, the wheel speed falls below the pressure-reduction reference speed (also derived by the microcontroller). The solenoid valve is then fired into its pressure-decrease position to reduce the pressure applied to the wheel.

To facilitate the pressure decrease, a hydraulic pump motor switches on to pump the hydraulic fluid from the wheel cylinders back to the master cylinder. This action causes the pulsing feedback on the pedal that the driver feels when ABS is engaged.

When the wheel speed increases again at t = c, pressure hold is reactivated until the wheel speed exceeds the control-reference speed, at which time the cycle starts again.

Incidentally, if the driver relaxes the pressure on the brake pedal, ABS is disengaged. The control system works only when the foot is pressed firmly on the brake pedal.

GET A GRIP

Traction-control systems (TCS) are like ABS, except that traction control optimizes wheel slippage when the vehicle is accelerating. This principle is illustrated in the right side of Figure 1, where the tire speed is greater than the vehicle speed.

TCS uses almost identical hardware to the ABS system with the addition of minimal hardware (usually two solenoid valves) and some software. Communication with the powertrain system is required to retard the engine torque while braking. This step is necessary, or the energy from the engine will dissipate in the transmission system.

RIDING STABLE

ABS improves stability by preventing the wheels from locking. If the front wheels lock, steering control is lost. If the rear wheels lock, the vehicle may yaw longitudinally and spin around. The next generation of ABS may further improve lateral stability using a system known as the electronic stability program (ESP).

ESP requires only slightly more complex hardware and software than a basic ABS. A yaw-rate sensor, lateral low-g accelerometer, and steering-angle sensor are added as inputs. The control algorithm compares the driver's actions (from the steering-angle sensor) with the motion of the vehicle (from the wheel-speed, yaw-rate, and low-g sensors).

If the vehicle becomes unstable, the brakes are automatically applied to create a yaw movment in opposition to the instability, canceling out the lateral forces that cause the instability.

Taking the concepts of ESP a bit further, a fully integrated chassis-control system would seamlessly handle the suspension, steering, and braking and would require real-time information on all six degrees of freedom of the vehicle as well as information on the status of each systems-control variable and a real-time communication link with relevant systems (e.g., powertrain).

Today, it's normal for the TCS to communicate with the powertrain system to adjust throttle angle while applying braking forces to achieve optimum traction. In the near future, one system will control the interoperability of these related subsystems.

The only real difference between the basic ABS, TCS, and ESP systems is the relatively small amount of incremental hardware and software, but a chipset approach works well. The software is written in a modular style (i.e., modules for wheel speed, traction control, etc.), and the electronic components' performance is determined with worst-case requirements in mind.

Figure 3 diagrams a chipset solution. There are four basic elements in every automotive electronic control unit—conditioning system inputs, conditioning system outputs, processing, and housekeeping functions (e.g., maintaining a stable power supply).

Three basic semiconductor technologies are applied—HCMOS for the processing portion, analog ASICs for the I/O conditioning and housekeeping, and Power-FETs for driving power stages (in this case, switching the hydraulic pump motor, which can be peak rated at over 100 A). With these basic technologies, infinite partitioning options are possible.

The input portion translates all the analog input signals to clean digital waveforms that can be applied

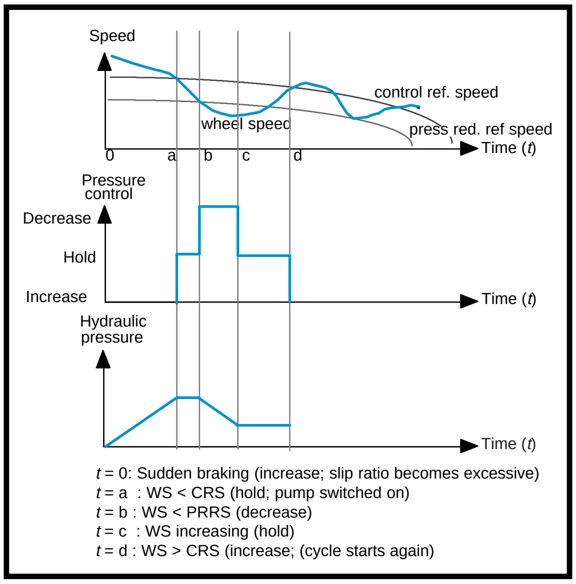


Figure 2—*These graphs demonstrate hydraulic pressure control using solenoid valves.*

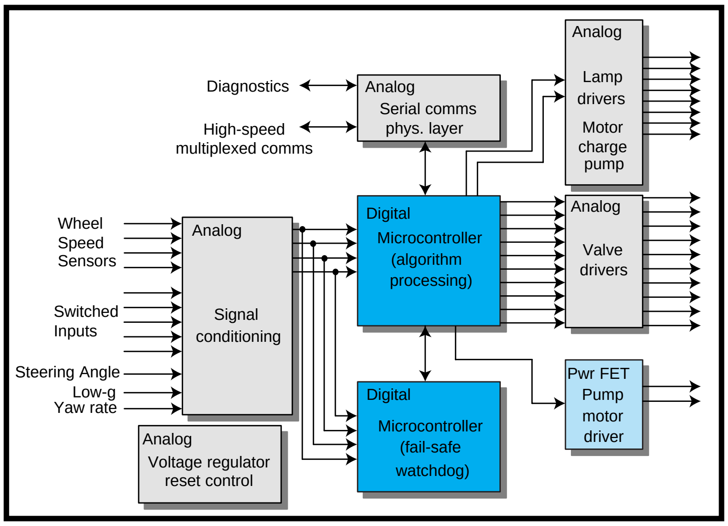

Diagnostics
High-speed multiplexed comms
Analog Serial comms phys. layer
Analog Lamp drivers Motor charge pump
Wheel Speed Sensors
Switched Inputs
Steering Angle
Low-g Yaw rate
Analog Signal conditioning
Digital Microcontroller (algorithm processing)
Analog Valve drivers
Digital Microcontroller (fail-safe watchdog)
Analog Voltage regulator reset control
Pwr FET Pump motor driver

Figure 3—*This block diagram is generic enough to cover the electronic controls required for two- or four-wheel ABS, ABS with traction control, and electronic stability programs.*

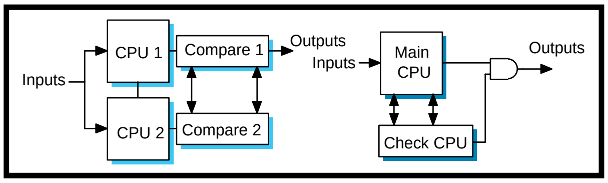


Figure 4—*In the variable reluctance wheel-speed sensor input circuit, the capacitors protect against electromagnetic interference and electrostatic discharge.*

directly to the microcontroller I/O. Figure 3 illustrates all of these sensor inputs grouped together in a single device. Although a single input conditioning device is possible, it is seldom implemented as such.

Because of the interfacing for steering-angle, low-g, and yaw-rate sensors, this device would only be required in the ESP system but would probably not be cost effective in a basic ABS. For this reason, at least two interface devices are usually specified. The second device is added to the basic chipset when the ESP system is developed.

Two processing elements are required in the processing portion of the circuit because a failsafe system is necessary. A failsafe system ensures that any faults in the electrical/electronic system are self-diagnosed and result in the system switching itself off safely, which would leave the conventional hydraulic brakes fully functional (pressure-increase mode) with the absence only of the ABS control.

Theoretically, a single microcontroller can observe and check each part of the system with the exception of itself. So, a second failsafe microcontroller is used to observe the operation of the master micro.

The output-conditioning portion of the electronic control system is, like the input portion, implemented in analog-based technology. SmarTMOS technology permits basic logic to be incorporated onto these devices to enhance performance. This smart functionality is used for diagnostics and to enhance failsafe operation.

For example, if a short or open circuit is detected at an output driver, the SmarTMOS device communicates that status to the microcontroller to ensure that the system remains in a safe operating mode.

FAILSAFE SYSTEM

The failsafe configuration is implemented using two microcontrollers (or two CPUs on a single microcontroller) and can operate in many different ways.

One approach—symmetrical redundancy, shown on the left-hand side of Figure 4—is to run the same software simultaneously on two identical CPUs. Symmetrical redundancy requires only one program to be written, but it's often inefficient in terms of silicon use.

The failsafe function can also be implemented by a less powerful CPU that performs only plausibility checks on key parts of the data. This equally popular approach is known as asymmetrical redundancy.

The chipset approach requires modular software that can be implemented selectively for different related systems such as ABS, TCS, and ESP. So, it's advantageous to use a pin-for-pin–compatible family of microcontrollers that differ only in memory size.

Because the ESP algorithm is more complex than the TCS algorithm, which is in turn more complex than the ABS algorithm, the CPU should be fast enough to execute the most complex algorithm within the minimum loop time of ~5 ms. Conversely, when the chipset is applied to only the basic ABS function, significant processing headroom is available. The wasted performance is certainly inefficient but is insignificant when microcontroller, hardware development, and software costs are considered.

SYSTEM PARTITIONING

The first step in electronic system partitioning is to evaluate system functionality and mapping the functional requirements to the basic technology that can implement these functions. Software must also be considered.

For example, it's possible to implement the wheel-speed timer system as a hardware unit that doesn't interrupt the CPU or as a basic unit that requires significant software control.

For the ABS/TCS/ESP chipset, technology mapping can be summarized as Sensors (wheel speed, steering angle, yaw rate, low-g accelerometer), Analog (signal conditioning, physical layers for mux comms, Vreg, solenoid drivers and charge pumps for high side drivers), Digital (MCU and failsafe), and Power (safety relay driver, pump motor driver).

At least in the near future, it's likely that sensors will continue to be implemented discretely with a conventional twisted-pair coupling directly to the ECU. But before too long, smart sensors that provide onboard diagnostics and communications will be deployed. These sensors will come with signal conditioning and will transmit information digitally on a multiplexed bus.

The high-rated power FET devices requiring current ratings that exceed the capability of a conventional analog ASIC may be implemented differently as well. One possibility is to remove the pump driver FET from the ECU and mount it mechatronically as part of the hydraulic pump motor assembly.

If the switching transistor and freewheeling diode were mounted inside the motor housing, a Faraday shield would be formed by the motor housing and would reduce radiated emissions. It may then be possible to switch the transistor faster and thus reduce losses as well as device costs.

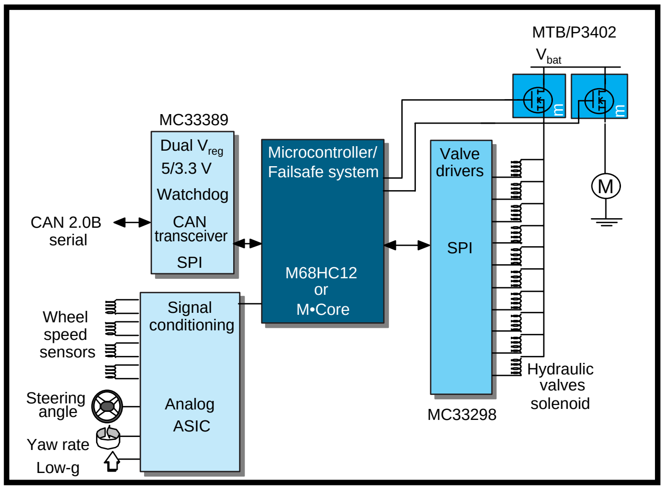


Figure 5—*If a master and a slave microcontroller are implemented and the slave only performs plausibility checks, the configuration is asymmetrically redundant.*

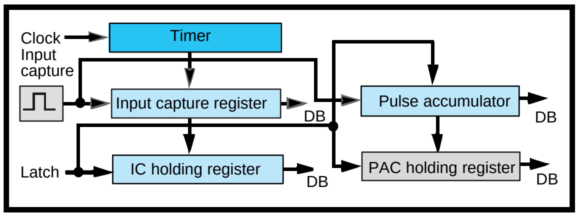


Figure 6—*The M•Core programmer's model is defined separately for two privilege modes—user and supervisor.*

The analog ASIC handles output signals and is expected to provide the drive signals for the dashboard lamps indicating ABS, TCS, and brake. In modern vehicles, these lamps are driven directly from an integrated dashboard control unit rather than the ABS unit. The ABS ECU talks directly to the dashboard ECU using the multiplexed serial communications link shown in Figure 3.

Figure 5 shows an implementation of a chipset that gives a tradeoff between standard components and custom devices. In reality, it's impossible to determine an optimal chipset until each of the system specifications for ABS, TCS, and ESP are understood, as well as the production forecasts for each system.

A physical interface for the controller area network (CAN) 2.0B serial communications system is included on the MC33389 device. If the preferred multiplexed communications was SAE J1850, this chip could be replaced by a pin-for-pin–compatible version with a different physical interface. These functions can also be implemented discretely by using stand-alone voltage regulators.

The MC33298 is used as the solenoid valve driver device, which, like the MC33389, includes a serial peripheral interface (SPI). There are eight power switches for the solenoid valves. In a TCS or ESP system, more solenoid valves are likely to be needed, so two MC33298 devices can be used or a custom driver can be implemented.

Reducing the chip count can reduce the physical size of the ECU, which lets you mount the control unit directly on the hydraulic valve block/pump motor assembly.

The best way to evaluate the CPU's processing requirements is via simulation. Once you know the processing requirements, you can select a microcontroller family.

Early systems were based on 8-bit CPUs, but as control algorithms become complex, 16- or 32-bit CPUs will be used.

To support the chipset's range of ABS, TCS, and ESP systems, three pin-for-pin–compatible versions of the microcontroller would be required, each with different memory sizes ranging from about 32 KB for basic ABS to 512 KB for an integrated vehicle-dynamics system.

THE M•CORE TEST

Implemented as a 32-bit RISC, the M•Core CPU is a good choice for the main algorithm controller in a braking and chassis-control chipset because it has a high throughput at a low clock speed and was developed specifically for automotive real-time control.

If only 16-bit performance is required, the 68HC12 architecture is suitable. The 68HC12 was developed for real-time embedded control applications and has custom features that were developed for ABS applications. One is the enhanced capture timer (ECT), which was implemented on the 'HC12BE32, 'HC12D60, and 'HC12DG128.

The ECT consists of a 16-bit software-programmable counter driven by a prescaler. The timer can be used for many purposes including input waveform measurements while simultaneously generating an output waveform.

There are eight input capture/output compare channels, four of which include buffers called holding registers. These buffers let two different timer values be memorized without the generation of an interrupt. Four pulse accumulators are associated with the four buffered channels to count pulses during a time specified by a 16-bit modulus counter.

In a braking application, the ECT offloads the timer interrupts associated with the wheel-speed sensors. The input frequency of the pulses is from 6 Hz per km/h to a maximum of 6 kHz for 250 km/h. This setup provides an interrupt, on average, around every 160 µs.

The ECT counts pulses from each wheel (using four timer channels) to generate the information required to calculate wheel speed every control cycle loop. After the information is saved in the input capture register, holding register, and pulse accumulator, only one interrupt is generated per cycle. The ECT timer is depicted in Figure 6.

Data is latched into the input capture holding register and into the pulse accumulator holding register. At the end of a control loop cycle, all relevant information required to calculate wheel speed during that cycle is available to be read directly from the CPU in these registers. The wheel speed can be calculated if the time of the first and last pulse is known along with the number of pulses acquired during the cycle.

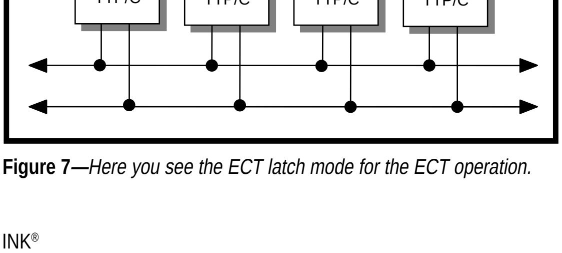


Figure 7—*Here you see the ECT latch mode for the ECT operation.*

BRAKING BY WIRE

No doubt, today's standard hydraulic-fluid braking system will someday be replaced by fully electrical systems. Brake-by-wire has several advantages—no brake fluid, reduced maintenance, lighter weight, increased performance (quicker response time), and minimized brake wear (spreads load across wheels more evenly).

There are also production advantages such as more simplistic/faster assembly and testing, more robust electrical interface, no mechanical linkage through the bulkhead, and fewer parts than a hydraulic-based system.

A few major issues must be addressed before brake-by-wire systems are adopted. For instance, in systems where the hydraulics are completely removed, there's no independent backup actuation system. So, fail-safely systems must be replaced with fault-tolerant (or fail-operational) systems.

The basic approach is usually redundancy. If nodes or ECUs fail, backups must come online without destroying the existing system integrity.

The degree of fault tolerance and where it's employed is likely to differ from application to application. But, it's reasonable to

Sensor	Analog	Digital	Power
Wheel speed	Signal conditioning	Microcontroller	Safety relay
Steering angle	Mux phy. layer I/F	Failsafe control	Pump motor driver
Yaw rate	Vreg and reset cct		
Low-g accel.	Solenoid drivers		
	Precharge cct		

Table 1—*Here's a basic list of the requirements and implementation needs for the typical chipset.*

expect that the important sensors and controllers (but not other components) will be replicated.

TIME-TRIGGERED PROTOCOL

Time-triggered protocol (TTP/C) is expected to become the standard technology in implementing fault-tolerant serial communications. The distributed embedded-control world already supports several serial communications systems (e.g., CAN, SAE J1850-DLC, and SAE J1850-HBCC specifications).

Three categories of communications systems are classified by the Society of Automotive Engineers. Class A is for low-speed networks used in vehicle body controls. Class B is for high-speed networks with no safety-critical requirements. Class C systems have stringent safety-critical requirements.

Because the existing protocols don't meet Class C requirements, the TTP/C was developed. The additional requirements are that message transfer must be deterministic with small and bounded latencies, all fault scenarios must be accounted for with a safe alternative operating mode, distributed clock synchronization (global time) must be supported, and the bus is guarded against "babbling idiot" nodes.

The existing protocols are unsuitable because they are event-triggered and the precise moment when a message will be received isn't specified. A communications protocol can only be predictable if worst-case transmission time and jitter are known during design and meet the application's requirements.

Real-time control applications are very sensitive to jitter. The time delay between presenting a message to be transmitted at the sender's interface and getting it at the receiver's interface is called the transmission time. Jitter is the variability of this transmission time (maximum transmission time minus minimum transmission time). The maximum jitter depends on the longest message that can be transmitted.

To ensure regularity of information transfer, the most suitable type of communications protocol is time-division multiple access (TDMA). Using a TDMA scheme ensures that nonpredictable message delays are not possible. Message transmissions are scheduled at the time of the design.

Each ECU is assigned a time slot in which it has exclusive access to the bus in order to send messages. Since every unit has its own time slot, collisions are impossible. Each transmission has the same priority for bus access so worst-case jitter can be calculated.

TDMA-based systems transmit state messages (e.g., whether a switch is on or off). State messages can be observed for a longer time than an event and are transmitted periodically. The state information is not consumed when it's read and no new value overwrites an old value until the next TDMA round.

In a typical distributed embedded control system, sensors are sampled or polled periodically during the control cycle. State messages are the most suitable message type for closed-loop control applications. But, events contain information that's valid at a certain point in time (until an overriding event).

Event messages are typically queued for consumption and consumed when read. They are more efficient in systems that have sporadic or rare occurrences that must be observed.

In time-triggered systems, all actions are derived from the progression of a globally synchronized time base accessible to all nodes. In event-triggered systems, all actions asre derived from the occurrence of events.

The resource requirements for a time-triggered system are determined before runtime so it behaves predictably and handles peak load situations deterministically. Event-triggered systems are less efficient than time-triggered systems when the system is operating at less than peak load because the system is set up to handle worst-case conditions that may rarely occur.

Figure 7 shows a TTP/C-based network. The four host controllers could be ECUs in a vehicle network (e.g., braking, steering, suspension, powertrain). Each node is composed of a host, controller network interface (CNI), and TTP/C controller. Two buses support redundancy, so if a fault develops on one, the alternative bus is available.

Fault tolerance is implemented by duplicating nodes. If one has a fault, a replica-determinant redundant node broadcasting the same result in another time slot replaces that node's signal.

The main strategy for fault tolerance in the TTP/C system is replicating fail-silent nodes. A fail-silent node must deliver a correct output or no output at all. When no output is generated, the hardware or software has a fault.

Several error-detection strategies are used to ensure fail-silence at the service level. The TTP/C controller uses watchdogs and a bus guardian, enabling the bus driver only during the node's transmission slot and disabling it at all other times.

COMING TO A STOP

Developing and selecting a chipset for an application is tough. And, the problem is compounded when the chipset must support several platforms.

The enabling technologies for conventional braking systems and brake-by-wire systems are similar. The major differences are the fault-tolerant and motor-control technologies. Just be sure to look at the total system.

Ross Bannatyne has worked with microcontroller products for the past seven years and currently works with automotive electronics in Motorola's transportation systems group. You may reach Ross at r11607@email.sps.mot.com.

SOURCE

M•Core, 68HC12
Motorola
(602) 952-4103
Fax: (602) 952-4067
www.mot-sps.com

FEATURE ARTICLE

Bob Perrin

Web-Based Circuit Engineering

You use the Internet, right? E-mail, online catalogs, search engines—they've become part of daily life. Soon, you may even get to simulate circuit designs online. Listen in as Bob sorts through the design trends available on the web.

The web is evolving so quickly, it's tough to keep track of what's hype and what's not. Anyone with a few bucks and an idea can put up a web page.

