

EMBEDDED PC MONTHLY SECTION

CIRCUIT CELLAR[®] INK[®]

THE COMPUTER APPLICATIONS JOURNAL

#102 JANUARY 1999

EMBEDDED PROCESSORS

Custom Processors:
The Price/Performance Tradeoff

Communication Between
Multiple Processors

Timing Functions
with a TPU
Coprocessor

Embedded Java
Made Easy



TASK MANAGER

Everything Old is New Again



ut with the old, in with the new—isn't that what we tend to say at the beginning of a brand new year? But this time 'round, it gives me pause. It's true that we're starting a brand-new year, but it's the last one of the century, last one of the millenium, and that's an odd feeling.

OK, OK, Yes, I'm educated. Yes, I've heard that the new millenium doesn't really start until January 1, 2001. But give me a break. Most people don't think that way. Twelve months from now, everyone's either going to be wildly singing, "We're gonna party like it's 1999," or huddling under their mattresses with their savings and flipping out over the Y2k bug. We're not going to be thinking too seriously about pronouncements from all those prescriptivists who are so determined that we say things 100% accurately.

Well, at least *I'm* not going to. I'm more determined to celebrate this special, once-in-a-lifetime event. It's the event I've been waiting for since I was a child. The event I've been waiting for since I could add up how many years old I was going to be when the calendar did its cartwheel into 2000. So, with the clock ticking down and only 12 months to go, my question is, how to prepare for it?

In one respect, we (the publishing, editorial types) prepare for these things way ahead of time. For example, the Circuit Cellar 1999 Editorial Calendar has been set for many months already. But it's only now that you, the reader, are starting to see it in action. It's only now that you are starting to see how Circuit Cellar is going to celebrate this special year.

We start off with this issue on embedded processors, and when next month arrives, we'll see some real-world applications of fuzzy logic. And following that, it's a spring whirlwind of informative issues on automation and control, DSP, measurement and sensors, and communications.

As we get into summer, you'll be sitting in the sun, seeing what's cool in the robotics field as well as the latest techniques and tools for development and debugging. September brings the embedded apps issue (just right for heading back to school), and in October, we'll be hearing more about software algorithms. Getting back to dealing with the bumps and hassles of the real world, November deals with analog problems. And, in the last issue of 1999, the focus will be on embedded interfacing.

Hey, wait a sec! All this doesn't sound so new, does it?! Uh-oh. Do you mean to tell me that for all this talk of brand-new year and a special way of celebrating the century, we've scheduled more of what we've been doing all along? Hmm... Well, on second thought, that's probably the best possible way for us to pay tribute to the years that have treated us so well. And, it's a good way for us to remind you that we're committed to providing the same quality of editorial in the future.

I hope you enjoy a productive 1999, and I wish you a happy new year!

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR[®] INK[®]

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER

Steve Ciarcia

ASSOCIATE PUBLISHER

Sue Skolnick

MANAGING EDITOR

Elizabeth Laurencot

CIRCULATION MANAGER

Rose Mansella

TECHNICAL EDITORS

Michael Palumbo
Rob Walker

CHIEF FINANCIAL OFFICER

Jeannette Ciarcia

ART DIRECTOR

KC Zienka

WEST COAST EDITOR

Tom Cantrell

ENGINEERING STAFF

Jeff Bachiochi

CONTRIBUTING EDITORS

Ingo Cyliax
Ken Davidson
Fred Eady

PRODUCTION STAFF

Phil Champagne
John Gorsky
James Soussounis

NEW PRODUCTS EDITOR

Harv Weiner

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
(860) 872-3064

Fax: (860) 871-0411
E-mail: bobbi.yush@circuitcellar.com

ADVERTISING COORDINATOR

Valerie Luster
(860) 875-2199

Fax: (860) 871-0411
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](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.

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

Entire contents copyright © 1999 by Circuit Cellar Incorporated. All rights reserved. Circuit Cellar and Circuit Cellar INK are registered trademarks of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

- 14 Multiprocessor Communications**
Part 1: Methods for Communicating
Stuart Ball
- 20 Developing a Custom Integrated Processor**
Analyzing the Price/Performance Tradeoff
Joe Circello and Sylvia Thirtle
- 26 Using Java in Embedded Systems**
Vladimir Ivanovic and Mike Mahar
- 36 Music at Your Fingertips**
Guitar Effects via Remote Control
Hank Wallace
- 62 The PCL3013 Step/Servo Motor Controller in Action**
Gordon Dick
- 68**  **MicroSeries**
TPU
Part 1: A Coprocessor for Timing Functions
Joe DiBartolomeo
- 76**  **From the Bench**
Can You Feel the Beat?
Jeff Bachiochi
- 80**  **Silicon Update**
Wires, Wires Everywhere
The RF Solution
Tom Cantrell

Task Manager	2
Elizabeth Laurençot	
Everything Old is New Again	
Reader I/O	6
New Product News	8
edited by Harv Weiner	
INK On-line	12
Advertiser's Index/ February Preview	95
Priority Interrupt	96
Steve Ciarcia	
'Net Worth	

INSIDE ISSUE 102

- 42 Nouveau PC**
edited by Harv Weiner
- 47** RPC **Real-Time PC**
Embedded RT-Linux
Part 4: Developing Under Linux gcc/gdb
Ingo Cyliax
- 56** APC **Applied PCs**
In the Face of Medusa
Part 2: A Whole New Solution
Fred Eady

READER I/O

COULD I SEE YOUR LICENSE, PLEASE?

For some years now I've followed Linux and considered its use in embedded systems. I found the Embedded RT-Linux (INK 100) article to be both timely and informative.

However, an unfortunate error occurred where BSD versus GNU GPL licensing was discussed. Although the BSD kernels follow a different licensing scheme, the Linux kernel *is* GNU GPL protected.

Aside from the brief mention that components developed for a GNU GPL-protected system don't fall under the GNU GPL license, the problem concerns the use of the GNU C library in embedded (especially ROMed) applications.

This problem has received a lot of attention recently because the GNU C library is protected under a modified license, the GNU LGPL (Library General Public License). The GNU LGPL states that if you distribute binaries that are linked with the GNU C library, you must make available linkable objects (or the source) to your binaries so any downstream users can recompile and/or relink your code with any updated GNU C libraries that are available. This way, downstream users aren't locked into a particular revision of the GNU C library that was linked into a vendor's closed-source application.

Linking with a library was seen as producing a derivative work. This would have forced GNU GPL licensing issues on the original code, making the GNU C library almost useless for most commercial developers.

So that the FSF software would get some use, the GNU LGPL was created. Object code generated by the GNU C compiler from your source isn't considered a derivative work. Therefore, using the gcc compiler to generate code doesn't place you under the GNU GPL.

GNU LGPL was designed to deal with desktop workstation situations where the GNU toolset is installed by default, enabling end users to relink an application (given the linkable objects) or use dynamic linking with distribution-supplied shared libraries. But, an embedded system that boots from read-only media immediately runs afoul of the GNU LGPL. Choosing an alternative library and runtime enables you to ROM an application without violating the FSF licenses.

In the case of an embedded system that includes enough facilities (writable filesystems, and a console interface or remote network attachment), supplying linkable application objects and the tools and instructions necessary to relink the application (or use the shared libraries) should be sufficient to be compliant with the GNU LGPL.

This isn't meant to scare anyone away from freely available software, but a little time spent reading and understanding licenses is time well spent.

Dave New

newd@esi.com

You're right about the GPL issue. Because I'm not a lawyer, I wanted to concentrate on the technical issues and just mention that, in contrast to Net/FreeBSD, Linux is GPL licensed. Also, Pat Villani discusses some of the issues in INK 95-96.

Anyone developing commercial products should consult a lawyer for advice on legal issues about GPL licensed code or license agreements of other codes. Because the situation may be different for each project, this is the best way to make sure the intellectual properties of the project are protected. Unfortunately, that's the way it is in this business.

Ingo Cyliax

IS THAT ALL I GET?

I enjoyed the article by Alberto Ricci Bitti about the Graphing Data Logger (INK 99), but it left me wanting more—more description of the protocol that the FX9750G uses. Does the Casio FX7400G share the same protocol? What links are available for describing Casio features? What links did Alberto find? What kinds of projects are Circuit Cellar readers undertaking with respect to the Casio/PIC combination?

Gus Calabrese

wft@frii.com

Editor's note: Any thoughts on the topic? We'd love to hear from you. Send any correspondence to editor@circuitcellar.com.

Editor's note: Thanks to James Horton for noticing that the www.res.gatech.edu/~bdixon/rtdlinux and www.r52h146.res.gatech.edu/~bdixon/rtdlinux URLs mentioned at the close of "Embedded RT-Linux" (INK 100) didn't work. Although they were current when Ingo wrote the article, it doesn't take long for things to get outdated. Now, there's an official RT-Linux site (www.rtdlinux.org) with links to projects, documentation, and downloadable modules for different Linux distributions.

NEW PRODUCT NEWS

Edited by Harv Weiner



MULTIPLE TAPE BACKUP UNIT

Ultera Systems has announced plug-and-play mirroring controllers for producing two or more backup tapes as quickly as one. The **Imager** series of controllers appears to have a single drive or autoloader but actually mirrors the data being backed up onto two drives or two autoloaders, running them at their maximum recording speed. By cascading the devices, a user can produce four, six, or more copies simultaneously without a sacrifice in speed.

The Imager series includes two models. Imager 1 operates at up to 20-MBps burst rate over a SCSI I or SCSI 2 host channel, and records at up to 10 MBps onto two individual drives or two autoloaders. Imager 2 runs at up to 40 MBps from the host and to the drives and also supports the robotics for controlling tape libraries. Any SCSI tape drive and any backup software can be used with either system.

Imagers can be managed on-line through a GUI that is compatible with Windows 95, 98, and NT and with DOS. Imagers can also be operated off-line through their own control panel for tape copying, comparing or verifying. Internal half-height 5.25", desktop, and rack-mount units are available.

Pricing for the Imagers begins at **\$2445**.