The web is one of the few refuges for low-overhead entrepreneurial startups. Ideas, concepts, and techniques are pretty fluid in this environment.

The interactivity of the web promises to change the way we work. Or does it? So far, the Internet has primarily been used the same way we use the telephone or mail.

If you want a book, CD, or video, you go to www.amazon.com, give them your credit card number, and a few days later, your new book shows up in the mail. If you want a datasheet for the AD620 instrumentation amplifier, you dial a URL, the web delivers the PDF, and you print the datasheet.

But, the web is more than a fancy telephone and express mail service. It has introduced some new tools as well. For example, search engines are front ends to massive databases, and they've changed the way many people do research. If you want to learn about mule-driving, for example, just run a search at www.yahoo.com.

The technology exists for the web to serve as a live front end to sophisticated CAD, CAM, and simulation tools as well. Many of the electronic design aids (EDAs) that engineers use every day could be used via the web.

WHY ONLINE EDA?

Traditionally, when an engineer needs an EDA, the company buys or leases it. Tool prices range from $35 to $350,000, and the rule is pretty simple: you get what you pay for.

But, the web may change all that. As pipe bandwidth increases, subscriber costs decrease, and tool vendors recognize there's money to be made, EDAs will become available online.

Imagine surfing to a site that offers a suite of PCB layout tools from Mentor Graphics, a SPICE simulation package, or an online version of MathCAD from MathSoft. The best tools in the industry would be available via the browser on your desk.

Will these tools be free? Probably not. Perhaps a user will rent time or a web site will sponsor the tool in order to attract people to the site.

The end effect would be an affordable way to use the best tools out there. Putting good tools in the hands of good engineers means shorter design cycles and faster time to market.

Small firms and consultants would reap the most benefits. For example, Z-World recently moved to a new set of schematic capture and PCB layout tools. The cash expenditure was significant because they purchased a large number of seats for the in-house engineering staff.

Z-World routinely uses a small pool of consultants, and these consultants had to purchase the new tool suite to continue to supply compatible CAD/CAM files. But, if the tools were on the web, both the company and the consultants could purchase time on the tools on an as-needed basis.

System administration issues associated with EDAs are significant, too. Installing the tools, getting them to work with existing hardware, and ensuring that they don't conflict with existing software are all overhead tasks that have nothing to do with product design.

In the web-based client/server model, the EDA vendor carries the system-administration burden. The user should not have to know anything beyond how to install a browser.

WHICH TOOLS?

EDA is a broad-brush acronym that encompasses everything from VHDL simulators to VLSI layout tools, from schematic-capture tools to word processors. So, what types of tools are most likely to be found on the web?

Circuit simulators (SPICE) and special-purpose calculators are already available. Currently, they are most useful as educational supplements, but over time, you can expect them to grow into usable engineering tools.

Special-purpose calculators are simulators dedicated to a single-circuit topology. For example, a special-purpose calculator may be set up to model an inverting amplifier designed around an op-amp. The user can adjust the parameters associated with various components (e.g., resistors, op-amp characteristics, power supplies, and input stimulus) and simulate the circuit.

These calculators can be helpful for learning about various circuit configurations. That's why they are cropping up as part of online courses and interactive app notes. But, they're typically too limited to be useful as a real tool.

To be useful, these calculators have to handle moderate scale end-use type circuits. For example, a calculator that models an inverting amplifier with sufficient feedback elements and parasitics would help you design a signal conditioner or transimpedance amplifier.

This type of application is easier to use than a full-blown SPICE simulator, and in that simplicity lies its beauty. Neurons, like iron, get rusty. When a tool like SPICE goes unused, it's hard to remember how to create simulations that provide reasonable results. Special-purpose calculators take a lot of the guesswork out of simulation.

Analog filter design is one area where a lot of engineers end up digging through dusty textbooks trying to remember how to put together a filter for their application. Many excellent texts show standard filter topologies normalized to 1 radian/s [1]. The engineer has to scale the component values to suit the current application, select standard values, and then simulate for component tolerances and temperature characteristics.

Special-purpose calculators could assist the engineer in all stages of design. The user would enter parameters describing the filter passband, ripple, stopband, and other characteristics. The calculator would then determine how many poles are needed, recommend a circuit topology, and compute target values for the components.

If the application was smart enough, it could juggle standard values into the equation and recommend the best compromise on standard parts and tolerances. As well, it could export a netlist for use with other EDAs.

Digital-filter design poses different obstacles. Quantization errors, numerical errors (rounding, truncation, overflow), sample rate, implementation (fixed-point, integer, floating-point), selection of coefficients and filter topology are all factors that must be considered when you design digital filters.

Many engineers find it daunting to understand, select, and simulate these parameters. Special-purpose calculators help remove some of the mysteries.

www.circuitsim.com offers several examples of special-purpose calculators as well as an analog-design wizard. Although the site and tools are still under development, there are some good examples of simple analog design and simulation aids.

See the sidebar "An Instrumentation Amplifier Tutorial" for a look at another type of online simulation tool—the interactive application note.

Filter design aids and similar tools (e.g., FilterCAD and SwitcherCAD from Linear Technology) exist today as autonomous applications. Their relatively limited scope makes them fairly easy to convert to web-based tools.

And, what HTML and CGI can't handle, Java can. Soon, instead of running some ancient filter-design calculator off of the 5¼″ disk packaged with an old college text, you'll find similar but enhanced tools on the web.

These applications will run over the web in a client/server model. You can probably expect them first from companies like Burr-Brown, Analog Devices, and Linear Technology.

Web-based tools will be easy to use, have software that's updated with current device models by the server's webmaster, and won't require installation of obscure and seldom-used software.

One nice feature of the client/server model is that you don't have to worry about downloading a virus-infected executable. Only the data of interest is delivered.

An Instrumentation Amplifier Tutorial

Application notes and tutorials from semiconductor manufacturers are plentiful, and many engineers rely on them to stay current with new technologies. Silicon vendors use app notes to showcase their latest products.

Consider *Errors and Error Budget Analysis of In-Amp Applications* [2], which can be found in PDF format at www.analog.com. It covers the basics of error sources in instrumentation amplifiers and contrasts Analog Device's AD623 with Burr-Brown's INA126. The tutorial contains a lot of information. Understanding the interaction of error sources in instrumentation amplifiers isn't trival.

Interactive web-based tutorials such as in Photo i also exist. In addition to theoretical models, the application contains models for two commercial devices—Analog Device's AD620 and Burr-Brown's INA118. This page is located at www.circuitcellar.com/simulation.htm

This application offers several advantages over text-only tutorials. As the reader navigates through the sections and examples, the simulator is automatically configured to illustrate the concept being discussed. The reader is free to use the HTML form to modify simulation parameters for additional self-guided experimentation.

(continued on page 33)

You'll also find simulators on the web. There are sites where a generic SPICE deck (SPICE netlist) can be submitted, the simulation run, and the graphic plots displayed. No special software is needed, just a browser.

Most of these sites are university pages and seem to come and go on a semesterly basis. Check out http://nn.uwyo.edu/sip for a good example of an online SPICE package.

In the future, we'll see schematic capture, PCB layout, and Gerber viewers online. Because these applications revolve around the creation of intellectual property (IP), security issues are sure to abound.

Users may be reluctant to do large designs over the web because the tool vendor will have access to the complete design. The possibility of leaking a customer's IP certainly exists, but this issue isn't new and is usually handled via a simple NDA.

REQUIREMENTS

In order for web-based tools to be implemented, it's going to take big fat network pipes, cheap bandwidth, continued innovation by tool companies, and, most of all, loud cries from engineers waving money at EDA vendors.

The fat network pipes and cheap bandwidth problems are interrelated. The network pipes have to be large enough to carry the vast amounts of information required to serve the graphic data over the web. And, bandwidth must be cheap enough for everyone to have on their desk.

Larger pipes are available in many cities. The monthly cost of a T1 is only a few hundred dollars. That puts reasonable bandwidth within the reach of most small- and medium-sized companies. And, you can expect the price/performance ratio to continue to drop.

Even if bandwidth is plentiful, online tools won't make it without continued innovation from EDA vendors. But, to predict the future behavior of vendors, you must first build a model. To do that, you have to look at history.

EDA companies seem to come and go. For the most part, they follow a me-too model.

Initially, a few simple schematic-capture and PCB layout tools cropped up. Then, almost overnight, there were dozens of CAD/CAM companies.

Most likely, this trend will continue. EDA companies already use the web as a distribution conduit for product and technical support. For example, the folks at www.geda.seul.org use the web to distribute GNU EDAs.

Some companies use the web to maintain collaborative libraries. Sooner or later, some company will offer a web-based EDA. And once again, the me-too folks will hop on the bandwagon.

The final ingredient is demand. Engineers must make the noise and wave the dollars to show they want web-based tools.

DESIGN TOOLS

EDA vendors are hard at work exploring various forms of a web-based client/server paradigm. Many are searching for a profitable business model. Others are boldly wading into the stream. Here's where a few of them are headed.

Intusoft manufacturers various simulation tools but is probably best known for their ICAP/4 SPICE simulator. Intusoft is already on the web with SpiceFarm.

SpiceFarm consists of 32 Pentium II 300-MHz computers (the workers) attached to a single "farm manager." The farm manager acts as a gateway to the web and distributes tasks to the workers. SpiceFarm is currently free to anyone using Intusoft's tools.

Failure-analysis simulations are SpiceFarm's specialty. This type of analysis is computationally intensive and involves many independent simulations. After these simulations run in parallel on the 32 workers, the farm manager collects the results and sends the data to the user via the web.

Intusoft hinted that they will have a "service side model" (client/server based) simulator online within a year. Issues related to security, scalability, maintenance, and billing still need to be worked out. But, no technological barriers are preventing these tools from becoming a reality.

After merging with Microsim, OrCAD added PSpice to its suite of schematic capture and PCB layout tools, which already contained Or-

(continued from page 29)

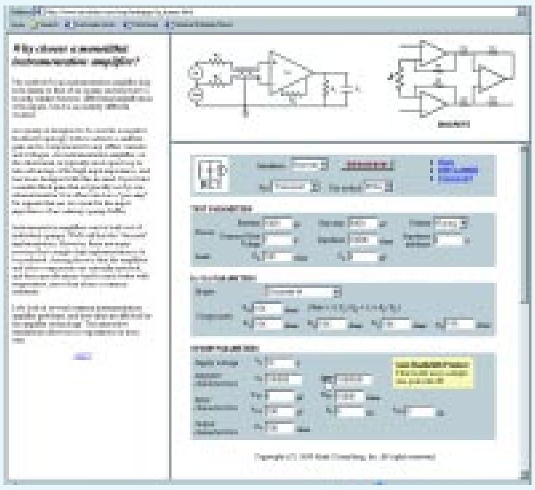


Photo i—*This is the first page of an interactive tutorial that compares monolithic instrumentation amplifiers with a three–op-amp discrete implementation.*

The user can also select the graphic format for results (see Photo ii). This feature is useful if you want to capture the results and use them in another document or if your browser displays some images better than others.

Traditional app notes for electronic components are nothing more than short tutorials. In the web environment, you can expect companies to imitate good ideas. An app note with a virtual workbench or breadboard (simulator) is an excellent idea.

One candidate for conversion to an interactive web-based application note is *Fast Settling Low Pass Filters* [3], which can be found at www.burr-brown.com/download/ABs/AB-022.pdf. It discusses how to use back-to-back diodes to reduce the setting time for filters subjected to large-amplitude step inputs. The techniques are useful in systems where a single antialias filter is multiplexed across several independent transducers.

But, learning the concepts and playing with the technique in op-amp circuits requires a bit of time to reason out. Reasoning is simply a manual simulation process. If the Burr-Brown note was converted to an interactive tutorial, the end user would be able to quickly simulate various circuit configurations. The engineer would spend less time doing thought experiments and there would be more efficient communication between Burr-Brown and the customer.

Another large on-line tutorial can be found at www.fp.physik.uni-konstanz.de/Applets/LockIn/LI1.shtml. This tutorial covers a somewhat obscure piece of instrumentation called a lock-in amplifier. These simulations require a Java-enabled browser like Internet Explorer 4.0 or Netscape Communicator 4.01. The tutorial isn't frames-based, so it's a bit harder to use than the application in Photo i. The site content, however, is excellent.

The Java front end for the lock-in-amplifier simulation is live. As you move sliders and click buttons, you immediately see the effects on the simulation waveforms, which constantly scroll across the simulation window. The look and feel is almost like sitting in front of a piece of test equipment.

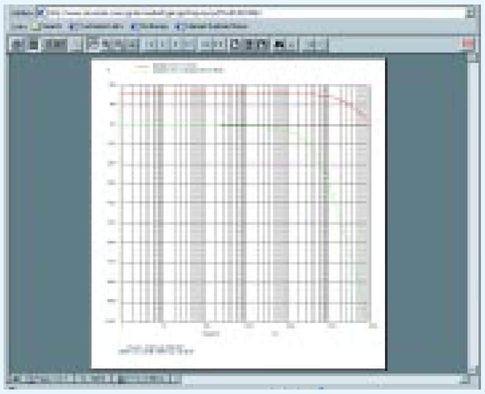


Photo ii—*When a simulation is run, the graphic results are displayed in a pop-up browser window.*

Java is the key to such interactivity. HTML is mostly limited to text-based forms of interactivity, but Java can provide a real-time front end to the application.

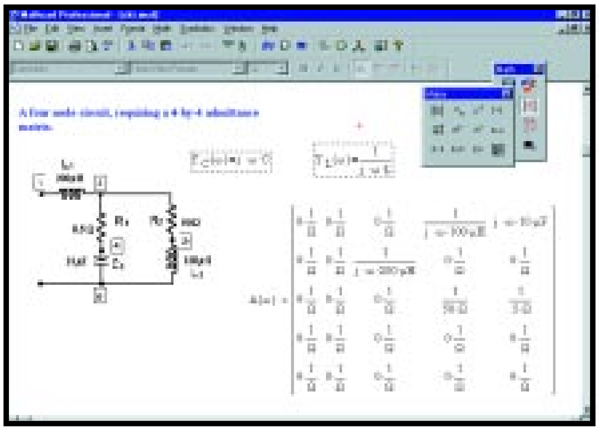


Photo 1—*MathCAD enables you to quickly create live simulations and solve problems in a human-readable freeform document.*

CAD Capture and OrCAD Layout. The new arrangement gave OrCAD a well-rounded set of EDA tools.

Although OrCAD doesn't have much in the way of web-based simulation, they do have the Component Information System (CIS), an online component library coupled with a web crawler. Users of OrCAD EDAs can use CIS to find and manage library components. The web crawler goes out to manufacturer sites and acquires component data.

As for future online EDAs, OrCAD is considering a thin client (client/server web-based) model. OrCAD also has a fresh approach to looking at business models for web-based tools. They don't consider traditional leasing and pay-per-use models to be issues and are more concerned with enhancing the tools.

Another popular Windows-based EDA vendor is Accel Technologies. Accel's EDA suite provides schematic capture and PCB layout tools that are easy to use and quite stable.

Accel has a project underway that will enable customers to download distribution files from an ftp site. At this time, they don't see the need for online web interfaces to their CAD tools. And, because they haven't received many requests for lease, loan, or rent plans, a pay-per-use model isn't a priority for Accel.

Accel has considered developing a web-based library assistant or library manager but is not currently working on the project. For now, they're content to wait and see where the rest of the industry leads.

Accel isn't the only company taking a somewhat passive approach. Mentor Graphics, the heavy hitter in the EDA arena, has no current plans to move its tools to a web-based client/server model.

Their tools already run in a client/server configuration over local high-speed Ethernet connections between workstations. Mentor believes that the web's current bandwidth limitations will make tools less than pleasant to use. Mentor still has a positive outlook on web-based tools, but it may be 5–7 years before they pursue such ideas.

COMPUTATION TOOLS

There are many other types of EDA tools in use. Several numerical analysis and symbolic evaluation tools can be found on the desks of engineers. MathCAD, MatLab, and Mathematica all fall under the broad brush of EDA tools.

MathSoft manufactures several numerical analysis and data manipulation packages. Their MathCAD V.8.0 is already web friendly.

MathCAD is a mathematical scratchpad. You enter equations, define functions, write programs, and build graphs

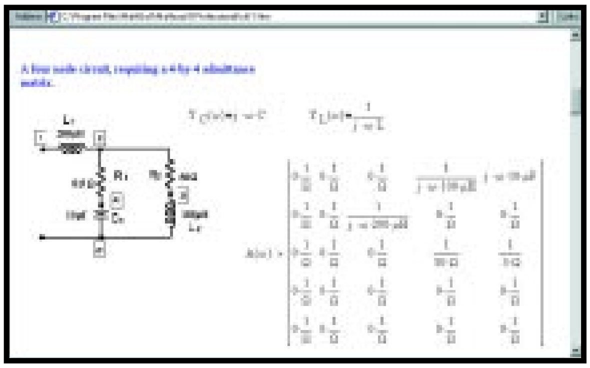


Photo 2—*Users of Netscape or Internet Explorer can view and use MathCAD worksheets over the web.*

in a graphical environment that's similar to using a pen and paper. The result is a document that looks like an engineer's notebook but is live and interactive. Photo 1 shows an example of a MathCAD document being created.

The computational engine links to the graphical page so the page evaluates and updates interactively. As you enter new data, the document updates, similar to how a spreadsheet recomputes. MathCAD supports both symbolic and numerical problem-solving techniques.

MathCAD 8.0 supports web interaction in two ways. It understands HTTP and can retrieve and use documents over the web. It can also save documents as HTML pages. For example, the 4 × 4 admittance matrix solver in Photo 1 was saved as an HTML document and opened over the web with a browser (see Photo 2).

Web friendliness was important in developing 8.0. The product's components (computational engine, symbolic engine, graphic interface and file manipulation unit) were cleanly separated to enable easy implementation of a web-based client/server architecture. MathSoft is trying to figure out a profitable business model before releasing a client/server version.

MathSoft has another application tool (StatServer) running under a client/server model over the web. This data-mining tool identifies and extracts patterns and trends from data.

FPGA TOOLS

Another EDA field concerns tools for developing applications using FPGAs. Xilinx, the foremost manufacturer of FPGAs, released its Silicon Xpresso initiative for Internet-based design.

Designers can use a number of tools, including evaluation fitters, resource estimators, power estimators, and an industry-wide (versus a Xilinx-only) search engine. Xilinx is watching to see how these tools are accepted and used. The feedback will determine the direction of the next phase of Silicon Xpresso's implementation.

The web-based fitter is currently limited to CPLDs and only returns the fitter reports, not a bitstream (loadable file). This tool helps evaluate how well a design will fit into a Xilinx CPLD.

Xilinx isn’t ready to have larger, more valuable IP uploaded to their servers. By restricting the system to a CPLD fitter evaluation, Xilinx hopes engineers will have a higher comfort level and use the tools more.

Security is a major concern. Silicon Xpresso web servers are in physically secure locations, and limited personnel can access the file system. Security checks are run on people with access to the equipment and files. Uploaded designs are deleted within 24 h, and no backups of user files are maintained.

WHERE IT STOPS...

There’s no question that the web has changed the way we work. But, predicting the future is a muddy affair. If online EDAs become a reality, it will be because you made the choice.

In a few years, we’ll see the impact the web has had on the workplace and we’ll know if all the uproar was hype or prophecy. Only time will tell.

Thanks to Boris Pevzner and Tina Guerette, MathSoft; Charles Hymowitz, Intusoft; Chris Rode, Rode Consulting; Jack Shandle, EDTN; Nancy Eastman, Accel Technologies; Phil Killcoin, OrCAD; Ted Iwanowski, Z-World; and Wallace Westfeldt, Internet Programmable Logic (Xilinx) for their assistance.

Bob Perrin works for Z-World. He has designed mixed-signal instrumentation for agronomy research, food science, harsh-environment industrial computing, and embedded control. You may reach Bob at bperrin@zworld.com.

REFERENCES

[1] A.B. Williams and F.J. Taylor, *Electronic Filter Design Handbook*, McGraw-Hill, New York, NY, 1995.

[2] E. Nash, *Errors and Error Budget Analysis in Instrumentation Amplifier Applications*, Analog Devices, App Note AN-539, 1998.

[3] Burr-Brown, *Fast Settling Low Pass Filter*, App note AB-022, 1994.

SOURCES

Accel EDA
Accel Technologies
(619) 554-1000
Fax: (619) 554-1019
www.acceltech.com

AD620, AD623
Analog Devices
(781) 329-4700
Fax: (781) 461-4261
www.analog.com

INA126, INA118
Burr-Brown Corp.
(520) 746-1111
Fax: (520) 741-3895
www.burr-brown.com

ICAP/4, SpiceFarm
Intusoft
(310) 833-0710
Fax: (310) 833-9658
www.intusoft.com

FilterCAD, SwitcherCAD
Linear Technology Corp.
(408) 432-1900
Fax: (408) 434-0507
www.linear-tech.com

MathCAD 8.0, StatServer
MathSoft, Inc.
(617) 577-1017
Fax: (617) 577-8829
www.mathsoft.com

MatLab
The Math Works, Inc.
(508) 647-7000
Fax: (508) 647-7001
www.mathworks.com

Mentor Graphics Corp.
(503) 685-7000
www.mentor.com

PSpice, Capture, Layout
OrCAD
(503) 671-9500
Fax: (503) 671-9501
www.orcad.com

Rode Consulting, Inc.
(781) 899-2290
www.circuitsim.com

Mathematica
Wolfram Research, Inc.
(217) 398-0700
Fax: (217) 398-0747
www.wolfram.com

Silicon Xpresso
Xilinx, Inc.
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com

NPC

PC/104 ENCLOSURE

Tri-M Engineering's **PC/104 Can-Tainer** protects PC/104 electronics such as instrumentation, data collectors, remote terminals, or SCADA packages that operate in hostile environments. The enclosures can accommodate any number of PC/104 modules and their cabling and peripherals in minimal space.