Ultera Systems
(949) 367-8800
Fax: (949) 367-0758
www.ultera.com

BATTERY MONITOR IC

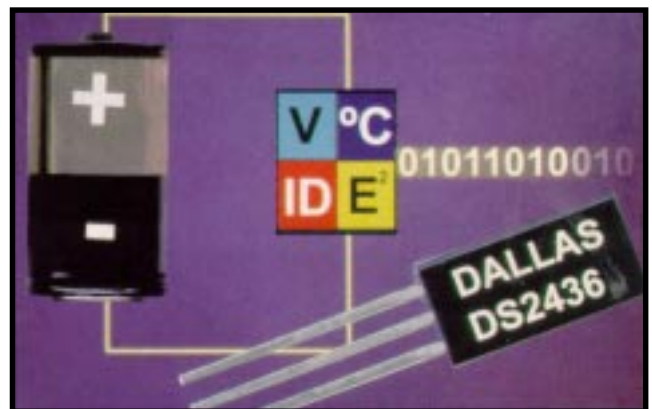
The **DS2436** battery identification chip provides a convenient method of tagging and identifying battery packs by manufacturer, chemistry, or other identifying parameters. The chip enables the battery pack to be coded with a unique two-byte identification number, and it stores information about battery life and charge/discharge characteristics in its non-volatile memory. Applications include cell phones, audio/video equipment, data loggers, scanners, and other hand-held instruments.

The DS2436 integrates a 10-bit voltage ADC and 13-bit temperature-sensing circuitry that monitors battery temperature without requiring a thermistor in the battery pack. A cycle counter manages battery maintenance intervals and helps the user to determine the remaining cycle life of the battery.

The DS2436 also measures battery voltage and sends the measured value to a host CPU. This feature is useful for end-of-charge or end-of-discharge determination or for basic fuel-gauge operation. Information is sent to and from the DS2436 over a one-wire interface, so the battery packs need only have three output connectors: power, ground, and the one-wire interface.

The DS2436 sells for **\$4.10** in quantities of 1000.

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



NEW PRODUCT NEWS

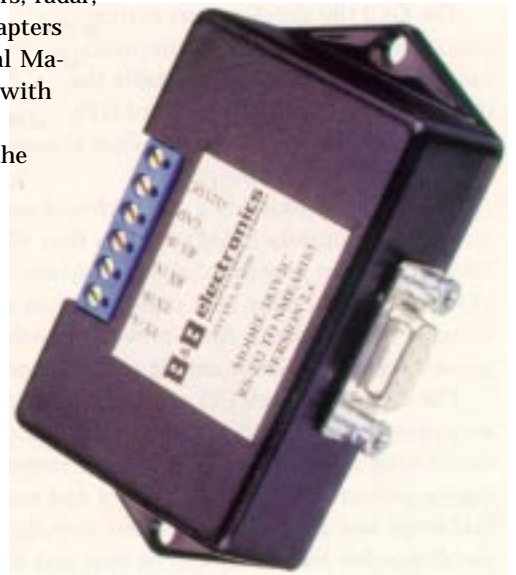
PC DATA INTERFACE ADAPTERS

A PC can now be used to log data from GPS satellites, depth sounders, radar, and other marine navigational devices with the latest data interface adapters from B&B Electronics. The plug-in connectors convert NMEA (National Marine Electronics Assn.) standard data signals so they can communicate with any RS-232/-422/-485 device, such as a PC or printer.

Two adapter models are provided to suit either the older NMEA or the latest NMEA specs. Model **183COR** converts the data signal from the older version of the specifications (NMEA0183 V.1.x) to EIA RS-232/-422/-485 signals. Model **183V2C** is for NMEA0183 V.2.x, which is the latest version of the NMEA specification. The **183V2C** converts one data signal in each direction between NMEA0183 and EIA RS-232.

The adapter model sells for **\$99.95** each.

B&B Electronics Mfg. Co.
(815) 433-5100
Fax: (815) 434-7094
www.bb-elec.com



PORTABLE EMBEDDED GUI

The **PEG** (Portable Embedded GUI) library is a professional-quality graphical user interface library created for embedded-systems developers. It is small, fast, and easily ported to virtually any hardware configuration capable of supporting graphical output. The default appearance of PEG objects is almost identical to common desktop graphical environments.

The PEG library is written in C++ and implements an event-driven programming paradigm at the application level. Each control type is built incrementally on its predecessor, enabling users to select and use only objects that meet their requirements. The PEG library provides an intuitive and robust object heirarchy. Objects may be used as provided or enhanced through user derivation.

PEG provides a set of hardware and OS encapsulation classes, so the PEG user interface can run as a standard 32-bit Windows application.

PEG includes two PC executable utility programs. PEG Font Capture enables users to convert standard

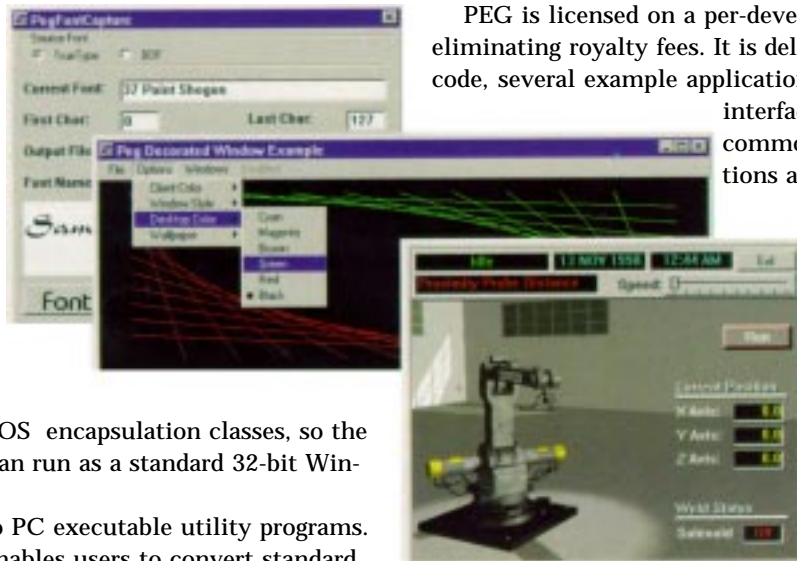
font files into a format required by PEG, and PEG Image Convert converts standard .pcx, .bmp, and .tga images into a compressed format supported by the PEG bitmap functions.

PEG is designed to work with any compiler/debugger combination. There are no internal restrictions on CPU type or hardware configuration. It currently supports standard EGA/VGA, SVGA, and LCD (320 × 240 × 4 color grayscale) video controller/display resolutions. PEG is designed to work with any combination of mouse, touchscreen, or keyboard input.

PEG is licensed on a per-developed-product basis, eliminating royalty fees. It is delivered with full source code, several example application programs, hardware interface objects for several common video configurations and input devices, and thorough documentation.

The cost of **\$5000** includes six months of free support.

Micro Digital, Inc.
(800) 366-2491
(714) 373-6862
Fax: (714) 891-2363
www.smxinfo.com



NEW PRODUCT NEWS

GLOBAL COMMUNICATION DEVELOPMENT SYSTEM

The **GC1100** development system integrates an embedded controller, GPS receiver, communications modem, and a user-command interface to enable the rapid design of tracking systems for a wide variety of GPS applications. The GC1100 is ideal for GPS-based fleet management, AVL, or asset-tracking systems.

The GC1100 contains a motherboard, an Ashtech G8 receiver, a Motorola 505SD modem that allows ARDIS Packet Data, an operator display interface, 32 digital user I/O lines, eight analog user inputs, and an active GPS antenna (15–30-dB gain). Also, prewritten software provides instant communication among all of the components.

The GC1100's high I/O count provides many options for user-specific applications. Digital and analog I/O enable monitoring several aspects of vehicle status, ranging from engine performance, cargo integrity and temperature, to fuel stops and door openings. Also, messages to and from the dispatcher and driver can be sent and displayed as text. The variety of I/O enables easy interfacing as well as connecting and monitoring digital and analog sensors.

The development system is available without a receiver or modem and can accommodate a variety of receivers.



The individual package includes a modem and receiver and sells for **\$1895**.

Z-World
(530) 757-3737
Fax: (530) 753-5141
www.zworld.com

NEW PRODUCT NEWS

SINGLE-CHIP DATA LOGGER

The **EDE702** serial LCD interface IC permits almost any text-based liquid crystal display screen to be controlled via a one-wire serial link. The chip, from E-Lab Digital Engineering, is ideal for embedded microcontroller applications where minimal I/O pin usage is desired.

The EDE702 enables full LCD control, including the creation of custom characters, scrolling text, cursor on/off, and so forth. With transfer rates of 2400 and 9600 bps as

well as selectable data polarity, the chip can interface to almost any microcontroller that is capable of sending asynchronous serial data.

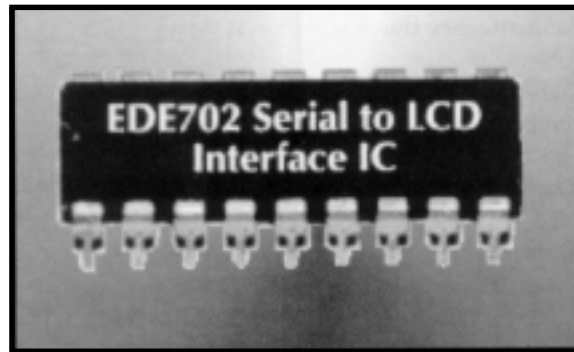
Another plus for designers is that this microcontroller connection can be made without any type of

voltage-level conversion hardware.

With the EDE702, circuit designers can easily add an LCD screen to their design without being concerned with the increased software overhead or I/O requirements that typically accom-

pany an 11-pin LCD interface. A serially controlled digital output pin makes the one-pin serial interface effectively a zero-pin interface.

The EDE702 is available in 18-pin DIP or SOIC packages, and it sells for **\$4.50** in quantities of 1000.



E-Lab Digital
Engineering
(816) 257-9954
Fax: (816) 257-9945
www.elabinc.com

NEW PRODUCT NEWS

UNIVERSAL SECURITY DEVICE

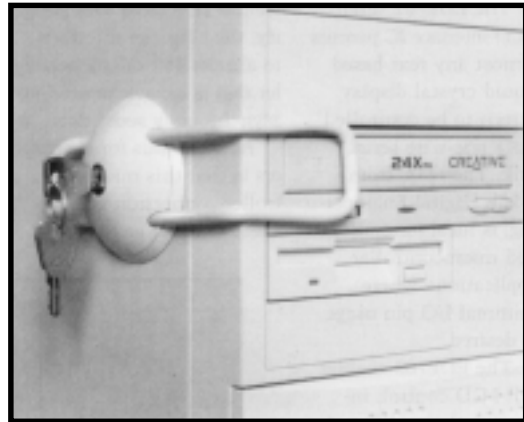
The **Safety Claw** is a universal lock for any drive on a PC or workstation. This device enables disk drives to be protected whether they're on a desktop or tower PC. All other drives, including CD drives, streamers, Zip drives, MO, or Syquest, can be reliably secured against unauthorized use whether they're installed in a PC or as an external drive.