The Can-Tainer ensures that the PC/104 modules receive maximum protection from vibration and G-forces by using a dual system of isolating and absorbing rubber mountings. Internally, each corner of the PC/104 stack is held in place by a rubber corner system that isolates the cards from the extruded aluminum enclosure as it absorbs high-frequency vibration.

Externally, the anodized aluminum enclosure mates with a thick rubber mounting pad. This permits the Can-Tainer to be attached to a bulkhead while it absorbs low-frequency G-forces.

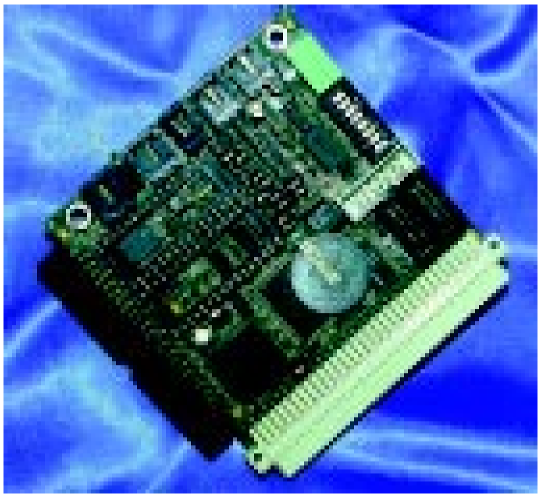
The Can-Tainer endcaps are available with or without I/O DE-9 and DB-25 holes. The Can-Tainer is NEMA rated when used with optional endcap gaskets and appropriate endcap and connectors.

The standard black anodized aluminum enclosure is available in three standard heights of 4″, 6″, or 8″. The Can-Tainer kit includes one solid endcap with no I/O openings, one I/O endcap with openings for four DE-9s and two DB-25 connections, 16 endcap screws, and one mounting kit.

The mounting kit has two gaskets, four rubber corner guides, eight rubber corner stops, CA glue, and an external thick rubber antishock mounting pad. A variety of other accessories including a mini-fan kit, cable kit, vertical divider, and heat tubing, are also offered.

The Can-Tainer sells for **$99**. Custom versions are available.

Tri-M Systems, Inc. • (604) 527-1100
Fax: (604) 527-1100 • www.tri-m.com



SINGLE-BOARD COMPUTER

The **MMT-188ES** is a low-cost SBC featuring the AM188ES 40KC/W microcontroller. This device provides a cost-effective solution for instrumentation and stand-alone embedded controllers.

The MMT-188ES features 512 KB of RAM and ROM, battery-backed memory, flash memory, and a real-time clock. Twenty-four programmable DIO lines and two serial ports (RS-232/-422/-485) provide the interfaces needed for a variety of OEM control applications.

Also included is a two-channel 8-bit DAC, a voltage-to-frequency converter, and eight channels of optoisolated I/O (three out, five in). Two DMA channels, four IRQ lines, one timer input, two status LEDs, and provisions for 12-V power are available as well. The MMT-188ES's PC/104 support enables the user to expand these interfaces even more.

The MMT-188ES developer's kit includes a programming cable, DOS in ROM, manual, and files.

Pricing is **$120** in 100-piece quantities.

Midwest Micro-Tek
(605) 697-8521
Fax: (605) 692-5112
www.midwestmicro-tek.com

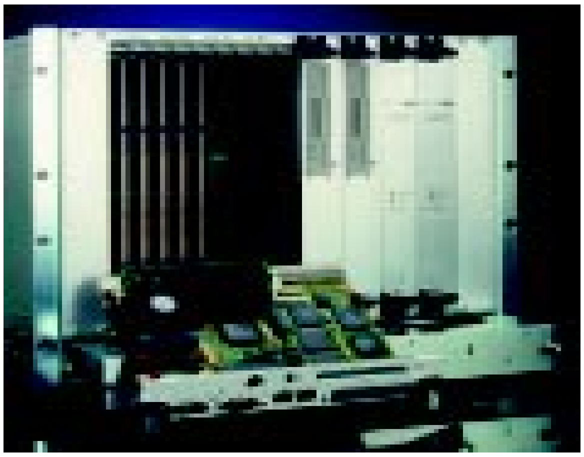
NouveauPC

edited by Harv Weiner

PENTIUM-BASED INDUSTRIAL SBC

TEK-CPCI-1004 is a CompactPCI SBC built around a Pentium processor and ALI Aladin V chipset. It supports Socket 7 and Super 7 Pentium processors at 133 and 166 MHz and Pentium processors with MMX technology at 200 and 233 MHz. The board supports a front-side bus of up to 100 MHz. and is targeted for datacomm, computer telephone integration, and industrial control/automation applications.

The board comes in 6U single slot (4HP) and 6U dual slot (8HP) form factors. The 6U-4HP board integrates an ultra fast/wide SCSI-3 interface, two 10/100Base-TX Ethernet interfaces, PCI-to-PCI bridge, and PMC mezzanine-support CompactFlash. There's a high-performance 64-bit AGP graphics interface with 2 MB of Rambus video memory as well as 64 MB of ECC SDRAM. The 6U-8HP supports all single-slot features, onboard hard disk and floppy drives, and up to 768 MB of ECC SDRAM or registered SDRAM on three industry-standard DIMMs.



The standard configuration has a built-in EIDE disk interface, which supports four hard disks, an onboard ultra-fast/wide SCSI-3 controller, two onboard PCI 10/100Base-TX Ethernet controllers, 512 KB of L2 cache, a PCI-to-PCI bridge, standard I/O devices, and two USB ports.

Pricing starts at **$1830** without SDRAM.

Teknor Industrial Computers, Inc.
(450) 437-5682
Fax: (450) 437-8053
www.teknor.com

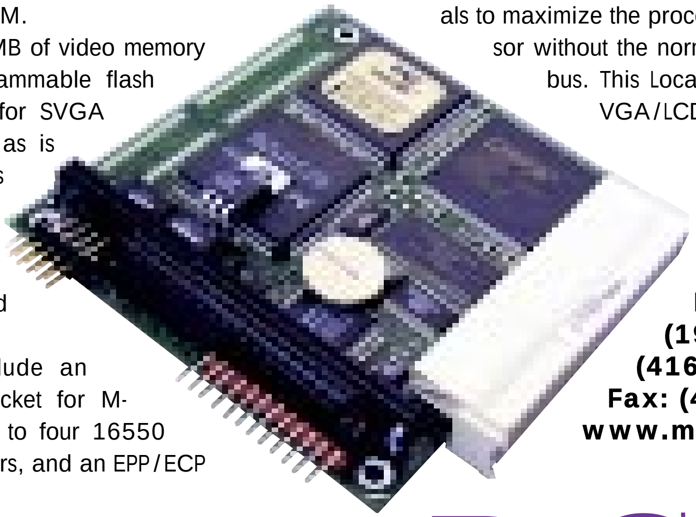
Nouveau PC

PC/104 SINGLE-BOARD COMPUTER

The **PC/II+DX** is a PC/104-compliant SBC that can be configured to optimize price and performance. The computer is based on an Intel or AMD '486DX 100-MHz CPU with up to 32 MB of EDO DRAM or 128 MB of SDRAM.

Provisions are made for 2 MB of video memory and up to 8 MB of user-programmable flash memory array storage. Outputs for SVGA and LCD displays are available as is support for two 3.5″ floppy disks and two IDE hard disk drives. A 10BaseT or AUI Ethernet LAN network connection as well as a SCSI-2 controller with optional offboard termination can also be added.

Other optional features include an SODIMM-144 socket, 32-pin socket for M-Systems DiskOnChip support, up to four 16550 serial ports with RS-232 transceivers, and an EPP/ECP parallel port. Also, there is a watchdog and dual-voltage power hardware monitor, real-time clock with battery, PC speaker output, AT style keyboard, and PS/2-style mouse ports.



The PC/II+DX uses a Local-bus architecture for its peripherals to maximize the processing power of its '486 processor without the normal speed limitations of the ISA bus. This Local-bus design is provided for the VGA/LCD controller to ensure maximum speed.

Pricing for the base model PC/II+DX starts at **$315** in quantity.

Megatel Computer (1986) Corp.
(416) 245-2953
Fax: (416) 245-6505
www.megatel.ca

Nouveau PC

Richard Russell

Embedded Framework and Foundations

Any structure needs a good foundation, and embedded applications are no different. Effective development of the structure requires a sturdy framework. Richard helps us out by putting one together for '186-based systems.

This article is about carpentry and foundations. No, not the kind you use to build a house—the kind you use to build small embedded systems. All systems need a framework, something on which to build the structure of an embedded application.

A good embedded framework includes basic functionality support such as a heap manager, a `printf()` engine, and debugging macros like `assert()`. A good framework also includes more than source code. It has `make` files, an easy-to-use directory structure, support for source-level debugging, and provisions for portability between compilers and other development tools.

Developing such a framework incurs some interesting problems. First, the environment in which an embedded program runs is different from that of a workstation. Workstation applications depend on the OS for basic functionality like I/O, memory allocation, and exception reporting.

In an embedded system, functionality is provided by the framework code. This statement holds especially true for smaller systems that can't afford a full-blown RTOS with an I/O subsystem.

Developing framework code often puts firmware engineers in a difficult position. Writing a new application, or porting an existing one, can't begin until the system framework is running. But, this low-level code can be difficult to write and must often be done with unfamiliar tools or hardware.

The third problem is tools. Workstation compilers are designed to develop software for workstations, not embedded systems. For example, most compiler libraries aren't suitable for embedded systems—especially standard C libraries that come with workstation compilers such as Microsoft, Borland, GNU, and the compilers distributed with most Unix systems.

These libraries are designed to interface to an OS and are written assuming workstation-level resources. Also, workstation libraries aren't designed to support storing data in ROM.

Take `printf()`, for example. The standard call is just an alias for `fprintf()` to the `stdout` stream. For small embedded systems, there is no `stdout` stream.

More complex embedded RTOS products do provide a file-system interface, much like a workstation OS. Even with no file-system–like I/O interface, these systems still let you configure the RTOS.

I've put together an application framework on an AMD Am186ES and Am186CC-based embedded systems that solves these (and other) problems. This framework pro-

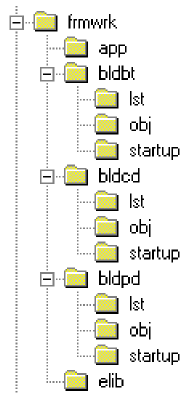


Figure 1—A good directory is important. This one is easy to understand and allows the entire application to be built of multiple tool chains.

vides the functionality needed for an embedded application, including a heap manager, a printf() engine, debugging macros, routines for handling interrupts, timer support, make files, and support for a source-level debugger.

GOALS

When developing the framework, I had several design goals in mind. First, it had to be simple, straightforward, and easy to understand. Too often, low-level code is a Byzantine mess of uncommented assembly and cryptic make files.

On many projects, team members are afraid to touch the low-level code because it has proven to be fragile. The make environment is often messy, difficult to understand, and prone to errors.

I recently worked on a project that only ran when it was built in debug mode and crashed when built with optimizations turned on. The debug build worked OK, so the binary file was shipped with the debug options. Worse, the make file dependency rules didn't take into account the project include files, so the system wouldn't recompile after header files had changed.

Second, the framework replaces some standard C library functionality that's unsuitable for use in an embedded system. Support for printf() and malloc() are prime examples.

Listing 1—Here's a basic printf implementation that uses a static buffer to hold the intermediate output. It's a typical implementation, but not as efficient as it could be.

```
int eprintf( const char* fmt, … ) {
  static char buf[ 128 ];
  int rv;
  va_list args;
  va_start( args, fmt );
  rv = vsprintf( buf, fmt, args );
  serial_send( strlen( buf ), buf );
  return rv;
}
```

Many printf() engines that come with compilers are big, rather slow, and coupled too closely with the I/O system. Mine is small, simple, and highly modular.

Third, I wanted the framework to be portable among different tool chains. Being portable between compilers is just the first step. The framework can link, locate, and be debugged with three different tool chains. Two 16-bit compilers (from Microsoft and CAD-UL) are supported.

My fourth goal was for the framework to be unencumbered by big-company licensing agreements. The framework is royalty-free and you can use it however you like. All I ask is that you keep the KUDOS section in each source file intact.

This framework certainly isn't the end-all be-all embedded library, but it does provide features I've found useful over the years (e.g., heap manager functionality not found in most workstation libraries).

Filename	Description
AM186LEDS.C	Supports the discrete LEDs on the AMD eval boards
AM186SER.C	Supports interrupt-driven communication on both serial ports on the Am186ES and the Am186CC
ASSERT.C	Provides a portable code checking macros and functions
BQUE.C	Contains a simple small FIFO queue handling routine used by the serial drivers
CRC32.C	Calculates 32-bit CRCs
ELIB.C	Provides some of the function calls normally found in the standard library
HEAP.C	Replaces the MALLOC functionality in the C libraries that come with the compiler
IN186.C	Contains system initialization code for the AMD boards
PRINTFE.C	Contains a small (about 1126 bytes), simple, and modular printf() engine. It's much easier to use than the one included with the compilers and is much smaller. It's also stand-alone and requires no other source files.
TMR186.C	Provides an interrupt-driven timer
TOASCII.C and ANYTOI.C	Provide functions for converting values to and from ASCII. These are more efficient to use than scanf().
LLIO.ASM	Provides support for I/O routines that are portable across compilers
LLINT.ASM	Supports using regular C functions as interrupt service routines without using compiler-specific keywords
LLPTR.ASM	Provides pointer manipulation routines that are used by HEAP.C to manipulate pointers without using the HUGE compiler keyword
LLAPI.ASM	Provides support for critical sections and test-and-set operation

Table 1—Here is a list of all the ELIB source files. The code is relatively modular and the lowest level code is in assembly language.

TOOLS

Management (and sometimes engineering) occasionally underestimates the value of good embedded development tools. I've seen countless projects struggle with poor or insufficient tools. Often, just getting an environment up and running takes longer than developing the application.

A complete embedded tool chain includes more than just a make utility, compiler, linker, and debugger. You need:

- an optimizing compiler with support for embedded targets
- an assembler
- a linker
- a locator (often part of the linker)
- a remote source-level debugger
- a remote monitor
- an in-circuit emulator (ICE)
- a good make utility and scripting language

Tool selection is often a matter of taste (i.e., the best tool is often the one you know). Other times, there are technical reasons for selecting a tool chain. Each of the tools I developed the framework with works well under Windows NT 4.0 and 95/98 and targets all AMD '186-based products.

CAD-UL provides a complete and fully integrated tool chain including a compiler, assembler, linker/locator, remote monitor, source-level debugger, and a graphical development environment. Their debugger also works with the QED emulator from Embedded System Products (ESP).

ESP provides the Beacon Suite, including the Metaware C compiler, a linker/locator, a remote monitor, and a nice debugger. Their tools also work with V.8.0c of the 16-bit Microsoft C/C++ compiler 16-bit Borland C/C++ compiler. ESP's hardware emulator (the QED) supports AMD '186 microprocessors and microcontrollers.

Paradigm Systems has long provided an embedded tool chain based on the

Borland 16-bit tools. Their latest version, based on the Borland V.5.0 development chain, supports their 16-bit locator and adds other debugging features. Their tools work with both the 16-bit Borland and Microsoft compilers.

The MKS toolkit from Mortice Kern Systems is a Unix-style Korn shell for Win32. It has a great make utility and is an excellent development environment. MKS also sells a version-control system called Source Integrity, which is fast and easy to use. It also handles complex projects well.

PC Lint V.7.50 from Gimpel Software is an indispensable tool that finds all kinds of hard-to-find nits in your source code that compilers miss. All of my framework code has been subject to Lint's scrutiny.

The Microsoft, CAD-UL, and Paradigm tools come with graphical development environments. V.1.0 of the MS Visual environment is just not suitable for embedded development. The CAD-UL tools come with an environment called the Workbench.

The Workbench is designed to drive a tool chain for embedded development. It has some nice features and enables you to easily plug in your own tools to the build environment.

I saw a comprehensive—and impressive—demo of the latest tools from Paradigm a couple of months ago. They extended the basic Borland environment with many features useful for embedded development.

In short, graphical development environments are fine for Windows development, but I don't use them for embedded development. I find command-line tools faster, easier to use, and much more flexible.

But that's just me. Try a graphical tool yourself. If you like it, use it.

LOCATING THE SOURCE

I tried to keep the framework directory structure simple. Although all of these tools support long file names, I didn't use any for source files. Most of the source code is in a directory called ELIB.

There are three build directories that contain make files specific for each tool chain—BLDBT for the Beacon Tools, BLDPD for the Paradigm Tools, and BLDCD for the CAD-UL tools. The source for the application (which you get to write) goes in the APP directory.

The make files for each environment are in the build directories with the start-up code for that environment. The build directories have some key files besides the make file.

The ENVIRON.KSH shell script sets up the environment for each tool chain. You'll need to modify it to suit your needs. If you don't use the MKS toolkit, you can easily change them to DOS-style batch files.

DEPS.MAK contains all the dependencies, which were auto-magically generated by a Perl script called MAKERULE.CML (available with the framework source code).

MR.KSH runs MAKERULE.CML to generate the dependency file. LIN.KSH runs Lint over the entire framework and dumps the output into LINT.TXT.

The ELIB directory contains the framework, and the APP directory is where the application source code goes (see Figure 1). In the example, only one file is named MAIN.C. Each of the build directories has a LST and an OBJ directory. That's where intermediate files are placed. Figure 2 shows the overall build flow.

All of the code can be compiled and linked with all three tool chains. The code is quite modular, and each source file supports a basic part of the functionality.

It's not rocket science, but you'd be surprised at the commercially available source-code products that pile all kinds of unrelated stuff into source and header files. The files in the ELIB directory contain the functionality found in Table 1.

Each file has a corresponding header file that declares the functions and data defined in that module. I like this approach better than the single include file that declares everything. Table 2 lists a couple of other header files that are also important.

Filename	Description
STDINC.H	Defines a set of portable data types used throughout the code. It's also designed to be portable across a wide variety of compilers and target processors. I've used this header file (or a simple derivative of it) on 16- and 32-bit 'x86, MIPS, 68k and PowerPC. The EDN Embedded Benchmark Consortium is also using this header for all its benchmarks that run on systems from 21 microprocessor manufacturers.
COMPILER.H	Used to define items that are compiler-specific. For example, the NEAR and FAR keywords are different between the CAD-UL and Microsoft compilers.
DEV186.H	Defines I/O locations and other information for the AMD 16-bit processors
ELIB.H	Declares functions normally found in the standard C library such as printf(), malloc() and exit(). Generally, with the standard C libraries that come with workstation compilers, it's not possible to use most of the functions from the following header files: assert.h, errno.h, signal.h, stdio.h, stdlib.h, time.h

Table 2—These are the most important header files. STDINC.H is particularly key and defines types in a portable, system-independent manner.

Listing 2—This implementation of printf() is much better. It doesn't use any intermediate buffer (either static or on the stack), which makes it inherently thread-friendly and removes output limitations.

```
static int consend(char c, void* ctx) {
   // Send a carriage return if you have a new line character
  if (c == '\n') {
   // Loop while the transmit queue is full
  while(Async186_putc(ctx, '\r') != Success)
  {;}
}
   // Now, send the character
  while(Async186_putc( ctx, c) != Success)
  {;}
  return 1;
}
int printf(const char* fmt, ...) {
  va_list args;
  va_start(args, fmt);
  return printfe(consend, Async186_ctx( CONSOLE_CHAN ), fmt, args);
}
```

Listing 3—Initializing the heap is easy. There are two required pieces of information: where the heap starts and how big it is.

```
extern Word heap_start_seg;    // Start segment of the heap
extern Word heap_end_seg;      // End segment of the heap
extern char FAR heap_start;    // First byte of the heap

heap_size = ((BlockSize)(heap_end_seg-heap_start_seg)) << 4;
heap = heap_initialize(&heap_start, heap_size);
```

To support items like `signal()` and `fopen()`, an embedded system needs significant support from a framework or embedded OS. Some RTOSs support an embedded file system that resides in flash memory. These products provide a significant amount of code to support the C library file I/O routines.

On the other hand, functions like `printf()` and `getchar()` are useful functions and don't require file handles. They are straightforward to implement over a serial port. `ELIB.H` defines these and other common functions, and `ELIB.C` maps them to their appropriate embedded implementations.

PRINTF

`printf()` is a good example of a useful function that doesn't require file handles. In an ANSI C standard library, `printf()` is part of the file I/O system and is a shell around the generic `fprintf()`.

The standard implementation of `printf()` formats the data and sends it to the `stdout` file handle provided by the C library, which sends it to the OS. This data can show up onscreen or end up in a file, depending on how the program's standard I/O pipes were initialized.

On many small embedded systems, there's no OS and the size and speed required to implement the C stream-based file I/O functions is burdensome. I've seen projects implement functionality similar to `printf()` using `sprintf()` or `vsprintf()`(see Listing 1).

There are a few problems with this function, and they all have to do with the character buffer `buf`, which holds the formatted results from `vsprintf()`. As coded, the buffer is static and is neither re-entrant nor thread friendly.