The Safety-Claw's security plate is affixed to the PC or external drive casing, and its bar is inserted to block the drive. The Safety-Claw protects the PC or external drive from robbery if the user inserts a steel cable through the loop in the bar and attaches the protected device to another firm object.

Additional uses of the Safety-Claw include preventing a scanner or copier lid from being opened or avoiding the unauthorized use of an interface and/or removal of the cable on any device.

There are 200 different keys available for the Safety-Claw, and keyed-alike systems can be ordered. The steel bar of Safety-Claw is 6 mm in diameter, so it is extremely difficult to cut.

The Safety-Claw sells for **\$29.95**.



Interface Security Solutions Corp.

(800) 254-4392

(203) 743-1228

Fax: (203) 743-1458

www.crocodile.de

FEATURES

- 14 Multiprocessor Communications
- 20 Developing a Custom Integrated Processor
- 26 Using Java in Embedded Systems
- 36 Music at Your Fingertips
- 62 The PCL 3013 Step/Servo Motor Controller in Action

FEATURE ARTICLE

Stuart Ball

Microprocessor Communications

Part 1: Methods for Communicating

Communication is tricky no matter what, right? But when you have several processors involved, well, it just gets worse. Stuart begins this two-part series by looking at ways to get the messages between processors on a single backplane.



Although most embedded applications can be handled with a single processor, every now and then, you find a job requiring a system with two or more processors.

Nearly every multiprocessor design needs a way for the processors to communicate. In this series, I look at the different methods for communicating between processors and the various tradeoffs involved.

To start off, I'll look at useful approaches when two processors share the same PC board or backplane. Let's say the processor communicates with a higher-level system, like a PC, and distributes commands to a lower-level processor that controls a DC motor (see Figure 1).

CPU 1 talks to the host system, and CPU 2 controls a DC motor, under the

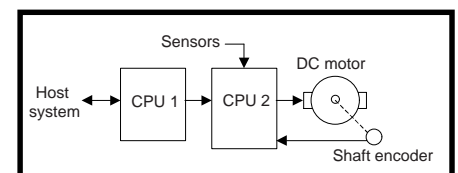


Figure 1—Although it's rather simple, this block diagram is representative of a typical multiprocessor application

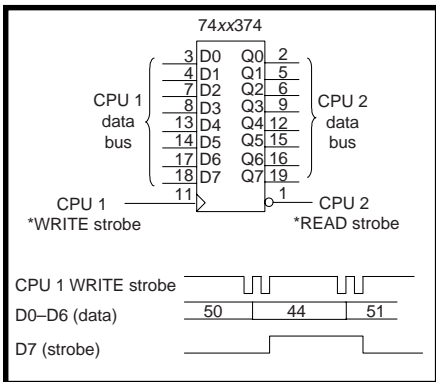


Figure 2—Note that CPU 1 performs two writes to the register for every byte transferred. The first write clocks the data in, and the second one toggles the most significant bit as a strobe.

control of CPU 1. CPU 2 senses the motor position via a shaft encoder and gets other sensor inputs as well.

In a typical real-world scenario, you might find this arrangement if CPU 1 needs to execute slow, complex tasks in response to the host, and CPU 2 has to execute fast, simple tasks to control the motor speed or position. You may find CPU 1 controlling multiple processors like CPU 2.

Clearly, CPU 1 must communicate with CPU 2 to get this job done. This requires commands like turn motor on, turn motor off, set motor speed to *x*, and start motor when sensor *y* goes active.

Figure 2 shows one method of communicating between processors. The circuit is an 8-bit register written by CPU 1 and read by CPU 2. The register is a 74xx374 (*xx* = LS, HC, ACT, etc.), but this scheme can be implemented in programmable logic or with any register that has tristate outputs.

The D inputs to the register connect to the data bus of CPU 1 (or to the lower 8 bits if CPU 1 is a 16- or 32-bit processor). The clock input (pin 11) connects to a write strobe from CPU 1. The write strobe is the same type you'd use to clock data into any register or a peripheral IC, and it goes low when CPU 1 writes to the specific address where the communication register is located.

The register's Q outputs connect to CPU 2's data bus. When CPU 2 wants to read the register, it generates a low-going read strobe (a decoded address strobe) at the register's Output Enable (pin 1). This strobe enables the tristate outputs, so CPU 2 to read the register data.

As Figure 1 shows, only seven bits of the register are used for transferring data. The eighth bit (D7/Q7) is a strobe that indicates when data is available.

The strobe bit is needed because CPU 2 may read the register anytime, including the exact instant when CPU 1 is writing to it. As you see from the timing diagram, when CPU 1 wants to change the register, it executes two write operations. The first write sets the lower 7 bits (D0–D6), and the second write toggles the strobe bit, D7, without changing D0–D6 again.

CPU 2 only reads data when it sees the strobe bit change state. So, if CPU 2 happens to be reading the register when CPU 1 is updating the data bits, CPU 2 won't see a change on the strobe bit and will ignore the data. If CPU 2 reads the register at the exact instant that CPU 1 is changing the strobe bit, it won't matter if CPU 2 sees the change, because the data bits are already stable.

Let's say you defined some commands for the control system as in Table 1. As Figure 2 shows, CPU 1 previously sent a Motor On command. This command is followed by a command to set the speed to 4, followed by a Motor Off command. Each new command changes the state of the strobe bit.