Because many systems don't use threads, that usually isn't a problem. The framework doesn't have any explicit support for threads, but most of it is re-entrant and thread friendly.

Using a buffer to hold the formatted output limits the size of the data. Dumping a lot of data using `printf()` overruns the internal buffer, corrupting memory and causing bugs.

One system I worked on had a 50-character internal buffer. It ran fine until the debugging `printf()` calls were enabled. Then the system crashed because most of them printed out more than 49 characters.

If you make the buffer big (1024 bytes or larger), then RAM is wasted. You may not think this situation is a problem, but try using an Am186ER that has 32 KB of RAM. When you use this processor with just a small flash memory, every byte counts.

If you need to make this function re-entrant for use from a thread or an interrupt service routine, the buffer needs to be made automatic (by removing the static declaration). Making the buffer automatic puts it on the caller's stack and makes the function re-entrant and thread friendly.

But, it eats up lots of stack space. Do you really want to add 128+ bytes to each thread's stack just so it can call `printf()`?

For this framework, I implemented a `printf()` engine that is completely standalone. It and all the support it needs are contained in `PRINTFE.C` and implemented using `printfe()`. The `printf()` engine supports the d, I, o, u, x, X, c, s, and p conversions but not floating-point data or the variable-width field specifier.

Most importantly, `printfe()` only buffers enough data to do individual conversions (i.e., all data is output as it is formatted). This technique supports arbitrarily long outputs with only one call to `printf()`, is fully re-entrant and thread friendly, and supports synchronization with output streams better.

All of this is done by passing `printfe()` a pointer to an output function (called the sender). As `printfe()` generates the formatted output, it calls the sender function to output the data. My `printf()` function is implemented in `ELIB.C` (see Listing 2).

`printfe()` is a pointer to `consend()`, which sends the formatted data out the console serial port. Note how `consend()` translates single `'\n'` (new line) characters to the `\n\r` (new line, carriage-return sequence). Other `printf`-style functions are implemented in the same manner.

MALLOC

Dynamic memory allocation is useful for many programming tasks, but it's seldom used for embedded systems. One reason is that, on small embedded systems, RAM is almost always a limited resource.

Traditional `malloc()` and `free()` routines are designed for workstations so they assume there's more than enough memory. And, it's often acceptable to report an out-of-memory error to the user and fail an operation or exit the program. But, this is unacceptable for embedded systems.

`ELIB.C` maps `malloc()` and `free()` to the `heap_alloc()` and `heap_free()` routines in `HEAP.C`, respectively. The heap manager is quite different from the `malloc()` support in the compiler's C library. As Listing 3 shows, it's straightforward to initialize.

The heap routines in the compiler's libraries are usually set up in the start-up code from assembly language. The framework's heap manager is portable, builds under many compilers, and supports many different targets.

Initializing the heap manager is simple. It just needs to know where the heap starts and how big it is. The start and end segments and the `heap_start` label are defined in `startup\heapdef.asm` under each build directory. These segments are put in the right place by the linker command scripts.

The heap manager has some important functionality not found in standard C libraries. This makes dynamic memory allocation in embedded systems more reliable.

`heap_reset()` permits a program to reinitialize the heap to its initial state. This function is easier to use than `heap_initialize()` and lets you keep all the initialization work in one place.

`heap_free_space()` returns the total number of free bytes in the heap. This function is helpful during debugging, and I often put a command in a small console interface that dumps this information.

Often it's more important to know the size of the largest free block than the total number of free bytes. `heap_biggest_free_block()` traverses the free blocks and determines the largest one.

`heap_check()` checks the entire heap and detects inconsistencies or corruption of the heap structure. `heap_stats()` handles statistics concerning the available free space, total size of the heap, number of free blocks, and number of allocated blocks.

There are two debugging compile-time flags for error checking and reporting. When enabled, the heap manager checks for writing past the end of blocks and the validity of pointers passed to its routines.

The heap manager also lets you support multiple heaps in a single application. That's useful when you must reserve memory for a specific function like communications buffers. You can also use the heap manager as a low-level fixed-size buffer manager (also called a pool).

The framework's heap manager isn't the most sophisticated, but it's small, simple, and easy to debug. Especially now that you have the source!

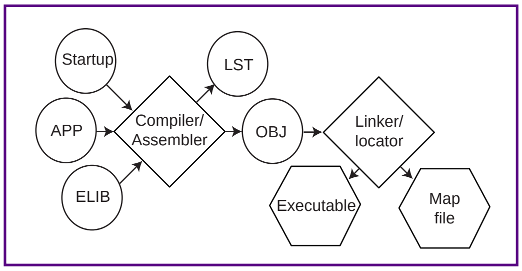


Figure 2—The build flow is pretty straightforward. The most important aspect is how the directories are separated. The library, application, and tool-dependent startup code are in separate directories, and I like to send the listing and object files to their own directories as well.

MOVING IN

Of course, no framework is the ultimate for every application, but this code shows how to write a simple, modular, and portable framework for a '186-based system. Enjoy! EPC

Richard Russell is the manager of systems software engineering for AMD's embedded processor division. He has been developing software for embedded systems for 15 years for devices such as wind-generator controls, point-of-sale systems, and fast Ethernet switches. He is also active in the EDN embedded benchmarking consortium. You may reach him at richardr@io.com.

SOFTWARE

Source code for this article is available via the Circuit Cellar web site. More information is in Design Forum.

SOURCES

Am186ES
Advanced Micro Devices
(800) 538-8450
(408) 732-2400
www.amd.com

Beacon Development Tools
(800) 769-943
(512) 338-9211
Fax: (512) 346-6382
www.beacontools.com

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

Gimpel Software
(610) 584-4261
Fax: (610) 584-4266
www.gimpel.com

Mortice Kern Systems, Inc.
(800) 265-2797
(519) 884-2251
Fax: (519) 884-8861
www.mks.com

Paradigm Systems
(800) 537-5043
(607) 748-5966
Fax: (607) 748-5968
www.devtools.com

Ingo Cyliax

Embedded RT-Linux

Part 5: Real-Time Programming

Information without an application doesn't take you far. So, Ingo finishes up this series by showing us how to put Linux to work. He describes how to install the RT-Linux extender and demonstrates how the different functions work.

It's hard to believe that Part 5 is here already. I was sitting in a coffeeshop with my laptop trying to figure out the best way to conclude this series on embedded RT-Linux. There's so much information left to cover…. Then it dawned on me—information is useless without application.

If you've been following this series, you know RT-Linux is a real-time extension to the standard Linux kernel. It permits priority-based preemptive multitasking under Linux and has facilities for dealing with interrupts. It also communicates with non-real-time processes (i.e., regular Linux processes) using pipe-like FIFOs.

Traditionally, you use a small real-time system to handle timing-critical control and connect it to a system that handles the higher level non-timing-critical management, perhaps over a serial port. That way, the real-time system can run independently and programs on the non-real-time system (perhaps for the user interface) are written in any way that's convenient.

With RT-Linux and FIFOs, you can combine both computers into a single system. The real-time system runs in the RT-Linux extender as a real-time task, and the higher level non-real-time component runs under the standard Linux environment without changes.

INSTALLING RT-LINUX

Because RT-Linux is an extension to Linux and isn't normally bundled with desktop distributions, you have to get the RT-Linux installation kit from www.rtlinux.org.

The kit comes in several versions, depending on which Linux kernel is on your system. Select the one that matches your kernel and download it.

The kit comes in a gzip'ed tar archive. To unpack it, use the `tar` command (e.g., `tar xzf rtlinux-0.5.tg`). This command creates a directory and unpacks everything you need to get started.

To build an RT-Linux kernel, you need the kernel sources, which are usually on the CD that comes with the Linux distribution. Once you install the kernel source tree and RT-Linux kit, you're ready to build the kernel.

The RT-Linux kit contains a patch file for the kernel. `patch` is a Unix/Linux utility that lets you update existing files based on a list of differences contained in the patch file. The file is a text file that you can look at.

The differences are generated with another program—`diff`. With an RT-Linux patch, you can install all the changes necessary to convert the generic Linux kernel into an RT-Linux kernel.

Of course, I didn't follow my own directions. I applied a patch for the 2.0.35 kernel to the 2.0.34 kernel that came with the RedHat 5.1 distribution. No big deal. `patch` was able to patch all of the files except one. By using the patch file as a guide, I could apply the difference to the remaining kernel module that it failed on.

The file had changed slightly from 2.0.34 to 2.0.35 to support a new feature. Because the patterns it was looking for weren't in the same place, the patch file got confused.

Luckily, the patch file format is readable and easy to understand (and I had kernel sources in the first place). That could have saved the day if I were in the field without a network.

In the latest patch kit (rtlinux-0.9h), you simply run a make and the kit will patch and install itself into the kernel sources. Once you apply the patch, you're ready to compile and compress the kernel and its modules. First, execute:

```
make zImage
make modules
```

To install the kernel, copy it to the `/boot` directory. I gave the RT version a different name than the standard kernel so I can use LILO to boot either one.

Listing 1 shows my `/etc/lilo.conf` file. To install the kernel and modules, use `cp arch/i386/boot/zImage /boot/vmlinuz-rt make install_modules`.

After installing the kernel, you need to run the LILO command to build the tables necessary to load the boot images at boot time. Now, the kernel will take effect when you reboot the system.

I'm assuming that you're doing all this on your development system. Being able to run the embedded application on your development system means you can develop and test it before embedding it into a flash disk. My laptop serves as my development system, portable information system, and terminal (on the rare occasions that it is connected to the network).

Once you boot the RT-Linux kernel, run some of the test programs that come with the RT-Linux kit (in the `testing` subdirectory). Each program has a `readme` file that tells you how to build and run each test program. There's also a `makefile` so you can build the module.

Most RT-Linux programs have two parts—a real-time module with the real-time–sensitive tasks and initialization routines, and a Linux user-level program that communicates with the real-time task through FIFOs. I'll run the `2Tasks` example and show you how it works.

First, I need to create the FIFO devices under the `/dev/` directory and load the real-time kernel modules that provide the scheduler and FIFO device driver. In rtlinux-0.9h, the `make` will create the FIFO devices automatically. Listing 2 shows what I need to do in order to build and run the `2Tasks` example.

Of course, you only need to install the FIFO device entries once. The kernel modules, however, need to be loaded each time you use the RT-Linux subsystem. Because they are modules, you can unload them when you're finished to preserve memory. Listing 3 shows how that's done.

PROGRAMMING

Now you've installed the system and verified that it works, so let's look at how RT-Linux is programmed. In Part 1, I showed you a table of the RT-Linux API (*INK* 100,

Listing 1—With this `/etc/lilo.conf` file configuration for my laptop, I can choose to boot either Windows 95 or one of several Linux boot images.

```
boot=/dev/hda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
image=/boot/vmlinuz
  label=linux
  root=/dev/hda5
  read-only
image=/boot/vmlinuz-rt
  label=rtlinux
  root=/dev/hda5
  read-only
image=/boot/vmlinuz-2.0.34-1
  label=linux-orig
  root=/dev/hda5
  read-only
other=/dev/hda1
  label=dos
  table=/dev/hda
```

Listing 2—The first steps to running the `2Tasks` example that's part of the RT-Linux installation kit are to create the FIFOs necessary for the real-time tasks to communicate and install the kernel-level drivers and the module containing the application (`rt_process`). Once the modules are installed, the user-level program (`app`) starts the tasks as well as reads and displays information from `app` onscreen.

```
[root@hugo 2tasks]# for i in 0 1 2 3; do mknod /dev/rtf$i c 63
  $i; done
[root@hugo 2tasks]# modprobe rt_prio_sched
[root@hugo 2tasks]# modprobe rt_fifo_new
[root@hugo 2tasks]# insmod rt_process.o
[root@hugo 2tasks]# lsmod
Module         Pages   Used by
rt_process         1   0
rt_fifo_new        2   [rt_process]   0
rt_prio_sched      1   [rt_process]   0
ds                 2   2
i82365             4   2
pcmcia_core        7   [ds i82365]    3
[root@hugo 2tasks]# ./app
FIFO 0: Frank Frank Frank Frank Frank Frank Frank Frank Frank
  Frank Frank Fra
FIFO 1: Zappa Zappa Zappa Zappa Zappa Zappa Zappa Zappa Zappa
  Zappa Zappa Zap
FIFO 0: nk Frank Frank Frank Frank Frank Frank Frank Frank Frank
  Frank Frank
FIFO 1: pa Zappa Zappa Zappa Zappa Zappa Zappa Zappa Zappa Zappa
  Zappa Zappa
FIFO 0: Frank Frank Frank Frank Frank Frank Frank Frank Frank
  Frank Frank Fra
FIFO 1: Zappa Zappa Zappa Zappa Zappa Zappa Zappa Zappa Zappa
  Zappa Zappa Zap
FIFO 0: nk Frank Frank Frank Frank Frank Frank Frank Frank Frank
  Frank Frank
...
```

p. 54). This API is implemented by the include files you link into your modules and the kernel modules that must be loaded before you run the application.

There are several include files for each subsystem in the RT-Linux kernel. rt_sched.h describes the main API to start and schedule real-time tasks. rtf.h describes the FIFO interface as seen by the real-time application, and rt_time.h is the interface to the timer subsystem.

The include files are stored in the Linux include directory (usually in /usr/include/linux). On my system they're installed in /usr/src/rtlinux/include/linux because I want to run several different versions of the kernel.

An RT-Linux application module has two entry points that are called by the module loader. The entry points—init_module() and cleanup_-module()—are called when the module is installed and removed from the system.

All of the initialization required to start the application (e.g., allocating tasks, installing interrupt handlers, creating FIFOs, and starting the tasks) is done in init_module(). When you remove the module from the system, cleanup_module() stops the tasks and frees up the resources (e.g., removing interrupt handlers and FIFOs) that were allocated in init_module().

To initialize a task, you need to allocate a task-data structure in global memory space by building a table of task slots. For example, RT_TASK tasks[2]; allocates two task slots.

If you want to use the tasks slots, initialize them with a call to rt_task_init(). This call takes a pointer to a task slot—a function that runs the stack size and priority of the task.

For example, rt_task_init(&tasks[0], fun, 0, 3000, 4); initializes a task with priority four and a stack size of 3000 bytes. The entry point is the function fun and it takes an argument (e.g., 0).

rt_task_delete() is the complement to rt_task_init(). It suspends the task and frees up the task slot for another task.

The task is initialized in a dormant state and needs an external event to wake it up. One way to wake up the task is to explicitly start a task off with rt_task_wake(), which makes the task runnable.

Typically, you want the task to wake up to a periodic timer. You can make a task periodic with a call to rt_task_make_periodic(). This function takes three arguments—the task slot pointer, a start time, and when the task needs to run.

When it's time to run, the task wakes up, goes to work, and puts itself back to sleep using rt_task_wait(). Other running tasks and handlers can put lower priority tasks to sleep via rt_task_suspend().

Another way to wake up tasks is via handlers. One of the two handler systems in RT-Linux is an interrupt handler, and you can register interrupt handlers for IRQ levels in the system.

When an IRQ occurs, all the interrupt handlers registered for that IRQ are run. A handler can then decide to wake up a task, which runs as soon as the interrupt handler is done and the interrupt is serviced.

To register an interrupt handler, use request_RTirq(). It takes two arguments—the interrupt request level and a pointer to a C function—to call when the interrupt is activated. To unregister an interrupt handler, call free_RTirq() for the interrupt level.

That brings me to the topic of I/O. All RT-Linux tasks run in kernel mode and thus have access to all I/O ports. In other words, writing real-time device drivers using RT-Linux modules is easy.

Listing 3—You can remove the modules needed to run real-time tasks under Linux when you finish. Removal is done with* rmmod*, which simply undoes the effects of* modprobe *and* insmod*.* lsmod *can be used to check the status of all installed modules.

```
[root@hugo 2tasks]# lsmod
Module          Pages           Used by
rt_process          1                 0
rt_fifo_new         2    [rt_process] 0
rt_prio_sched       1    [rt_process] 0
ds                  2                 2
i82365              4                 2
pcmcia_core         7    [ds i82365]  3
[root@hugo 2tasks]# rmmod rt_process
[root@hugo 2tasks]# rmmod rt_fifo_new
[root@hugo 2tasks]# rmmod rt_prio_sched
```

Listing 4—Here's the `rt_process.c` ***listing. This module contains the code to start and remove*** `2Tasks`***, as well as provide communication with the user-level program through the FIFOs.***

```
#define MODULE
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <linux/rt_sched.h>
#include <linux/rtf.h>
#include "control.h"

RT_TASK tasks[2];
static char *data[] = {"Frank ", "Zappa "};

void fun(int t)       /* t—the fifo number */
{
  while (1) {
    rtf_put(t, data[t], 6);
    rt_task_wait();
  }
}

int my_handler(unsigned int fifo)
{
  struct my_msg_struct msg;
  int err;
  RTIME now;
  while ((err = rtf_get(2, &msg, sizeof(msg))) ==
    sizeof(msg))
  {
    switch (msg.command) {
      case START_TASK:
      now = rt_get_time();
      rt_task_make_periodic(&tasks[msg.task],
        now, msg.period);
      break;
      case STOP_TASK:
      rt_task_suspend(&tasks[msg.task]);
      break;
      default:
      return -EINVAL;
    }
  }
  if (err != 0) {
    return -EINVAL;
  }
  return 0;
}

int init_module(void)
{
  rtf_create(0, 4000);
  rtf_create(1, 4000);
  rtf_create(2, 100);  /* input control channel */
  rt_task_init(&tasks[0], fun, 0, 3000, 4);
  rt_task_init(&tasks[1], fun, 1, 3000, 5);
  rtf_create_handler(2, &my_handler);
  return 0;
}

void cleanup_module(void)
{
  rtf_destroy(0);
  rtf_destroy(1);
  rtf_destroy(2);
  rt_task_delete(&tasks[0]);
  rt_task_delete(&tasks[1]);
}
```

Listing 5—The user-level control process (`app.c`) is the program you run to start real-time tasks and communicate with them. When the program is done, it shuts down the real-time tasks.

```
#include <stdio.h>
#include <errno.h>
#include <sys/time.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/rtf.h>
#include <asm/rt_time.h>
#include "control.h"
#define BUFSIZE 70
char buf[BUFSIZE];

int main()
{
  fd_set rfds;
  struct timeval tv;
  int retval;
  int fd0, fd1, ctl;
  int n;
  int i;
  struct my_msg_struct msg;
  if ((fd0 = open("/dev/rtf0", O_RDONLY)) < 0) {
    fprintf(stderr, "Error opening /dev/rtf0\n");
    exit(1);
  }
  if ((fd1 = open("/dev/rtf1", O_RDONLY)) < 0) {
    fprintf(stderr, "Error opening /dev/rtf1\n");
    exit(1);
  }
  if ((ctl = open("/dev/rtf2", O_WRONLY)) < 0) {
    fprintf(stderr, "Error opening /dev/rtf2\n");
    exit(1);
  }
  msg.command = START_TASK; /* now start the tasks */
  msg.task = 0;
  msg.period = (RT_TICKS_PER_SEC * 5000) / 1000000;
  if (write(ctl, &msg, sizeof(msg)) < 0) {
     fprintf(stderr, "Can't send command to RT-task\n");
     exit(1);
  }
  msg.task = 1;
  msg.period = (RT_TICKS_PER_SEC * 2000) / 1000000;
  if (write(ctl, &msg, sizeof(msg)) < 0) {
     fprintf(stderr, "Can't send command to RT-task\n");
     exit(1);
  }
  for (i = 0; i < 100; i++) {
    FD_ZERO(&rfds);
    FD_SET(fd0, &rfds);
    FD_SET(fd1, &rfds);
    tv.tv_sec = 1;
    tv.tv_usec = 0;
    retval = select(FD_SETSIZE, &rfds, NULL, NULL, &tv);
    if (retval > 0) {
      if (FD_ISSET(fd0, &rfds)) {
        n = read(fd0, buf, BUFSIZE - 1);
        buf[n] = 0;
        printf("FIFO 0: %s\n", buf);
      }
      if (FD_ISSET(fd1, &rfds)) {
        n = read(fd1, buf, BUFSIZE - 1);
        buf[n] = 0;
        printf("FIFO 1: %s\n", buf);
      }
    }
  }
  msg.command = STOP_TASK; /* stop the tasks */
  msg.task = 0;
  if (write(ctl, &msg, sizeof(msg)) < 0) {
    fprintf(stderr, "Can't send a command to RT-task\n");
```

(continued)

Listing 5—continued

```
    exit(1);
  }
  msg.task = 1;
  if (write(ctl, &msg, sizeof(msg)) < 0) {
    fprintf(stderr, "Can't send a command to RT-task\n");
    exit(1);
  }
  return 0;
}
```

Of course, you have to make sure you don't put something Linux needs in a strange state. The practical solution is to not share devices with Linux and RT-Linux applications unless you make special arrangements with the corresponding device driver under Linux and ensure that both devices know about each other.

I've covered field interrupts, how to start and manipulate tasks, and how to control hardware directly in an RT-Linux application. Now let's see how RT-Linux applications communicate with Linux processes. Remember, even though RT-Linux runs underneath Linux, the Linux processes aren't aware that it's running.

RT-Linux FIFOs are virtual device drivers that have ports in both the Linux and RT-Linux contexts. You can think of them as pipes that transport bytes from one domain to another. Whatever you write into one side comes out the other.

On the RT-Linux side, FIFOs are created with `rtf_create()`. This call identifies which FIFO to use and the buffer size. If an RT-Linux task wants to read and write the FIFO, it uses `rtf_put()` and `rtf_get()`.

Under Linux, a process opens a special file in the `/dev` directory corresponding to the FIFO. If you want to use FIFO 2, then the device is `/dev/rtf2`. Once opened, Linux processes read and write the FIFO via the `read()` and `write()` system calls.

By implementing a simple character device, the designers of RT-Linux enabled any process under Linux to communicate with a real-time task. It's even possible to communicate with real-time tasks using commands from the command line.

For example, you can use redirection to read and write a FIFO. `cat /dev/rtf1 & echo "s0" > /dev/rtf0` sends the string `"s0"` to a real-time task via pipe 0 and continuously reads the results from pipe 1.

The `2Tasks` example shows how it all fits together. Listing 4 contains the source code to the real-time module for `2Tasks`. Listing 5 is the user-level application code that controls the real-time task and reads the results.

In the real-time module, two tasks are initialized in `init_module()`, which is the module's entry point. `init_module()` creates three FIFOs that are used to communicate with the Linux application in Listing 5.

FIFO 2 is used as a control channel for the Linux application, and a handler is installed. A FIFO handler behaves like an interrupt handler. When there's activity on the FIFO, the registered handler (e.g., `my_handler()`) wakes up and goes to work.

`my_handler()` implements the task manager. The Linux-based application can send commands to this handler to start and stop each of the two tasks.

Remember, when tasks are initially created, they are suspended in a dormant state. The Linux application sends a start command over the control FIFO for each task. When `my_handler()` receives the commands from the FIFO, the tasks are scheduled periodically. Tasks can also be suspended so they stop running.

What do the tasks do? They send a six-character string to one of the two remaining FIFOs (one sends `Frank` and the other sends `Zappa`). After adding the string to the FIFO, they suspend themselves using `rt_task_wait()` and wait for the next periodic timer wakeup.

When the module is removed, `cleanup_module()` deletes all the FIFOs, makes the tasks inactive, and removes them from the scheduler. Because deleting the FIFO automatically removes the handler, you don't have to unregister the handler for the FIFO. Any attempt to use the FIFO from the RT-Linux or the Linux side results in an error.

Now, let's look at the Linux application side. Listing 5 starts by opening the Linux side of the FIFOs created on the RT-Linux side (i.e., /dev/rtf0, /dev/rtf1, and /dev/rtf2).

After opening the necessary FIFOs, the program creates a message to start each task. It chooses a period of

$$\frac{RT\_TICKS\_PER\_SEC \times 5000}{1000000}$$

or 5 ms. In my system, the hardware timer resolution is 0.8 ms, and RT_TICKS_PER_SEC equals 1,1931,80, which is the frequency that the timer chip sees.

Once the two tasks are started, the program sits in a loop and reads data from the FIFOs. It uses the select() system call, which is a Unix/Linux facility that handles event processing.

With a bit mask, you select which open file descriptors (like file handles under Windows/DOS) to block on. If there's I/O activity on the specified file descriptors or a specified timeout occurs, the select() call unblocks. This method is an efficient way to deal with many I/O channels and makes Unix and Linux popular in communication applications.

After reading from the FIFOs, the data is sent to the standard output channel, which is the display. When the final count of 100 is reached, the process sends messages to the FIFO handler to suspend the tasks. Having done the work, it closes the FIFOs and exits.

THAT'S ALL, FOR NOW

As you see, RT-Linux is a great tool. www.rtlinux.org has links to interesting projects like a haptic interface, data acquisition systems, and a control software prototype for a CNC machine (see Photo 1).

Look for more RT-Linux applications in upcoming articles. Because it runs on my laptop, I prototype ideas there first. Once I iron out the initial concept and take care of some bugs, I may port it to another system. Or maybe not. RPC.EPC

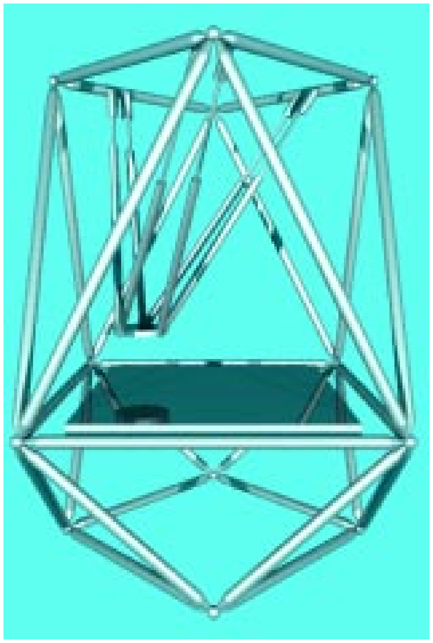


Photo 1—The kinematics necessary to control this Stewart platform–based CNC machine machine were prototyped in RT-Linux.

Ingo Cyliax has written for Circuit Cellar *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

ICE on Tap

Part 1: SuperTAP Setup

Faster than a speeding bit, able to emulate '486 processors in a single bound—it's SuperTAP. This virtually crash-proof system uses two processors for debugging, and Fred's got all the hardware details right here.

Recently I started a new hardware design project. I began by searching the Circuit Cellar Florida room for all the necessary goodies. You know the routine—wire cutters, voltmeters, o-scopes, solderless breadboards, things like that.

The most interesting tool I had to revive and verify was an in-circuit emulator (ICE). I've reached the point where I won't sit down and design anything complex unless I can twiddle the bits electronically. No more burn and churn for me. I depend on my emulator.

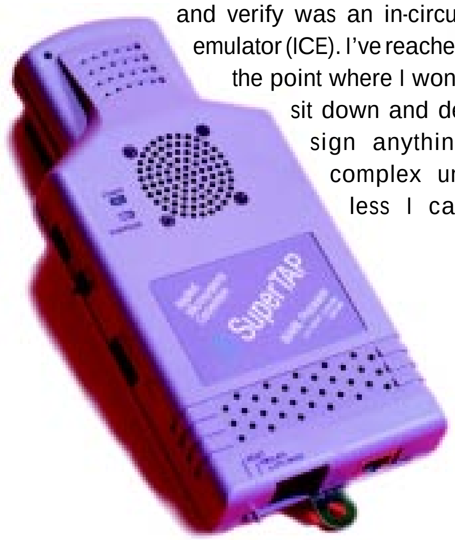


Photo 1—Everybody needs a toy like this one.

In fact, I thought my PICmaster ICE was broken, so I replaced every IC on the interface card only to find out later that an Ethernet card and Windows NT in the host PC were the bad guys masking the existence of my ICE card.

Whether it's PIC or '*x*86, the ICE gives the designer an incomparable view of the firmware and hardware. Such a beneficial tool deserves a closer look. Thanks to the folks at Applied Microsystems, you and I will be privy to the hardware and software that make up the '486-based ICE called SuperTAP. There's 3.46^2 of SuperTAP documentation, so let's get going.

INVENTORY

The SuperTAP emulation system came in a rather large box. Let's open it up and take a look inside.

The first thing we'd look for is the SuperTAP hardware itself. Look no further than Photo 1. The SuperTAP is palm-sized and packs all the hardware necessary to emulate a '486 processor. The whole idea here is to debug stuff.

The SuperTAP replaces the target system's embedded processor. Communication with a host PC is done through RS-232C or Ethernet. I like this already. Those of you who follow my column know how much I like Ethernet interfaces. The Ethernet lashup provides data-transfer speeds that approach 7 MB/min.

The ultimate destination for any information emanating from SuperTAP is the CAD-UL XDB high-level language debugger loaded on the host PC. Using the SuperTAP with XDB, you can interactively control and examine the state of the target system.

For the advanced user, SuperTAP comes standard with features like trace disassembly, overlay memory, and a four-level event system. There's also the capability of operating with low-voltage targets. The voltage function is switch-selected on the SuperTAP in Photo 1.

If you've ever built an ICE device, do you remember struggling with the hardware design to leave as many open I/O lines as possible after you implemented your emulator hardware? Remember writing the code as efficiently as possible in an attempt to leave as many resources as possible for your target? Remember writing the monitor code?

Success depended on which platform you targeted and how rich the I/O set on that target was. Well, if you ended up working for Applied Microsystems and you worked on the SuperTAP project, you learned to use no target resources.

None. No memory, no I/O ports, no serial ports. SuperTAP is transparent to the target system right down to the software level. SuperTAP doesn't interfere with target interrupts and doesn't insert wait states during emulation.

How can this be? The answer lies in the processors—SuperTAP uses not one but two. The emulation processor replaces the target processor. The second processor (the control processor) is in charge of communicating with the debugger and is also responsible for monitoring the emulation processor activity.

The result: a virtually crash-proof system. By isolating the target-system processor from the debugger, two things automatically happen. First, the target can hang and the debugger side keeps on truckin'. Second, real-time operation is guaranteed.

Another side effect was also incorporated as a feature. The feature is nonstop emulation (NSE), and it enables the target to run unabated even while you define events and upload traces.

```
This is a sample HOSTS file used by Microsoft TCP/IP for Chicago.

This file contains the mappings of IP addresses to host names. Each entry should be kept on
an individual line. The IP address should be placed in the first column followed by the
corresponding host name. The IP address and the host name should be separated by at
least one space.

# Additionally, comments (such as these) may be inserted on individual
# lines or following the machine name denoted by a '#' symbol.
#
# For example:
#
#    102.54.94.97    rhino.acme.com       # source server
#    38.25.63.10     x.acme.com           # x client host

#    127.0.0.1       localhost            this is the original entry

     126.1.1.1       taphost              # Host PC
     126.1.1.2       tapcard              # SuperTAP ethernet card
```

Figure 1—This is the raw hosts file that came with my "Chicago."

Applied Microsystems claims that 95% of software bugs are the result of data corruption in the stack and at the data pointer level, which causes program leaps to nonexistent code. These leaps typically take the emulator with them…right into la-la land.

Fortunately, SuperTAP's dual-processor configuration provides crash recovery, which shields the control/debugger portion of the emulation process from the target.

In a nutshell, the SuperTAP boasts:

- real-time emulation up to 66 MHz
- crash recovery
- 3- or 5-V operation
- hardware and software breakpoints
- overlay memory
- a GUI-based state-machine event system
- high-speed trace with timestamp
- trigger I/O capability

It also detects CPU clock frequency, target V_{CC}, and hung-bus conditions.

You've seen the star player, so let's look at the rest of the team members. The box also contained an ISA Ethernet card and all the necessary cables to attach the SuperTAP to a target and host. I found it interesting that a couple of Ethernet adapters for AUI and BNC were included. These folks have it covered.

Box number two (inside the main box) contained an Ethernet adapter (see Photo 2) designed to attach the SuperTAP to a network and a 3Com ISA Ethernet adapter for a PC. Again, the Applied Microsystems engineers covered all the bases, right down to terminators for the BNC Ethernet link. The RS-232C interface gear was stowed here, too.

Box three included more power cables and a big ol' brick power supply. Hmmm, what's that in Photo 3? A full-blown '486 (without processor, of course) made by Adastra Systems.

I searched the world over for documentation, but found none in the boxes. Good thing I do embedded. It looks like a standard configuration.

Thanks to the Internet, documentation (or the lack thereof) should pose no problem. If I need to know more about the Adastra board, I'll get on the 'Net and put out the word.

***Photo 2**—This module sets the SuperTAP apart, in my eyes.*

SUPERTAP 101

I know most of you are experienced engineers and thinkers. I use the word "most" because some readers may be reading their first embedded feature. For those of you who've been embedded for some time, bear with me. I think it's appropriate to take time to define some of the features for the newer readers.

BREAKING IT DOWN

Let's start with breakpoints. What they are and how they work is generally understood. The object here is to define how they work with the SuperTAP. I'll begin with software breakpoints.

SuperTAP provides you with 256 software breakpoints. They replace instructions in the target with a special opcode that forces a specific predefined task.

A typical software breakpoint vectors the main program to a diagnostic routine that reports user-defined status to the designer. SuperTAP responds to a software breakpoint by halting execution and replacing the breakpoint opcode with the original instruction. Software breakpoints in SuperTAP can only be implemented in RAM because you can't write to ROM.

If you can't write to ROM and you need to break inside ROM space, SuperTAP gives you eight hardware execution breakpoints that work in the ROM domain. Four of these are in user-mode address space, and the remaining four are in system management mode (SMM) space.

Hardware breakpoints work like software breakpoints, with one exception.

Hardware breakpoints work in ROM and RAM. They occur only when the instruction is actually executed to prevent breaking on a prefetched break opcode.

TRIGGER HAPPY

Remember, at first, I was seeking out test equipment. Why? Because sometimes you have to "see" a signal. Embedded computers run very fast and the only way to capture or see a particular event is to trigger on it.

SuperTAP provides one input pin and one output pin to support external triggering. This trigger I/O enables logic analyzers and storage scopes to capture the event for later viewing and debugging.

Now, let's define the event system and its relation to the triggering process. The SuperTAP NSE lets you modify events without stopping emulation so you can effectively debug in real time.

Event triggering is based on event comparators. There are four levels of events and eight comparators that are defined by address, data and status conditions. These comparators enable you to set event breakpoints on program events, target hardware events, and CPU bus state. Several events can be set up:

- address match
- address range
- data match
- data range
- external trigger input
- trace full
- counter value
- event system level

You can include bus cycle status as part of your comparator condition for event-system hardware breakpoints. The conditions include:

- memory read and write
- instruction fetch
- I/O read and write
- system-management mode
- branches

CPU signals like NMI or INTR also qualify as an event. Once the event conditions are specified and met, SuperTAP responds by:

- breaking emulation
- switching between event-system levels
- counting occurrences with 32-bit counters
- tracing a single cycle
- turning trace on or off
- generating a trigger output

SuperTAP's four-level architecture and "trace one cycle" action lets you define a sequence of events for capturing a trace of a particular cycle or set of cycles during program execution. The timestamp function provides accurate timing information ranging from 30 ns to 8 h.

TRACING HISTORY

If you single-step through code and write down all the information you think you need, you don't need trace history. If your code is lengthy, you may be debugging that project for a few months.

But, if you use SuperTAP's trace-history feature, you can capture and record real-time execution history of your target system's processor. SuperTAP can store up to 64k bus cycles, all of which are timestamped. You can even view and upload the trace data without stopping the emulation.

Trace history enables you to view raw bus cycles, assembly and/or C-level source, and data cycles. The CAD-UL XDB debugger permits scrolling and searching to speed the trace-reading process.

MORE THAN MEMORY

Technically, overlay memory is just RAM. But not just any RAM. Overlay memory is SuperTAP emulator RAM that can wear many different hats.

Overlay RAM can be target read/write, target read-only, reserved or guarded, 8-, 16-, or 32-bit. One advantage of using overlay RAM as guarded memory is that if your application decides to write to the guarded area, a breakpoint is generated and an access violation is reported.

If this happens, your program has tried to get at memory outside its limits. The same is true for read-only overlay memory. You can think of it as automatic debugging.

NSE GOTCHA

One last comment before we start passing electrons through SuperTAP. NSE is neat, but you never get something for nothing. You can dynamically view registers and variables during program execution with NSE, but there's a price.

Every time the screen updates, it momentarily stops emulation to read the emulator memory. If your application chokes on this type of activity, you can disable the dynamic update operation in the XDB debugger application.

HARDWARE SETUP

Basically, there are two ways to implement the SuperTAP—RS-232C and Ethernet. Let's try both, starting with RS-232C.

SuperTAP requires a host PC with a '486 or greater processor and at least one 16550-equipped serial port. If you're serial-port challenged, an empty ISA slot for that Ethernet card included in the SuperTAP evaluation kit will do. Fortunately, all of those requirements are met with the equipment at hand.

As you might imagine, connecting the SuperTAP to a serial port is child's play. A modular RJ-45 cable connects the SuperTAP to a 9- or 25-pin adapter that plugs into the PC's serial port.

While you were reading, I took the liberty to install the CAD-UL XDB debugger software. SuperTAP automatically adjusts for the transfer rate selected by the debugger. That's pretty straightforward. Let's move on to the Ethernet setup.

There's a couple ways to get SuperTAP attached to Ethernet. The first one assumes that you don't want to share it on a LAN. Let's do a little SuperTAP LAN 101.

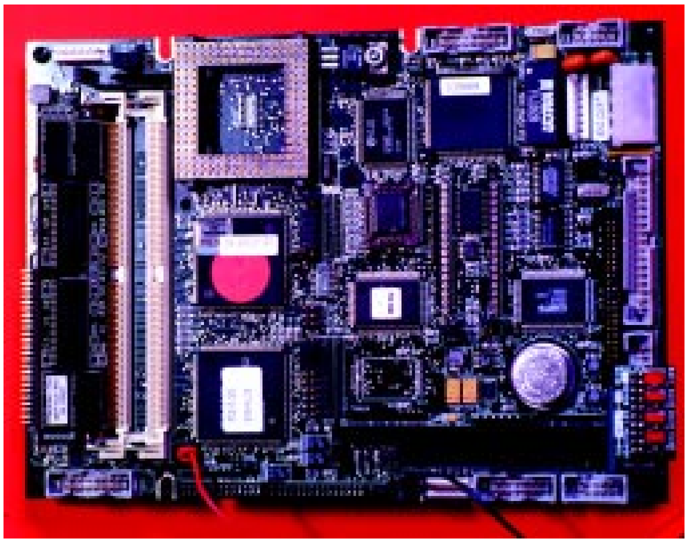


Photo 3—Looks like a '486, feels like an embedded '486, smells like....

SuperTAP uses TCP/IP to communicate with the host PC in a stand-alone Ethernet network configuration. As you know, each device on a TCP/IP network has its own unique address.

To set up our stand-alone TCP/IP network, make sure Bill's TCP/IP software is installed and the Ethernet adapter works. The Applied Microsystems folks supplied the necessary parts in the evaluation kit.

Next, install the 3Com Ethernet card. Once that's done, it's simple Windows 95/NT stuff to load a TCP/IP stack. I prefer NT, but due to the limitations of HiJaak, I'm forced to use Windows 95 here.

With the physical card and TCP/IP stack loaded, we can identify the PC host. Let's use 126.1.1.1 with a netmask of 255.255.0.0 to identify our PC host.

The next step is to create a hosts database to identify the PC host and the Ethernet adapter and associate the IP addresses and host names of both devices. When you do this, be careful not to overwrite IP information that already exists. If your file has good stuff in it, save it somewhere and create a new hosts file.

Either way, edit the hosts file to reflect the Internet address and host name of the host PC and the same for the Ethernet adapter. Our hosts file is shown in Figure 1.

That special Ethernet adapter (see Photo 2) is the Applied Microsystems Ethernet communications adapter. It connects to the network via a 15-pin AUI interface.

A modular cable connects this adapter to the SuperTAP. Remember the interface adapters? Now we know what they're for.

Now, you connect a terminal (either a dumb 3101 type or a smart Windows Hyper type) to the Ethernet adapter and load the flash memory with the Ethernet adapter's IP address and netmask. Of course, these entries must match what you put into the hosts file.

ON TARGET

So far, I've described the SuperTAP evaluation system and its physical components. Although it runs in a stand-alone mode, the next step is to connect it to the embedded PC that came with the package.

But, connecting SuperTAP is only the beginning. We still haven't explored using the it over a multimachined LAN. And, right now, all you know is the name of the debugger software. I want to show you how it plays with the SuperTAP hardware.

And let's not forget the Adastra '486. Looks like next month's article is set. Check in to see if SuperTAP can help take the complicated out of 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

'486
Adastra Systems Corp.
(510) 732-6900
Fax: (510) 732-7655
www.adastra.com

SuperTAP
Applied Microsystems Corp.
(800) 426-3925
(206) 882-2000
Fax: (206) 883-3049
www.amc.com

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

ISA Ethernet adapter
3Com
(800) 638-3266
(408) 326-5000
Fax: (408) 326-5001
www.3com.com

DEPARTMENTS

62 MicroSeries

70 From the Bench

76 Silicon Update

MICRO SERIES

Joe DiBartolomeo

TPU
Programming in Microcode

Part 3 of 4

If you're going to program the TPU, you'd better know how to use its native microcode! But, the real trick to writing microcode is understanding the hardware. Fortunately, Joe has all the explanations right here.

In Parts 1 and 2, I looked at some common implementations of the timer/counter function found on microprocessors. I also introduced the time processor unit (TPU) and explained how it increases system throughput by freeing up the CPU from having to handle timing/counting functions.

Essentially, the TPU is a coprocessor that runs its own microcode and operates semiautonomously from the CPU. To program the TPU, you must use its native microcode.

In Part 1, I provided a table of the canned TPU functions for mask sets A and G of the Motorola '68332 (*INK* 102, p. 70). You may have noticed that these TPU functions are well suited for automotive and motor/motion-control applications.

For example, in the automotive field, the TPU's ability to autonomously provide pulses of controllable times and widths is used in ignition systems. As well, its ability to read encoders and measure pulse widths or frequency is useful in flywheel/gear-tooth applications.

The TPU also has extensive motor/motion control functions such as reading Hall-effect sensors associated with brushless motor applications and controlling stepper motors. Given the number of microprocessors used in

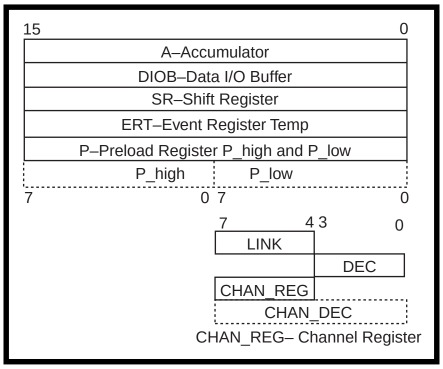


Figure 1—*Here's the register set of the TPU's execution unit. All TPU channels share these registers. The active channel is determined by CHAN_REG.*

the automotive and motor/motion-control areas, tailoring the canned functions to them is a good marketing decision.