The advantage to this scheme is simplicity. A single 8-bit register is used and may be embedded in an FPGA or ASIC, or it may be implemented with a multibit parallel I/O IC. Of course, 16- or 32-bit processors can use wider registers.

A simple system may not even need a command structure. Instead, it can assign a separate bit to each function. Two-way communication can be implemented with a second register, written by CPU 2 and read by CPU 1.

Unfortunately there's no feedback to tell CPU 1 when CPU 2 has read the data. This drawback has serious implications for the system's throughput.

Say CPU 2 checks the register every 10 ms in response to a timer interrupt. CPU 1 can't send data any faster than this or CPU 2 may miss a byte.

If CPU 2 polls the register on an irregular basis, such as in a background loop, then the fastest that CPU 1 can send data is the longest time it takes

Code (hex)	Command
4x	Set motor speed to <i>x</i> (<i>x</i> = 0 to F)
50	Turn motor on
51	Turn motor off

Table 1—These are some examples codes for various control commands.

CPU 2 to execute the loop.

REGISTER WITH FLAG

To achieve faster throughput, the circuit in Figure 3 adds a set/reset flip-flop to the basic register circuit. The flip-flop is set when CPU 1 writes to the data register and reset when CPU 2 reads the register. The flip-flop can be constructed from a pair of NAND gates, or it can be half of a 74xx74 with the clock and D inputs grounded.

The flip-flop's output is provided to both processors so it can be read as an empty/full flag for the data register. It can connect to a status register or to a microcontroller port bit.

To use this scheme, CPU 1 writes something to the register. CPU 2 sees the data available (because the flip-flop was set) and reads it. CPU 1, polling the flip-flop output, sees it go low and knows that the register is empty and ready for another byte of data.

Now, the basic register circuit is more complex but the potential throughput is greatly increased. The maximum throughput is still limited, though, because both CPUs must poll the empty/full bit.

For example, if CPU 2 still polls for data once each pass through a background loop, the worst-case transfer rate is the same as in the single-register

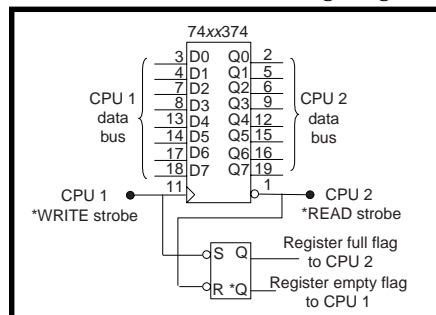


Figure 3—Adding a set/reset flip-flop improves the efficiency and speed of the register-based communication method by providing an empty/full status to the two CPUs

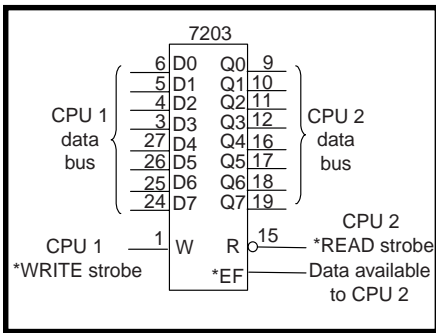


Figure 4—A FIFO provides a very fast interface

implementation. The maximum rate at which data can be transferred increases because CPU 1 can transfer data at the actual polling rate instead of at the slowest possible rate. The price is that both CPUs must have an available port bit or status input to read the empty/full flag.

INTERRUPT-DRIVEN SYSTEM

You can improve performance by connecting the outputs of the set/reset flip-flop of figure 3 to an interrupt on each CPU (instead of to status bits). CPU 2 gets an interrupt when the register is full, and CPU 1 gets an interrupt when the register is empty. Be sure you get the polarity of the interrupts correct if you make this change.

This scheme greatly increases the potential data throughput. The maximum data rate becomes the sum of the interrupt latency and processing time for both processors. The price for this approach is the need for one free interrupt on each CPU (two if a reverse path is also implemented).

There's also the potential for CPU 2 to get hammered with constant interrupts if CPU 1 has a lot of data to send. One way around this is to have CPU 2 turn off the communication interrupt

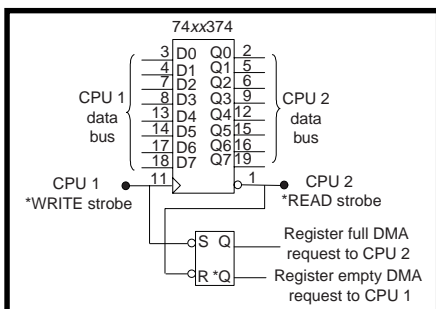


Figure 6—If the processors support it, adding DMA to the register-based scheme provides extremely fast, low-overhead communication.

during critical processing. However, this approach decreases the overall throughput.

FIFO

Figure 4 shows a FIFO interface. This example uses a 7203, which is a 2×9 -KB FIFO. The 7203 is an industry-standard part from a family of parts available in 512×9 -KB, 1×9 -KB, 2×9 -KB, and up. This example circuit doesn't show all the 7203 pins, just those we're interested in here.

The 72xx family of FIFOs contains an internal SRAM and logic to control access to the RAM. Data written to the FIFO input by CPU 1 is placed in the internal FIFO memory. Any time the FIFO is empty, the Empty Flag (EF) output is low. If the FIFO is not empty, EF is high.