There are also several other canned functions that aren't related to either of these areas. For example, one canned function produces a universal asynchronous receiver transmitter (UART).

The canned functions not only increase system throughput by off-loading some of the CPU's work, they also reduce programming time. Given the extensive list of canned functions, learning to program the TPU may seem like an esoteric exercise done merely for the edification of the programmer.

However, no matter how extensive the canned functions are, there can never be enough mask sets to match the requirements of all users and applications. More importantly, having the TPU resources under direct programmer control provides another level of flexibility.

Keep in mind that TPU use isn't limited to just timing/counting functions. Once you can program the TPU, only your imagination limits what the TPU can be used for.

You can think of the TPU as a small microcontroller. I've found that the best way to understand a new micro is to study the assembly-language instruction set and the hardware diagrams.

In a complex microprocessor, of course, that can be a time-consuming exercise. However, with the TPU, it's fairly quick because the TPU command set consists of only 11 subinstructions. So let's review the instruction set and structure of the TPU's microcode, as well as the TPU hardware structure.

When you write in microcode, understanding the hardware is much more important than if you were writing in C. It also helps to keep the hardware registers in mind, visualize the flow of data, and understand the TPU time slices. All of these actions are necessary because programming in microcode is far less accommodating than programming in assembler or C.

The TPU lacks many standard hardware features that are taken for granted in C or assembler. For example, when you're programming the TPU, remember to latch the condition codes if you want to use them in the next instruction. The hardware doesn't automatically latch them.

EXECUTION UNIT

The execution unit—the heart of the TPU—executes the microcode instructions. Figure 1 shows the execution unit's register set. These registers are connected to one another by two buses, as you can see Figure 2. Notice that some registers can access both buses whereas some have access only to bus A.

The execution unit is a shared resource among all the TPU channels, but each TPU channel also has three individual 16-bit registers—the match event register (MER), capture register, and greater than–or–equal to register. These registers communicate with the execution unit via the ERT register.

The accumulator (A), data I/O buffer (DIOB), and preload (P) registers are all general-purpose registers. Each channel's registers can place their contents on bus A or B but can only accept data from bus A.

The DIOB and the P registers can read and write data to TPU parameter RAM. These registers are the only ones in the execution unit that can access TPU RAM. Parameter RAM is the only means of data transfer between the TPU and CPU. The P register has the added feature of 8- or 16-bit operation.

The AU unit performs arithmetic operations, as well as shifts using the AU shifter. When combined with the shift register, a 32-bit shift is possible.

Two latches, one for bus A and one for bus B, are associated with the AU unit. The AU unit also has carry-in and B-invert bits. Note that data can be written into the AU unit from either bus A or B but that the AU unit can only write to bus A.

There are also four status bits—carry (C), overflow (V), zero (Z), and negative (N)—associated with the AU unit. They are similar to status bits found on standard microprocessors.

The link register links channels together and enables a channel to respond based on events occurring on other channels.

The decrementer register (DEC) is used in the `REPEAT` and `DEC_RETURN` subinstructions, and it is also used as a counter value.

The channel register (CHAN_REG) holds the active channel. It is required

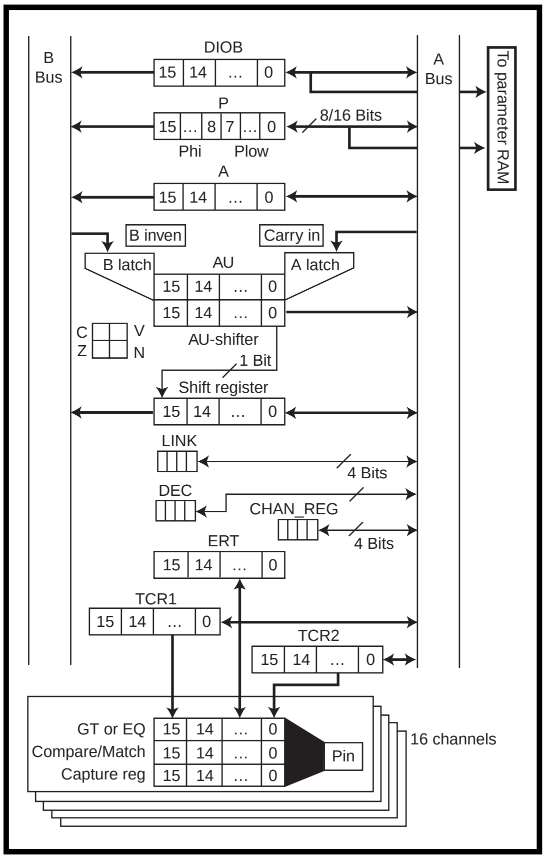


Figure 2—*The execution unit is shared by all TPU channels. Notice that some registers have access to both A and B buses, and other registers have only limited access. Note also that registers DIOB and P are the only ones with access to TPU parameter RAM, and are used to pass parameters to the CPU.*

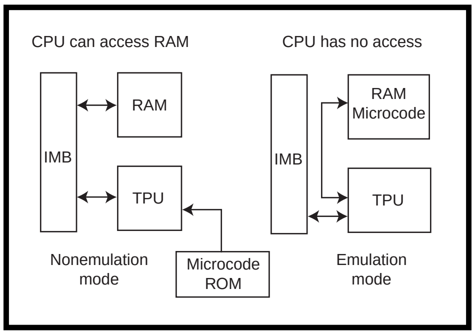


Figure 3—*In nonemulation mode, the TPU runs the canned functions stored in the processor's ROM and the CPU has access to onboard RAM. In emulation mode, the TPU runs microcode that has been downloaded into onboard RAM, prior to entering emulation mode. In emulation mode, the CPU has no access to onboard RAM.*

because the execution unit is a shared resource.

The event-register temporary (ERT) register captures the contents of one of the active channels' registers based on the channel's setup. Note that the ERT can only access the active channel or A bus. It has no access to the B bus.

As I discussed previously (*INK* 102), the '68332's two timers (TCR1 and TCR2) are free running. Both TCR1 and TCR2 have access to the TPU channels and A bus but not to the B bus.

I made a point of indicating which registers have access to which bus to emphasize that, when you're writing microcode, you must keep accessibility in mind. Hardware accessibility defines which commands are valid and which are not. Now, let's look at the microcode instruction set.

TPU MICROCODE

To aid in the development of microcode, a TPU assembler is available. I use the TPUASM assembler which executes on IBM-compatible machines and assembles the microcode from a user source file.

Motorola maintains a master library of functions (including the canned functions) which can be downloaded and used by any programmer. This library enables you to combine some of the canned functions with custom functions. You can even order your own custom mask set containing any combination of functions, provided you meet the minimum order quantity.

In normal mode, the TPU runs microcode that is stored in processor ROM (i.e., mask set A or G). For the TPU to run user code, it must be placed in emulation mode. You do this by setting the EMU bit in the TMCR register (covered in the previous articles in this series).

Before setting the EMU bit, you have to load the microcode into onboard RAM or flash memory (depending on the micro). For example, the TPUASM produces an S19 file that must be downloaded into onboard RAM. For the '68332, it is loaded into 2 KB of onboard RAM. Once the EMU bit is set, the CPU cannot access the onboard RAM (see Figure 3).

MICROCODE STRUCTURE

TPU microcode is made up of micro-instructions that have the following structure:

```
{label} subinstruction 1 {;
  subinstruction 2 ;
  subinstruction 3 …}
```

where items enclosed in `{}` are optional. Each micro-instruction is 32 bits long and must have one of the five possible formats shown in Figure 4. Subinstructions have the structure `keyword field 1 { , field 2, field 3);`.

Each keyword specifies a TPU resource. Within the same micro-instruction, no two subinstructions can have the same keyword. `field` provides operands for the subinstructions.

The syntax for a micro-instruction is quite simple. Micro-instructions end with a period. Subinstructions (also called subcommands) are separated by semicolons, and fields are separated by commas.

In the following sample microcode, `ram`, `chan`, `goto`, and `au` are keywords. Note that the carriage return is not a delimiter.

```
ram field;
chan field, field;
au field, field.
chan field, field;
goto label.
```

A micro-instruction executes in two CPU clocks. As Figure 5 shows, this arrangement gives four time slices per instruction. It's important to note when certain actions take place relative to the four clock slices, as I'll show you shortly.

SUBINSTRUCTIONS

Let's look at the 11 subinstructions in more detail.

`au` performs arithmetic and shifting operations. The operands are sourced from the A and B buses and the results placed on the A bus. The syntax of this subinstruction is:

```
au adest op (constant / ex-
  pression) {,ccl} {,shift}
  {,read_mer}
```

where *adest* is the A bus destination. Valid word destinations are `a`, `diob`, `p`, `sr`, `tcr1`, `tcr2`, `ert`, and `nil`. Byte destinations are `p_high`, `p_low`, and `chan_dec`. The valid nibble destinations are `link`, `chan_reg`, and `dec`.

`op` is the operator. It can perform its operation on the valid constants or expressions listed in Table 1. Valid B sources are all word length and can only be `a`, `p`, `sr`, or `diob`.

The optional `ccl` instruction latches the condition codes at the end of the instruction. If `ccl` isn't included in the subinstruction, the condition codes aren't changed. That's a detail that isn't an issue in assembly or C programming.

The optional `shift` instruction shifts the contents of the shift register to the right one-bit position.

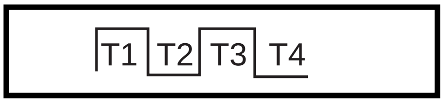


Figure 5—*A TPU instruction is executed in two CPU clocks. There are four time slices (T1–T4) associated with each TPU instruction.*

The optional `read_mer` instruction reads the match event register (MER) into the ERT register.

You can see that the `au` subinstruction can have up to two sources—one from bus A and one from bus B. At the start of the subinstruction, the AU latches are loaded and added together. Subtraction is accomplished by inverting the Bsource before addition.

The results pass through the AU shifter and are placed back on the A bus and routed to the destination. Note in Figure 5 that addition occurs in T2 and a fetch of data from parameter RAM occurs in T4. So, if a value in parameter RAM is part of the `au` instruction, an `au` instruction to load `p` or `diob` with the desired parameter RAM value is required before the addition operation.

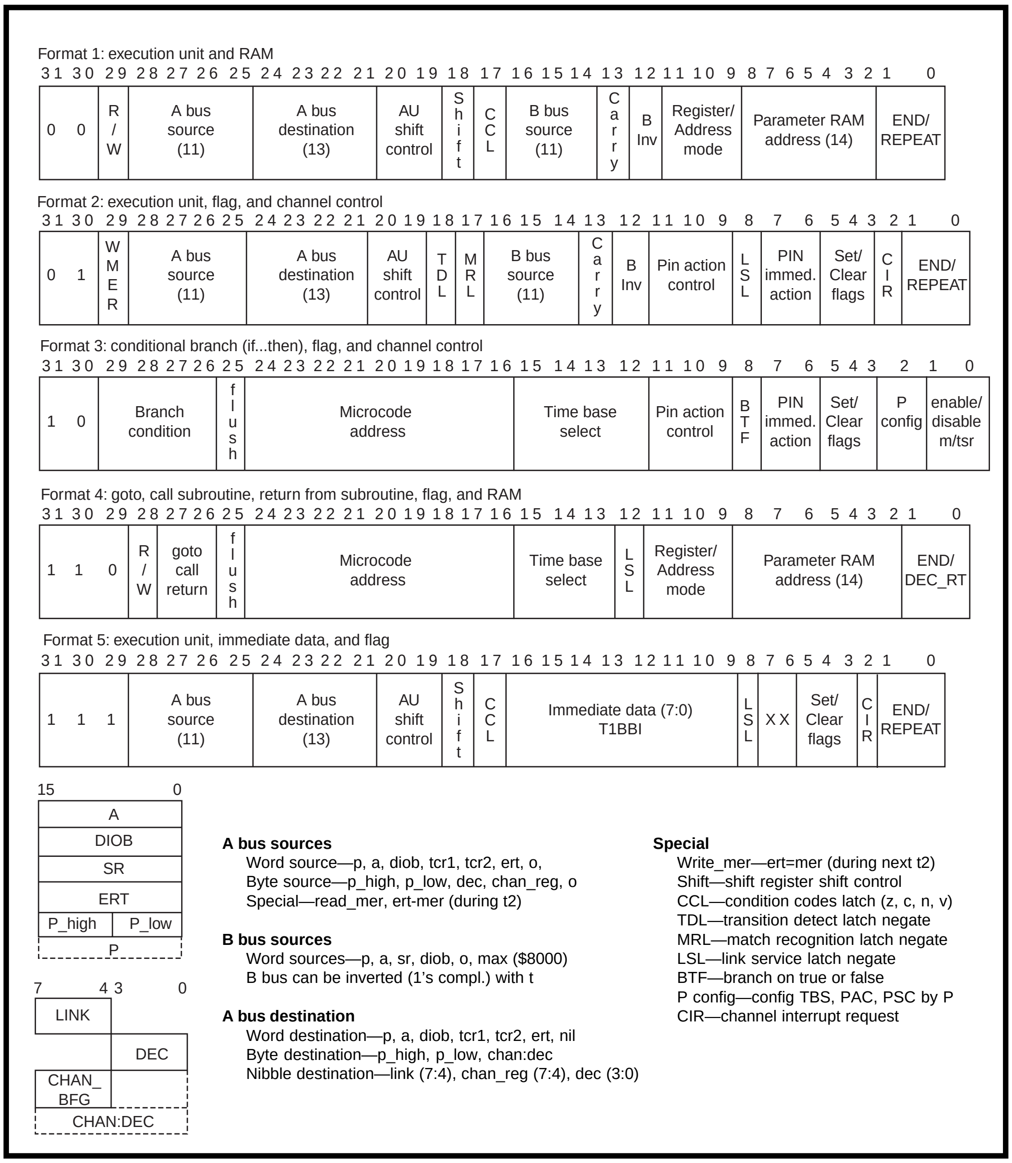
The micro-instruction `au := diob +para0.` (where `parm0` is parameter RAM location zero of the active channel) appears to add the `parm0` value to DIOB and store it in A. But, it doesn't give the desired result. The desired result is obtained by:

```
ram p:=parm0.
au a:=diob +p.
```

Take a look at the instruction formats shown in Figure 4. Note that all of the `au` instructions are format 1 or 2, except for the `au` instructions that have immediate data (format 5).

Knowledge of the instruction format is important because you can identify valid instructions. The assembler catches most of the invalid instructions, but it doesn't make any guarantees.

BINV is asserted in subtraction operations and in-



Format 1: execution unit and RAM

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	0	R/W	A bus source (11)	A bus destination (13)	AU shift control	Shift	CCL	B bus source (11)	Carry	B Inv	Register/ Address mode	Parameter RAM address (14)	END/ REPEAT

Format 2: execution unit, flag, and channel control

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

0	1	WMER	A bus source (11)	A bus destination (13)	AU shift control	TDL	MRL	B bus source (11)	Carry	B Inv	Pin action control	LSL	PIN immed. action	Set/ Clear flags	CIR	END/ REPEAT

Format 3: conditional branch (if...then), flag, and channel control

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	0	Branch condition	flush	Microcode address	Time base select	Pin action control	BTF	PIN immed. action	Set/ Clear flags	P config	enable/ disable m/tsr

Format 4: goto, call subroutine, return from subroutine, flag, and RAM

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	0	R/W	goto call return	flush	Microcode address	Time base select	LSL	Register/ Address mode	Parameter RAM address (14)	END/ DEC_RT

Format 5: execution unit, immediate data, and flag

31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

1	1	1	A bus source (11)	A bus destination (13)	AU shift control	Shift	CCL	Immediate data (7:0) T1BBI	LSL	X X	Set/ Clear flags	CIR	END/ REPEAT

A bus sources
Word source—p, a, diob, tcr1, tcr2, ert, o,
Byte source—p_high, p_low, dec, chan_reg, o
Special—read_mer, ert-mer (during t2)

B bus sources
Word sources—p, a, sr, diob, o, max ($8000)
B bus can be inverted (1's compl.) with t

A bus destination
Word destination—p, a, diob, tcr1, tcr2, ert, nil
Byte destination—p_high, p_low, chan:dec
Nibble destination—link (7:4), chan_reg (7:4), dec (3:0)

Special
Write_mer—ert=mer (during next t2)
Shift—shift register shift control
CCL—condition codes latch (z, c, n, v)
TDL—transition detect latch negate
MRL—match recognition latch negate
LSL—link service latch negate
BTF—branch on true or false
P config—config TBS, PAC, PSC by P
CIR—channel interrupt request

Figure 4—*When programming, keep this format in mind. A valid instruction must be in one of these formats.*

Valid operations

```
  := Assignment
  :=<< Assignment and shift left
  :=>> Assignment and shift right
  :=R> Assignment and rotate right
```

Valid constants

```
  0, +1, -1, $8000, $ffff.
```

Valid expressions

```
  Asource
  Asource + constant
  Asource + Bsource
  Asource + Bsource + 1
  Asource - Bsource
  Asource - Bsource - 1
  Asource + !Bsource
  Asource + !Bsource + 1
  Asource + #data
    (where #data == immediate data)
  #data
```

Valid A sources
Word sources—`a,diob, p, sr, tcr1, tcr2, ert, nil`
Byte sources—`p_high, p_low, chan_dec`
Nibble sources—`link, chan_reg, dec`

Valid B sources
Word sources—`a,diob, p, sr`

Table 1—*These are valid operations and expressions that are used with the `au` subinstruction. Note the valid A and B bus sources.*

verts the B bus data. Carry in (CIN) is asserted in subtract operations, when the 1 and $8000 constant are specified.

Valid `au` subcommands are:

- `au a:=1.`—assign the value `1` to `a`
- `au a:=p+diob.`—add `p` to `diob` and assigns results to `a`
- `au p:=a+diob +1, ccl.`—add `a` to `diob` plus `1` and assigns results to `p` and latches the condition codes
- `au diob:=#34`—assign `p` with the `34`

The only invalid `au` subcommand is `au a:=p+tcr1.` because the B bus doesn't have access to tcr1.

CALL

The `call` subinstruction branches to a subroutine. The syntax is `call label {,flush | , no_flush)`.

The `flush/no_flush` option determines whether the next subinstruction after the call is executed before the branch to the subroutine or on return from the subroutine (see Figure 6). This detail has caught me more than once.

For example, consider:

```
au a:=1.
call sub1, flush.
au a:= diob.

au a:=1.
call sub1, no_flush.
au a: = diob.
```

In the first example, the A register enters `sub1` with the value of 1. In the second example, it enters sub1 with the DIOB value.

The `flush` and `no_flush` are necessary because the TPU prefeteches the next inline subinstruction. `no_flush` is the default condition.

DEC_RETURN

`dec_return` returns the program sequence from a subroutine when a certain number of micro-instructions in the subroutine have been executed. The number of micro-instructions is set by the value in the decrementer (DEC) when the subroutine is called.

Every time a micro-instruction in the subroutine is executed, the value of the decrementer is reduced by one. When it reaches zero, the program sequence is returned. You need to load `dec` first before calling the subroutine:

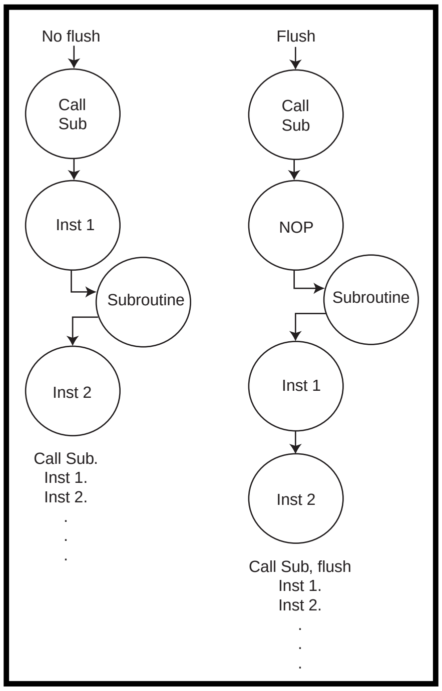


Figure 6—*When the call to `sub` is made with `no_flush`, Inst 1 is executed prior to the execution of the subroutine. But, when `flush` is included in the call, Inst 1 is executed after the subroutine is complete.*

```
au dec:=#6
call SUB1,flush:dec_return.
label1 next command
```

Six commands in SUB1 are executed. Then, the program returns to `label` and executes the next command.

The code block called doesn't have to be a subroutine. It can be any block of code identified by a label. This feature enables the programmer to save code space, which is somewhat limited in the TPU.

Let's say that somewhere in the code you perform the operation:

```
label1
ram p:= para1;
diob:=para2.
au a:=p + diob.
ram a:>parm0.
```

This operation adds parameter RAM values in locations one and two and stores them in location zero. If, in another part of the microcode, you wish to use the exact code, you can use `dec_return`:

```
au dec:=#4.
call label1,flush:dec_return.
next command
```

OTHER INSTRUCTIONS

`end` controls the end of a state, ends the current state once it is completed, and returns control to the hardware scheduler. The syntax is simply `end`, and it can be used with the `au`, `chan`, and `ram` subcommands.

The `goto` subinstruction branches unconditional to the specified location. The syntax is `goto label{,` `flush | no_flush}`, which is similar to the `call` subinstruction.