The incoming data is stored so that CPU 2 reads it out in the same order it was written. So, the FIFO acts as a deep register that allows CPU 1 to write multiple bytes without worrying about how fast CPU 2 is reading them.

To use this method, CPU 1 typically writes a complete, multibyte message to the FIFO. When EF goes high (not empty), CPU 2 reads the data. This type of interface requires very little overhead from the processors. The rate at which CPU 1 sends data is not limited to the rate at which CPU 2 reads it.

The first drawback to this system is a throughput limitation. Although CPU 1 can send a message without worrying about how fast CPU 2 can read it, the average transfer rate can't exceed the capacity of CPU 2. If it does, the FIFO fills up and data is lost. So, the FIFO doesn't really increase the overall data throughput, it just decreases the overhead of transferring the data.

A second problem with the FIFO interface is time delays. If CPU 1 sends a command, such as Motor Off, there may be a delay before CPU 1 reads the message and acts on it. With register-based approaches, CPU 1 always knew CPU 2 was reading data as it was sent. But with the FIFO design, that's no longer the case.

The third problem is related to the second and involves message priorities. Suppose this hypothetical system had messages of differing priorities. The normal Motor Off command may

allow the motor to coast to a stop, but there's an Emergency Stop command that brakes the motor instantly.

If Emergency Stop is received, you presumably want to service it immediately. If the interface uses a FIFO, there's the possibility that commands can stack up in it, as shown in Figure 5. A low-priority command, like a speed change, can be in front of a command such as Emergency Stop.

Suppose that your command set consists of long, multibyte messages and that the software in CPU 2 reads the first byte of each command to see what kind of command it is. If the command is low priority, like a speed change, the software may decide to read and process the command later. This decision leaves the possibility that Emergency Stop won't be acted

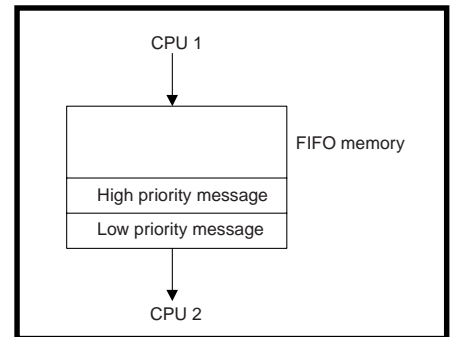


Figure 5—It's possible for a high priority message to stack up behind lower priority messages in a FIFO. This situation can cause the high-priority message to be serviced later than expected.

on right away if it's behind a low-priority command in the FIFO.

The solution to this priority problem is for CPU 2 to read all messages as soon as they are received. Lower priority messages can be stored for later execution, and high-priority messages can be executed immediately. The drawback is that all messages must be treated as high priority because any message could have a high-priority message stacked behind it.

DMA-BASED INTERFACE

Some microprocessors, such as the '186 and '386EX, have built-in direct memory access (DMA) controllers. For these applications, the circuit in Figure 6 eliminates nearly all of the disadvantages I've described so far. This circuit goes back to the register-and-flip-flop approach, but with a twist.

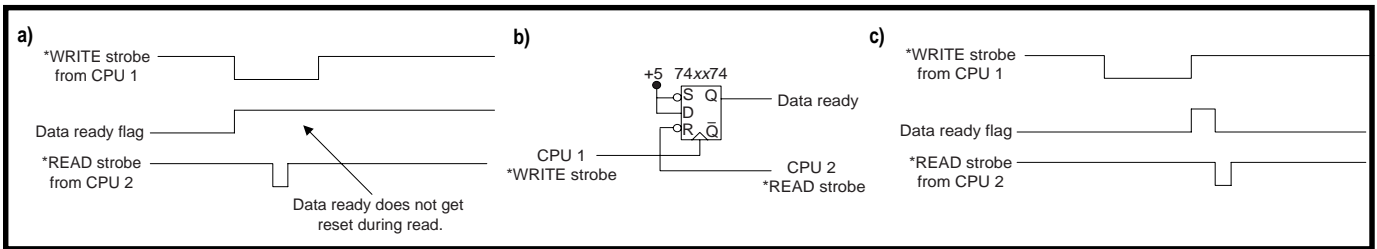


Figure 7a—If CPU 2 is much faster than CPU 1, CPU 2 may detect the register full condition and read the data while CPU 2 is still performing a write cycle. This results in the register full condition remaining active and CPU 2 reading two bytes instead of one. **b**—By connecting the CPU 1 write strobe to the clock input of the status flip-flop (instead of the SET input), the status flip-flop is not set until the end of the write cycle. **c**—The change prevents the race condition from occurring, regardless of the relative CPU speeds.

In Figure 6, the flip-flop outputs do not connect to status bits or to interrupt inputs. They connect to the DMA request signals on both processors. CPU 1 receives a DMA request when the data register is empty and CPU 2 gets one when the register is full.

Let's look at three ways to implement this interface in software. First, all messages have a predefined length (16 bits, 32 bits, etc.). Shorter messages are padded out to this length. CPU 2 sets up its DMA controller to transfer one complete message. When CPU 1 wants to transfer data, it tells its DMA controller to send the block of memory to the data register.

As each byte is sent, the DMA request to CPU 1 goes inactive and the DMA request to CPU 2 goes active. When CPU 2's DMA controller reads the byte, the DMA requests swap states again and CPU 1's DMA controller sends the next byte.