`if` branches conditionally to a specified location. The syntax is `if {condition := condtion_value} then goto label {,flush | no_flush}`.

The possible conditions are Less_than, Low_same, V, N, C, Z, Flag0, Flag1, Flag2, TDL, MRL, LSR, HSQ1, HSQ0, PSL, and PIN. Possible condition values for the `if` subinstruction are 1, 0, True, and False.

The `if` subinstruction is similar to other if branches found in standard assembly language (with the added concern of the prefetch taken care of by the flush operation). The instruction `if a:= less_than diob goto label, flush.` reads: if A is less than or equal to diob, then branch to label.

`NOP` performs no operation, but it is useful in terms of timing. For example, you can use it to ensure that data is stetted with respect to time slices.

The `ram` subinstruction reads or writes to parameter RAM. Only execution unit registers P and DIOB can access parameter RAM, and all accesses are 16 bits. The syntax is:

```
ram ram_reg r/w ram_address.
```

where `ram_reg` is either P or DIOB, and `r/w` indicates whether it's a read (<-) or a write (->). `ram_address` is the current channel's parameter RAM, including prm0 to prm6.

There are several ways to address parameter RAM, but the most common are `ram diob <- prm5` and `ram diob -> prm5`. These two subinstructions show how to write into and read from the parameter RAM, location 5, of the current channel. In `ram p -> (2,3)`, `p` is written into parameter RAM location 3 in channel 2.

`repeat` places a micro-instruction under the control of the decrementer (DEC). The micro-instruction is executed the number of times set by the decrementer, plus one more.

`repeat` is similar to `dec_return`, only there's no branch. In this example, `au a:=a+p` executes seven times:

```
au dec:=#6.
repeat;
au a:=a+p.
```

`return` returns control of the program to the address stored in the return address register (hidden from user). It's a standard return instruction with the condition (like all sequencing in microcode) that the prefetch must be taken care of with `flush`. The syntax is `return {,flush | no_flush}`.

Because there's only one return address register, calls cannot be nested:

```
return.
au a:=diob.
```

Here, the `return` jumps to the location stored in the return address register and sets A equal to DIOB.

ALMOST THERE

Once you understand the few idiosyncrasies associated with microcoding, it becomes clear that programming in microcode isn't much different than programming in a higher level language.

Next month, I'll look at the last subinstruction—`CHAN_REG`. I'll also discuss scheduling, entry points, and microcode programming examples. ▪

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

REFERENCES

T. Harman, *The Motorola MC68-332 Microcontroller*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Motorola, *Central Processor Unit*, 1990.

Motorola, *Time Processor Unit Macro Assemble Reference Manual*, TPUASM, Rev 3.

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

Motorola, *TPU Microcode Technical Training Course*, MTT39/CN, 1996.

SOURCE

Microcontroller
Motorola
(512) 328-2268
Fax: (512) 891-4465
www.mot.com

FROM THE BENCH

Jeff Bachiochi

Working with CoolPLD

A lot of manufacturers are seeing the advantages of JTAG, especially when it comes to in-circuit serial programming. Join Jeff as he takes a look at one of the low-cost tools that make it all possible.

I avoided programmable logic (PALs and PLAs) until International CMOS Technologies (ICT) introduced programmable electrically erasable logic (PEEL) in 1987. Before this, there were only OTP devices that gobbled up high currents compared to standard logic devices.

Companies were rather hush-hush about their proprietary programming algorithms. Special programming voltages were necessary, which required special programmers.

If you happened to be at a company with established programming hardware, you might be able to get a hold of the algorithms. But, as for you or me building a programmer, forget it.

The PEEL devices added a new twist to the equation—CMOS devices that were reprogrammable. What did that mean to the designer using programmable logic devices in a design? Reprogrammability gave the designer a level of flexibility that was never achievable before.

Suddenly, all the logic between the input and output pins of the device could be altered without having to design a new PCB. That is, assuming the device was a DIP part and was inserted into a DIP socket on the PCB.

I was already convinced that the payback for using IC sockets on PCBs was justifiable, so using PEELs merged easily with my mindset. Now, I'm not trying to tell you that a programmable device can always fix the problems without PCB redesign, but in many instances, it does just that.

One of the simplest examples is using the programmable logic device as an address decoder. Perhaps you need to move or swap some I/O selects. A quick change to the logic equations can reconfigure any or all of the I/O without any change to the PCB. This procedure also saves cuts and jumpers to existing boards.

REAL ESTATE BROKER

Using programmable logic may have been thrust on you for reasons other than flexibility of reconfiguring the internal logic. Population density may have been a factor.

With programmable-logic devices, you can eliminate a large number of the standard bubblegum-logic devices. Although the programmable device costs more, it pays for itself in saved real estate both on the PCB and in the inventory room.

Today, programmable devices are growing in size and density. The complex programmable logic device (CPLD) not only has more I/O pins but also more macrocells. In fact, the number of macrocells available can outnumber the actual I/O pins. This added density can potentially eliminate even more support chips.

Your company may have already made the leap of faith into the world of SMT designs. SMT parts aren't easily socketed, so reprogramming these logic devices is a hassle. What's a designer to do?

Enter JTAG. (You probably knew I was going to say that.)

ISP, JTAG STYLE

Somewhere along the development path, programmable-logic manufacturers started eyeing the flexibility of the JTAG testability standard. JTAG architecture enabled these manufacturers to solve the in-circuit programming problem by simply adding additional programming logic and allowing control via JTAG extensions.

By increasing the instruction register size (bit width), you allow additional

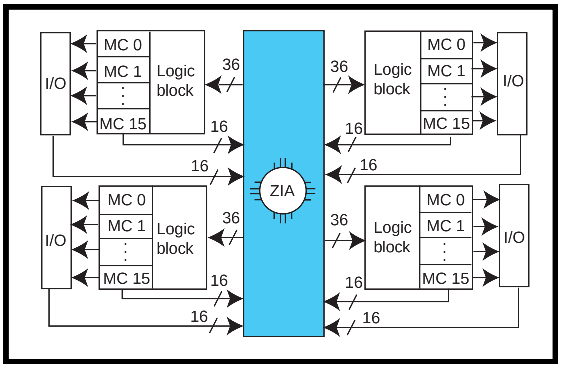


Figure 1—*The XPLA architecture shows how the logic block and macrocells are interconnected using a cross point switch (ZIA).*

programming instructions. These instructions enable an additional data register called the ISP register.

The ISP register is how the programming information actually gets to the array fuses (E^2). Voilà—in-circuit programming via the JTAG port.

Philips Semiconductor is one manufacturer of CPLDs. They have a family of fast (almost) zero-power (FZP) devices with JTAG ISP in 3- and 5-V versions.

These FZP devices use extended programmable-logic arrays (XPLA) to achieve macrocell propagations in the low nanoseconds while keeping currents in the low milliamperes at speeds above 1 MHz. Figure 1 gives you an idea of how the XPLA is laid out.

XPLA ARCHITECTURE

The PZ3065 and PZ5064 (3 and 5 V, respectively) are the smallest CPLDs that Philips manufactures with the JTAG ISP. They are packaged in various package sizes and styles from 44-pin PLCC to 100-pin QFP.

The packages with more pins offer more I/O without necessarily increasing the macrocell count. If you need more facilities, the larger pin packages let you substitute a higher density CPLD without having to redesign the board.

The XPLA architecture is made up of multiple logic blocks (LBs), which are interconnected by a zero-power interconnect array (ZIA). The ZIA is a large row/column array consisting of inputs to each LB and feedback from each macrocell and I/O pin (see Figure 1).

Figure 2 shows the PAL array of 16 individually programmable five-input AND gates (built for speed) that make up each LB. Each gate is OR'd into one of 16 macrocells along with a programmable PLA array of 32 AND/OR gates used when there's a need for increased product term density. The ZIA also supplies six possible control terms in support of the macrocells.

Each macrocell consists of a flip-flop that is configurable for either D or T types (see Figure 3). The clock to the flip-flop can be synchronous or asynchronous and can clock off either the rising or falling edge.

Up to four different clocks are available to the macrocells. They can be from synchronous external sources or asynchronous internal macrocell outputs. Powerup initializes all F/Fs into an initial 0 state, but preset and reset inputs can be implemented using two of the four control lines from the ZIA.

The remaining four control terms can be selected as output-enable control. Notice that the macrocell can be bypassed if necessary. Also, all outputs have a global tristate (*GTS) control.

GTS control, when enabled, permits all the pins on the device to be tristated by a single input pin. This arrangement releases the driving potential of all outputs to aid in testing (which could also be accomplished via the JTAG facilities).

Feedback to the ZIA comes from the macrocells' output and the I/O pin, with the output driver between the two. When the pin is used as an output, it's driven from the macrocell (via its output driver) or the ZIA.

When the pin is used as an input, the macrocell's output driver is disabled. The input then goes to the ZIA, and the macrocell's output is also fed back to the ZIA.

DESIGN TOOLS

Philips has a full line of tools starting with a $99 Coolrunner XPLA designer kit. To simplify the hardware side, a $50 ISP prototype board lets you see some immediate results. The ISP board comes with a PLCC PZ5128 CPLD device, a two-digit seven-segment LCD, and lots of prototyping area.

The XPLA designer software is a Windows-based suite of tools that carries you through the five steps of a CPLD design—design definition, device fitting, functional simulation, timing simulation, and programming. The XPLA designer produces a JEDEC output file that the ISP prototyping board and programming software use to program a device through the JTAG port.

The process begins with the design definition. XPLA designer uses a hardware descriptive language (HDL) for design entry. This HDL is supported in one of three formats—Boolean equations, state machines, and truth tables.

A Philips hardware design text file (.PHD) is created to hold the design definition. This file contains three basic sections—the header, declarations, and logic description. The header is information (e.g., title and special notes of interest) about the design.

The declaration section is where all of the I/O pins and internal nodes are defined and the constants, variables, and macro functions are declared. A macro is a reusable predefined function such as an octal latch, which is made up of eight individual latches with the appropriate control logic. Because this device may be used multiple times in the same CPLD design or in future designs, it makes sense to save it as a macro function.

Naturally, it takes time to learn all the shortcuts. The most important con-

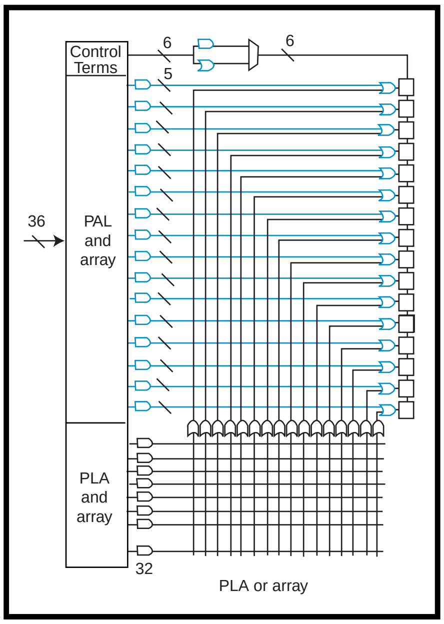


Figure 2—*The logic block section consists of PAL and PLA arrays, providing a combination of speed and increased product term density.*

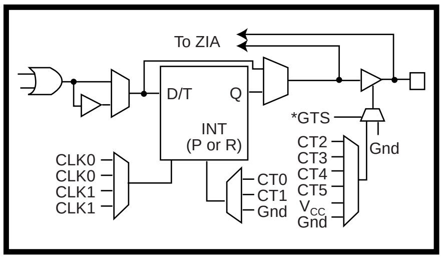


Figure 3—*The macrocell has a variety of clocking options as well as feedback before and after the I/O buffer.*

cept here is that every net of the design must be identified for the compiler to see the whole picture.

The logic description holds all the equation information on how the output signals respond to various input signals. Equations can be entered in Boolean format (e.g., a simple two-input AND gate where `ANDOUT = ANDIN1 & ANDIN2`). Listing 2 shows the truth-table format for the same gate.

The last equation format is that of the state machine. This registered equation uses a clock for state transitions. The equation to determine direction of a biphase optical encoder might follow the form shown in Listing 1.

SIMULATION

Once the `.PHD` file has all the proper information, the file can be compiled for a specific part. There are several options that simplify compilation (i.e., pin assignment). The compiled file can then be fitted into the part yielding a resource report (`.FIT`), a timing report (`.TIM`), and the JEDEC file (`.JED`) used to program the device.

The XPLA designer's simulator provides a graphical logic-analyzer view of each input, internal node, feedback, and output signal. The inputs can be stimulated through an interactive waveform editor or vector file. Once the compiled design is fitted to a part, the simulator gives true timing characteristics based on the selected device.

ISP PROGRAMMING

The XPLA ISP PC programmer software enables you to bulk erase, blank check, program, verify, read device IDs, secure devices, program, and read the user signature register.

The JTAG download cable (which comes with the ISP programmer) connects from the ISP's JTAG port to any parallel printer port on your PC.

Because JTAG devices are designed to be daisy chained, if each device's JEDEC file doesn't know where its device is on the daisy chain, you have a problem. Issuing commands and having

Listing 1—*Sometimes a state-table format is more appropriate, as in this example for an optical encoder.*

```
state_diagram  sreg;
state s0:
  when (phaseB==1 & phaseA==0) then (dir=1)
  else when (phaseB==0 & phaseA==1) then (dir=0)
  if (phaseB==1 & phaseA==0) then s2
  else if (phaseB==0 & phaseA==1 then s1
  else s=0
state s1:
  when (phaseB==0 & phaseA==0) then (dir=0)
  else when (phaseB==1 & phaseA==1) then (dir=0)
  if (phaseB==0 & phaseA==0) then s0
  else if (phaseB==1 & phaseA==1 then s3
  else s=1
state s2:
  when (phaseB==1 & phaseA==1) then (dir=1)
  else when (phaseB==0 & phaseA==0) then (dir=0)
  if (phaseB==1 & phaseA==1) then s3
  else if (phaseB==0 & phaseA==0 then s0
  else s=2
state s3:
  when (phaseB==0 & phaseA==1) then (dir=0)
  else when (phaseB==1 & phaseA==0) then (dir=0)
  if (phaseB==0 & phaseA==1) then s1
  else if (phaseB==1 & phaseA==0 then s2
  else s=3
```

them go at the appropriate device becomes a nightmare. You must remember the state of each device in the chain and how many clocks it takes data to shift into and out of each device.

Fortunately, the ISP JTAG chaining software enables you to identify each device along with its JEDEC file. When you perform functions on specific devices in the chain, the software keeps track of the state of each device.

THE NEXT STEP

Designing with JTAG-compliant parts doesn't have to be difficult or expensive. Using reprogrammable devices may pay you back with dividends if you need to alter the logic later on.

Of course, Philips would love you to use their parts in your next board design, but suppose you need to use a combination of manufacturers' devices. Shouldn't be a problem, right? All JTAG-compliant devices can be used together, right? Right.

But, every reprogramable device has its own programming characteristics. So, the same programming algorithm the Philips programmer uses may not work with other manufacturers' devices.

Engineers at Altera came up with an open standard to specify ISP programming for PLDs. Their Jam language should enable programming equipment vendors to get the data and programming algorithm in the same Jam file.

A Jam composer would create the JTAG chaining files, and a Jam Player would interpret the Jam file and program the targeted devices. Oh boy, universal tools!

Listing 2—*This truth-table format can be used to describe a function.*

```
truth_table
([ANDIN1,ANDIN2]->[ANDOUT])
  [0,0]->[0];
  [0,1]->[0];
  [1,0]->[0];
  [1,1]->[1];
```

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.*

SOURCES

PZ3065, PZ5064
Philips Semiconductor, Inc.
(800) 326-6586
Fax: (423) 475-0411
www.philips.com

Jam tools
Altera Corp.
(408) 894-7000
Fax: (408) 544-6403
www.altera.com

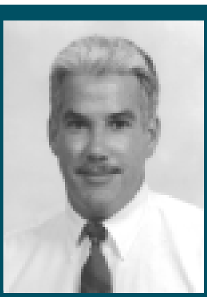
PEEL
ICT, Inc.
(408) 434-0678
www.ictpld.com

Flash Forward

SILICON UPDATE

Tom Cantrell

The 68HC908GP20



Tom thinks the 8-bit MCU market is getting back on the growth track and flash MCUs are going to be a big part of it. Join him as he takes a look at a new player whose name you may already be familiar with.

At the Embedded Systems Conference West in 1998, with a bit of help from Tom Starnes of the market-research outfit, Dataquest, I presented a session entitled "8-Bits: Onward and Upward."

Everyone knows I've got an 8-bit chip on my shoulder, so I asked Tom to provide independent confirmation of what I know in my gut. Despite the sizzle surrounding fancy chips, the 8-bit market is where it's at when it comes to embedded.

In fact, even with the slowdown in '98 (due to global market weakness and declining prices), the 8-bit MCU market is expected to get back on a growth track. Indeed, Starnes predicts that 8-bit MCU revenue will continue to exceed the combined total of 4-, 16-, and 32-bit (see Figure 1).

Don't forget that, given the recent price cutting, a revenue perspective understates the 8-bit MCU's popularity in terms of units. As Carl Sagan might have put it, we're talking "billions and billions" of 8-bit MCUs per year.

I explained the key technology trends that should invigorate design-ins. These trends include things like speed (the 100-MHz MCU from Scenix), mixed signal (already starting to see regulators, hi-V I/O, temp sensors, etc.), intellectual property (the 8-bit MCU market is even bigger if you count all the cores that will be buried in ASICs), and so on.

But, the most apparent trend is the widespread move to flash MCUs. Consider...

- flash pioneer Atmel says flash share is doubling each year, far faster than the overall market
- Hitachi claims 30% of their H8 volume is already flash
- Microchip is increasing their lineup from 2 to 22 devices
- Philips is dealing themselves into the flash fray via a deal with Macronix
- Analog Devices kicks off their foray into MCUs with a flash '51

It looks plenty rosy for flash MCUs, but what's wrong with this picture? Let's ask numbers guru Tom Starnes for a hint. Hey, Tom, who's the unquestioned 8-bit MCU leader?

Why, Motorola, of course. Their chips (notably the 'HC05 family) far outsell even the most popular competitors like '51s and PICs.

Motorola may be number one, but not when it comes to flash MCUs. Sure, they've dabbled with specialized 68k (68F333) and 16-bit chips like the 'HC12 and 'HC16, but they haven't had anything to offer the mainstream 8-bit market in the way of flash MCUs.

Motorola may be late to the party, but judging by the 68HC908GP20, they're making up for their tardiness with a grand entrance.

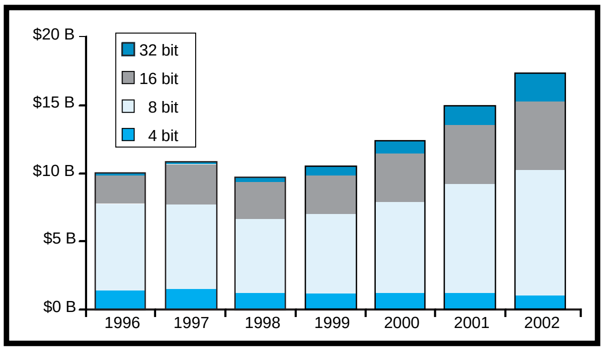


Figure 1—*Although it's been a bit soft of late, the 8-bit MCU market is expected to get back on track soon and remain dominant well into the future. (Source: Dataquest).*

A WHOLE NEW ’HC05

As a matter of fact, the ’HC908 is the subject of the *Circuit Cellar* Design99 Contest, which is already under way. No doubt, those of you who’ve already entered are up to speed on this chip, so get back to work! The rest of you, read on.

Let’s start with the big picture shown in Figure 2—’HC05 upward-compatible CPU, 20 KB of flash memory, and 512 bytes of SRAM. The peripherals include all the usual suspects: 33 I/O lines, 2 ´ 16-bit timer/counters, 8 ´ 8-bit ADC, UART, and SPI.

If you’re unfamiliar with the Motorola architecture, let me explain. Like the ’51 and PIC, the ’05 is one of those oldies but goodies. As best I can recall, the first ’05 chips were introduced about 20 years ago. So, it’s no surprise that the basic architecture, comprising an accumulator, index register, and stack pointer, is about as simple as it gets.

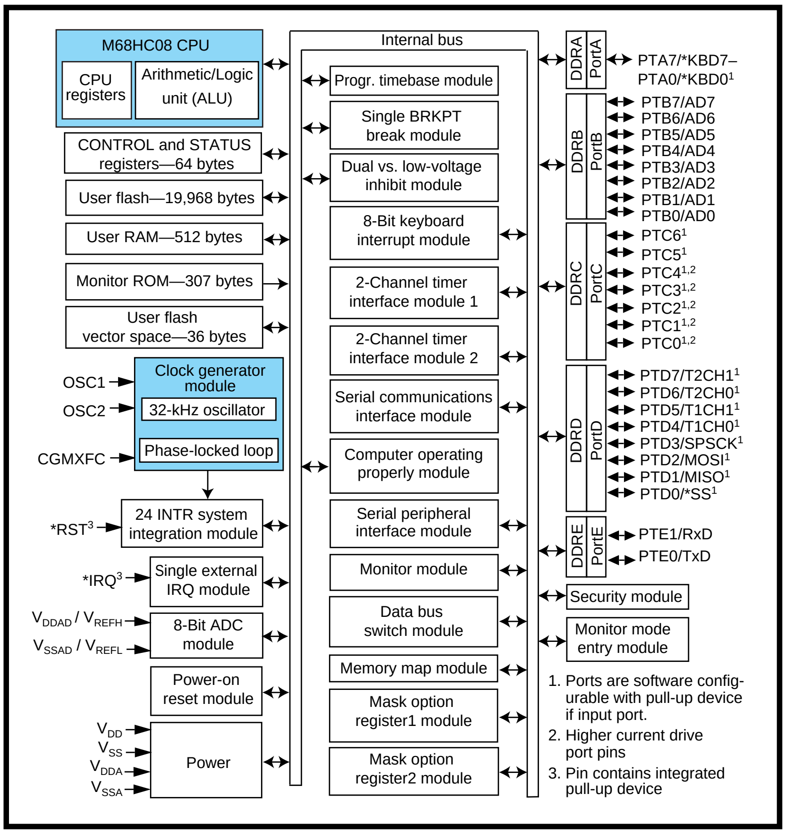

M68HC08 CPU
CPU registers
Arithmetic/Logic unit (ALU)
CONTROL and STATUS registers—64 bytes
User flash—19,968 bytes
User RAM—512 bytes
Monitor ROM—307 bytes
User flash vector space—36 bytes
OSC1
OSC2
CGMXFC
Clock generator module
32-kHz oscillator
Phase-locked loop
*RST3
24 INTR system integration module
*IRQ3
Single external IRQ module
VDDAD / VREFH
VSSAD / VREFL
8-Bit ADC module
Power-on reset module
VDD
VSS
VDDA
VSSA
Power
Internal bus
Progr. timebase module
Single BRKPT break module
Dual vs. low-voltage inhibit module
8-Bit keyboard interrupt module
2-Channel timer interface module 1
2-Channel timer interface module 2
Serial communications interface module
Computer operating properly module
Serial peripheral interface module
Monitor module
Data bus switch module
Memory map module
Mask option register1 module
Mask option register2 module
DDRA
PortA
PTA7/*KBD7–PTA0/*KBD01
DDRB
PortB
PTB7/AD7
PTB6/AD6
PTB5/AD5
PTB4/AD4
PTB3/AD3
PTB2/AD2
PTB1/AD1
PTB0/AD0
DDRC
PortC
PTC61
PTC51
PTC41,2
PTC31,2
PTC21,2
PTC11,2
PTC01,2
DDRD
PortD
PTD7/T2CH11
PTD6/T2CH01
PTD5/T1CH11
PTD4/T1CH01
PTD3/SPSCK1
PTD2/MOSI1
PTD1/MISO1
PTD0/*SS1
DDRE
PortE
PTE1/RxD
PTE0/TxD
Security module
Monitor mode entry module
1. Ports are software configurable with pull-up device if input port.
2. Higher current drive port pins
3. Pin contains integrated pull-up device

Figure 2—*Although it’s descended from the humble 68HC05, the ’HC908 delivers a lot more than you might expect from an entry-level micro, including a big chunk (20 KB) of flash memory.*

The more recent ’08 represents a midlife tweak of the ’05 design. It’s still simple, but with a bit more oomph, especially when it comes to toting the load of C.

Upgrades include boosting the index register from 8 to 16 bits as well as enabling all 16 bits of the stack pointer and program counter for full 64-KB addressing. There are also extra addressing modes, the most appreciated being stack relative for subroutine parameter passing.

And, there are a bunch of instructions including stalwarts such as decimal adjust accumulator (DAA) for BCD, looping (Z80-like decrement and branch), and multiply and divide (five and seven cycles, respectively).

The standard device features premium specs such as a –40° to +85°C temperature range, 3–5-V ±10% V_{CC} and up to 8-MHz throughput (at 5 V; 4 MHz at 3 V). Branches and instructions that access memory using the fancy addressing modes take 3–5 cycles, but some of the simple register ops only take one cycle, legitimizing multi-MIPS performance claims.

Naturally, the ’HC908 is available in a 44-pin QFP for small-is-beautiful apps. But, Motorola also reminds us that DIP isn’t dead with their 40-pin version for the through-hole crowd (especially welcome for prototyping, not to mention design contests).

PECK O’ PERIPHERALS

MCUs have come a long way in terms of the quantity and quality of built-in I/O—and the ’HC908 is no exception. Indeed, you don’t have to get further than traditionally mundane items like reset and the clock generator to see what I mean.

Besides external reset (i.e., yank *RST low), the ’HC908 includes five additional ways to get the chip back on track including power on, watchdog timer, low-voltage inhibit, illegal opcode, and illegal address. Unlike lesser chips, there’s no detective work needed to figure out what happened. The source of a reset is explicitly identified in a status register.

Motorola has a knack with PLLs, and they put it to use by supporting a 32-kHz watch-crystal operation. The frequency is boosted on-chip as high as 32 MHz, which is four times the internal bus clock.

Yes, PLLs have gotchas, especially when they’re called on to increase frequency by a factor of a thousand. Lockup time, noise immunity, jitter, power consumption, and support circuits (see Figure 3) are all concerns.

Without discussing all the details of the 30 pages devoted to the clock generator in the datasheet, I’ll just say that Motorola does a pretty good job covering the bases. If you prefer, the PLL can be disabled in favor of a typical 4´ crystal or external clock configuration.

One handy feature of watch-crystal mode is the built-in timebase feature, which generates an interrupt at one of eight programmable rates between 1 and 4096 Hz. Using these interrupts to wake the MCU from its low-power wait and stop modes can eliminate the need for an external real-time clock.

As usual, most of the 33 (31 in DIP) I/O lines wear multiple hats. Twenty-four lines feature bit-by-bit selectable pullups. All ports feature at least 10-mA drive, with five lines capable of 15 mA. Eight lines can be configured as so-called keyboard inputs, with edge- or level-interrupt capability.

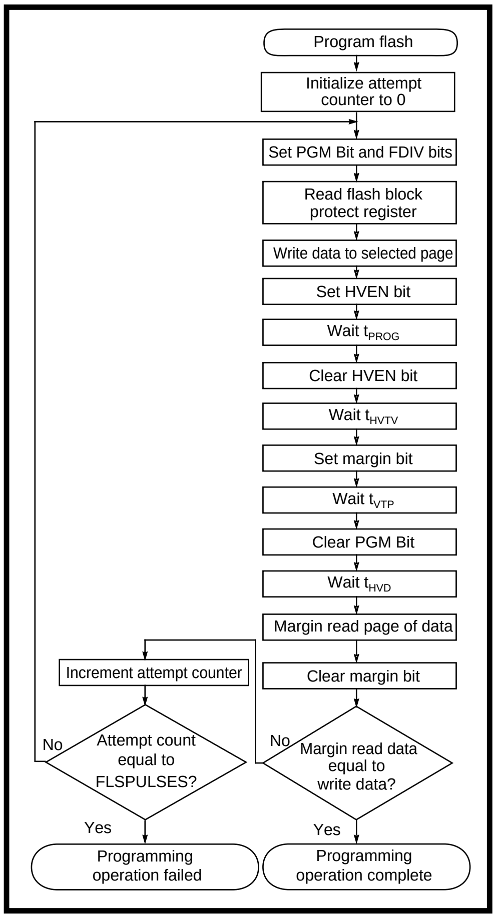
The eight-channel multiplexed 8-bit ADC is a successive approximation type with conversion time on the order of 20 µs. A conversion-complete interrupt is an option, as are single or continuous conversions. Pins not required for the ADC are available for general-purpose I/O use.

The two 16-bit timer/counters operate at up to the bus clock frequency of 125-ns resolution at 8 MHz and feature both input capture and output compare. For input capture, the counter latches when an active (programmable as high, low, or any) edge is detected on the input pin.

Output compare controls the output pin (programmable set, clear, or toggle) when the counter matches a preprogrammed value. The toggle-on-overflow mode is especially useful for PWM.

Those of you who've fiddled with output compare and PWM schemes have probably learned the hard way that glitches can occur because of asynchronous updates. For instance, setting a new compare value can generate runt or missing pulses depending on when the update occurs relative to the count.

The 'HC908 features a unique buffered mode that combines both channels in a ping-pong arrangement to get around the problem.



Program flash
Initialize attempt counter to 0
Set PGM Bit and FDIV bits
Read flash block protect register
Write data to selected page
Set HVEN bit
Wait t_{PROG}
Clear HVEN bit
Wait t_{HVTV}
Set margin bit
Wait t_{VTP}
Clear PGM Bit
Wait t_{HVD}
Margin read page of data
Increment attempt counter
Clear margin bit
Attempt count equal to FLSPULSES?
Margin read data equal to write data?
No
No
Yes
Yes
Programming operation failed
Programming operation complete

Figure 4—*The 'HC908 programming algorithm is finely tuned to match the requirements of the flash circuitry. Sloppy coding or trying to roll your own algorithm is definitely a no-no.*

Unlike the stripped-down UARTs in some low-end chips, the 'HC908 SCI has all the trimmings. For one thing, there's a built-in data-rate generator that's good for standard rates up to 115 kbps, depending on the MCU clock rate.

In addition to all the usual formats, the 'HC908 offers the 9-bit data wakeup mode popular in multidrop applications. It features a full complement of error detection, including overrun, framing, and parity, as well as some sophisticated glitch suppression.

Finally, the four-wire (select, clock, data in, and data out) SPI-clocked serial port is versatile enough to adapt to various standards (including I^{2}C), thanks to programmable master/slave mode, phase, and polarity.

Although it isn't Ethernet or Firewire, don't underestimate the SPI port's ability to move data around. Separate double-buffered receive and transmit registers can handle up to 4 or 8 Mbps (i.e., bus clock divided by two or one) in master and slave mode, respectively.

COOL FLASH

Flash memory is the point of the 'HC908, so let's take a closer look. The 20-KB array is organized in 8-byte pages, with eight pages composing a row, which corresponds to the minimum program-and-erase granularity.

The eight-to-one ratio isn't a coincidence but a requirement. The flash technology dictates that a row should not be programmed more than eight times before it's erased.

Eliminating the need for a separate V_{PP} programming voltage isn't easy, given the wide operating range (2.7–5.5 V). Nevertheless, with a built-in charge pump and voltage regulator, Motorola pulls it off.

Unlike other flash MCUs, the 'HC908 is completely self- or in-operation programmable. In a process akin to performing brain surgery on yourself, the chip executes software in one portion of the flash that programs the other. Lest things go awry, the 'HC908 incorporates a block-protect mechanism as malpractice insurance against a self-lobotomy.

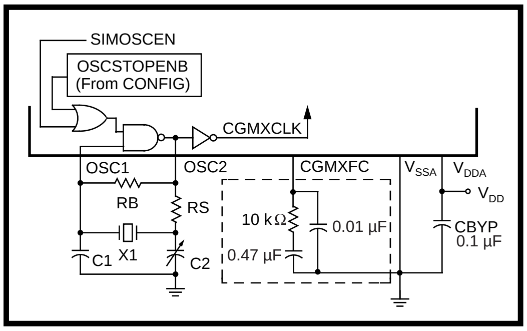


Figure 3—*Operation from a 32-kHz watch crystal is a nice feature, but it does call for a few external discretes, including a loop filter that optimizes PLL stability and tracking.*

The programming operation isn't for the faint of heart. As Figure 4 shows, it's a rather convoluted process with specific sequencing and timing that must be followed exactly. The algorithm relies on a margin-read scheme to program the flash memory just enough to gather the 10-year data-retention spec, without overstressing and compromising the 100-cycle/row write endurance spec.

The concept of self-programming raises the question of how to get a factory-fresh blank chip programmed in the first place. The answer is a tiny (307 bytes) built-in ROM monitor with just enough smarts to get past the who's-on-first dilemma.

The monitor includes only a few commands—read and write memory (i.e., SRAM), READ the stack pointer, and GO. On this minimal foundation, arbitrarily complex functions, such as programming the flash, can be built.

The monitor is entered in one of two conditions. First, if the chip is new or recently erased (i.e., the RESET vector is blank). However, normal operation of an already-programmed chip can be bypassed in favor of the monitor by applying a high voltage (V_{DD} + 2.5 V) to the *IRQ pin.

The monitor incorporates a security feature to discourage prying eyes. On entry, the host must send a sequence of eight bytes that match those stored at $FFF6–FFFD.

If you care about keeping your code secret, don't leave these locations blank. If the security check fails, attempts to read or execute from flash are blocked. But, the monitor permits the flash to be erased, so at least the absentminded won't have to throw the chip away.

HERE COMES DE BUG

Besides flash-memory programming, the monitor provides the hooks for a low-cost debugging scheme, which Motorola exploits in their evaluation kit (see Photo 1).

At $295, the kit may seem pricey, but it's more upscale than most. Its sturdy metal enclosure base unit hosts the 'HC908 personality card and connects to a PC serial port. The QFP MCU on the personality card is socketed so the 100-cycle endurance limit isn't a showstopper for us crash-and-burn types.

A ribbon cable with a 40-pin DIP on the end targets your own app hardware as well as the requisite wall-wart supply, which includes mains adapters that offer worldwide compatibility.

Also, there are printed versions of the documentation, and the package includes a nice development suite from P&E Microcomputer Systems. It features an IDE (see Photo 2) wrapped around various components including an assembler, programmer, a soft simulator, an in-circuit simulator, and an in-circuit debugger.

The soft simulator represents the traditional approach, relying only on the PC to host a virtual chip. It's a good way to check out the chip and try some programming without spending the bucks (it's free via the web).

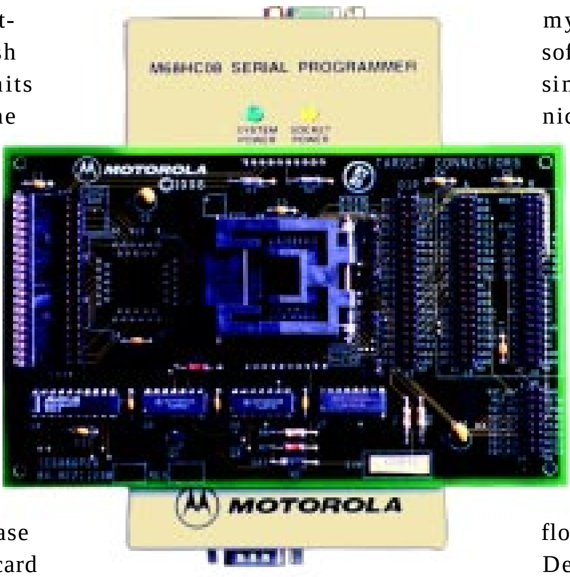


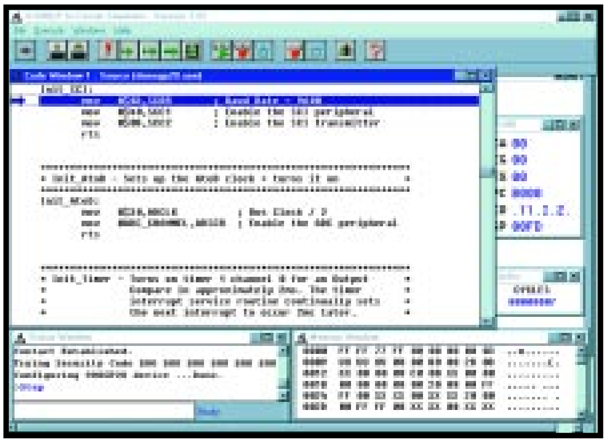
Photo 1—*The $295 development kit includes the hardware, software, and documentation needed to program and debug assembly-language apps for the 'HC908.*

The in-circuit debugger does a fairly good job of controlling and monitoring execution of an application-programmed chip. The process is aided by the fact that the 'HC908 incorporates a hardware breakpoint so the flash doesn't wear out by jamming in software breaks. There's no real-time trace, but the system does execute at full speed between debug events.

In-circuit simulation combines the total visibility of a soft simulator with real-world I/O. Instruction execution is completely pantomimed by the PC, but operations that talk to pins are passed over to the real chip. It's much slower than real time and, because of the to and fro, even slower than purely soft simulation. But, it does enable you to perform basic reality checks of your circuit design and wiring.

Although the documentation says the P&E software needs little more than Windows 3.*x* or 95/98 and 640 KB of RAM, it didn't pass my clunker (33-MHz '386) test. The software-only stuff (assembling, soft simulating, etc.) worked, but communication with the hardware was flaky.

No problems on my 300-MHz Pentium II, though, so the threshold of pain must be somewhere in between.



```
Init_SCI:
        mov   [illegible]          ; Baud Rate = [illegible]
        mov   [illegible]          ; Enable the SCI peripheral
        mov   [illegible]          ; Enable the SCI transmitter
        rts

*********************************************************************
* Init_[illegible] - Sets up the [illegible] clock + turns it on    *
*********************************************************************
Init_[illegible]:
        mov   [illegible]              ; Bus Clock / 2
        mov   [illegible]              ; Enable the [illegible] peripheral
        rts

*********************************************************************
* Init_Timer - Turns on timer 1 channel 0 for an Output             *
*              Compare in approximately 1ms. The timer              *
*              interrupt service routine continually sets           *
*              the next interrupt to occur 1ms later.               *
```

```
Contact Established.
Trying Security Code [illegible]
Configuring [illegible] device ...Done.
```

Photo 2—*The P&E Micro software included in the kit (and free on the web) features an assembler and simulator, as well as tools that work specifically with the Motorola hardware, like an in-circuit simulator, debugger, and programmer.*

FLASH TO THE FUTURE

Motorola didn't kick off the flash-memory trend, but their entry into the market will add fuel to the fire, especially since the chip is priced at $4.95. That's enough to get any designer's creative juices flowing, as I expect we'll see when Design99 entries start rolling in.

Hey, Elizabeth, any chance we forgot to put in the no-employees boilerplate in our contest rules? ▣

Editor's note: Sorry, Tom, no luck. But, I've heard that some people are having problems obtaining an 'HC908. Now they can check our web site for updated information on distributors.

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.

SOURCES

68HC908GP20
Motorola
(512) 328-2268
Fax: (512) 891-4465
www.mot-sps.com/sps/general/chips-nav.html

Development suite
P&E Microcomputer Systems, Inc.
(617) 353-9206
Fax: (617) 353-9205
www.pemicro.com

You don't need to pay for a development kit for Design99. www.circuitcellar.com features free development software and tools, and well as information on third-party sources.

PRIORITY INTERRUPT

Paper is Dead?

these days we have to deal with all the experts claiming that on January 1, 2000, there will be a complete social and commercial meltdown. If that weren't bad enough, apparently if I happen to survive into the new millenium, I won't have a job because printed magazines will be dead, too. Frankly, I could ignore all the Y2K hullabaloo. I've got a diesel backup generator, a full freezer, and a well-stocked wine cellar, but I'll be damned if I'm going to let that rubbish about print magazines being dead go by without commenting on it. Prophecies like that have as much merit as the one about the paperless office.

People who make these ridiculous prognostications must live in a cave. Television was supposed to kill off radio. Cheap fares would do away with the airlines. Faxes and FedEx would kill off the Post Office. And best of all, computers would do away with paper!

If we lived in a static environment, such predictions might have some credence. Fortunately, the opposite is true. Radical predictions fail to recognize that people do things a particular way because it's simply the most logical way to accomplish a specific task. The fact that a new alternative exists doesn't automatically signal the demise of all previous methods. Only an evolution in the task itself can do that.

The paperless office is a prime example. In 1982, when the early PCs started to gain a foothold, Americans used about 1.7 million tons of office paper. It should be less now, right? Well, in 1997, even with recycling, we were up to 4.6 million tons of nice white office paper! If you have any doubts, just look at the piles on your own desk.

The problem with these original predictions was ignoring that there would be something new and more interesting to put on paper. With the advent of the Internet, e-mail, and readily available wisdom on virtually any topic, we have more information than can be comfortably viewed on a video screen. Add to that the plethora of low-cost printers capable of spewing out a dozen pages a minute and you define a behavior where people simply collect piles of information for later viewing or just make lots of hardcopies because it complements normal computer activities (or the inevitable crash).

When someone says to me, "Print magazines are dead," I laughingly answer, "Which ones?" Yes, I know that absurd predictions like this presume that the demise of the media instantly makes us all obsolete. What's totally ignored, however, is the reason people pay hard money for a subscription. It's the *content*! I'd be in a lot of trouble if the only reason you subscribed to *Circuit Cellar* was because you have a paper fetish.

Don't get me wrong. I'm not fighting the trend. *Circuit Cellar*'s evolution will eventually include an online magazine, but the incentive to expand in that direction will be to enhance the current content with web-specific resources—not simply to reformat the magazine for electronic delivery. At the same time, if we're talking about obsolescence, I wasn't kidding when I said "which magazines?" In my opinion, electronic trade magazines have to look very closely at their "task" if they are to survive as presently structured.

Traditional trade magazine content is predominately advertiser-written application notes and promotional material. They charge exorbitant advertising rates to distribute this advertorial. Who among us doesn't have a pile of unread free trade magazines some-place?

In the past, manufacturers had few alternative vehicles to promote their products and readers had few places to look. With the advent of the Internet, however, manufacturers have the same and often more application editorial on their web sites. Engineers only have to enter a few key words into an Internet search engine to find this material. In view of the proliferation of engineering-related e-mail lists and other online vehicles, advertisers will be hard pressed to spend big dollars in a magazine without the informative content necessary to attract qualified readers.

Although converting all magazines from printed page to electronic screen is possible, that doesn't make it a mandated necessity. The staying power of a paid-circulation magazine (either online or printed) is first and foremost an issue of editorial quality, not the communication medium. Just like the issue of going from black and white to color magazines, when a redefinition of the task results in an improvement in the quality, the old ways won't be so much obsolete as the new ways will appear to be the logical evolution.

Steve

steve.ciarcia@circuitcellar.com