After the entire message is received, CPU 2 gets an interrupt from its DMA controller and processes the received data. The entire transfer is accomplished in a few tens of microseconds, although the processing may take longer.

If your application requires variable-length messages, you can define each message so the first byte defines the length. As before, CPU 1 sets up its

DMA controller to send the entire message. CPU 2 has already set up its DMA controller to transfer a single byte.

When the first byte of the message is sent, CPU 2 gets an interrupt (from its DMA controller) and reads the byte. CPU 2 then sets up its DMA controller to transfer the rest of the message based on the length byte. This method enables variable-length messages to be sent, but CPU 2 now has to service two interrupts for each message and the maximum transfer rate is slower.

The third method is for CPU 2 to set its DMA controller to transfer more data than the longest possible message. When CPU 1 sends data, it gets an interrupt (from its DMA controller) indicating that the transfer is complete.

CPU 1 notifies CPU 2, via another interrupt path, that a message is available. CPU 2 then reads the length from its DMA controller pointer registers and processes the message normally.

In a DMA scheme, CPU 2 sets up a block of memory as a buffer for the DMA data. For example, if each message is 16 bytes in length, CPU 2 can set up a 256-byte block of memory that contains 16 message buffers.

Using DMA also avoids the FIFO priority issue in two ways. First, the transfers are executed directly to memory in hardware, making the process of reading the data less of a bottleneck. Second, if CPU 2 uses multiple buffers, lower priority messages can be left in their buffers until they are acted on, whereas high-priority commands can be executed immediately.

Even if only one of your CPUs has built-in DMA, you can take advantage of this approach. The CPU with DMA can transfer messages using DMA, eliminating the overhead of polling or servicing one interrupt per byte. While you won't get the throughput of a

dual-DMA design, sometimes you get simpler software with this approach.

With any DMA-based approach, make sure the timing of setting and resetting the flip-flop meets the requirements for the DMA controller. The primary drawback to this approach is that one or both processors must have DMA capability.

The primary advantage is extremely fast data throughput, and although it's probably not worth changing processors just to use this technique, it can provide a fast communication path if the processors support it.

RACE CONDITION

All the variations of the register-and-flip-flop design are susceptible to a race condition if one processor is considerably faster than the other. In Figure 7a, CPU 2 is much faster than CPU 1, so CPU 2 sees the flip-flop get set and reads the data while CPU1 is still writing to the register. As a result, the flip-flop doesn't get reset properly.

A typical scenario where this might occur is if CPU 2 is a very fast DSP

communicating with a slower general-purpose microprocessor.

The solution: use a synchronous design (where everything is referenced to one of the CPU clocks) or use a clocked flip-flop. Figure 7b shows how a flip-flop like the 74xx74 would be connected to fix the timing problem.

Figure 7c shows the new timing. Because the write strobe from CPU 1 is connected to the clock input of the 74xx74, the data ready flag doesn't get set until the end of the write cycle, eliminating any timing conflict.

DUAL-PORT RAM

I also want to mention dual-port RAM (i.e., RAM that can be accessed by either processor). One option is to use an off-the-shelf dual-port RAM IC with two addresses and data buses. You can get controller IC's that convert standard RAM devices to dual port.

The second method is to use the existing RAM associated with one of the processors. This approach is simpler than an external RAM, but it can affect the throughput of both processors.

All of these approaches can be mixed and matched. For instance, if your application has a CPU 1-to-CPU 2 interface that requires long data messages at high rates, you might implement a DMA controlled register for that interface. The return path, from CPU 2 back to CPU 1, might carry only infrequent status bytes, so it may be a simple polled register interface.

Next time, I'll look at methods you can use if your processors must communicate over a greater distance. ☐

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

REFERENCE

- S. Ball, *Embedded Microprocessor Systems, Real World Design*, Butterworth-Heinemann, Newton, MA, 1996.

Developing a Custom Integrated Processor

Analyzing the Price/Performance Tradeoff

FEATURE ARTICLE

Joe Circello & Sylvia Thirtle

If standard processor configurations aren't quite what you need, consider a processor-independent core.

Joe and Sylvia bring to your doorstep a customizable core, so you can manipulate valuable variables in the price/performance equation



ew uses for advanced embedded microprocessors are emerging everywhere, especially in the highly competitive, fast-paced market of consumer electronics. Thanks to cooperative efforts with silicon vendors, embedded-system developers can manipulate powerful variables in the price/performance equation that were previously beyond their control.

Optimizing an embedded processor presents an earnest challenge and it requires the system designer to perform a delicate balancing act between performance and cost. Ultimately, this approach produces an embedded-processor solution that is fine-tuned for a given system and/or application.

DESIGN MODELS

With the traditional system design model, the engineer remains at the mercy of standard product offerings from the semiconductor vendor. A chipmaker's catalog of standard processor configurations may or may not include precisely what's required for a given application.

With a standard product, system designers may have to pay for functions

(and silicon) they'll never use. Additionally, the device may lack certain functions that could significantly enhance the system performance of a given application if integrated on-chip.

But, a new system design model is emerging, brought on by the availability of modular, fully synthesizable, process-independent microprocessor cores. For the first time, design engineers have unprecedented control over defining and configuring embedded processors.

Customizable cores, like the Motorola ColdFire family, can be cost-effectively tailored to meet the demands of specific applications. The ColdFire architecture was developed to address this class of applications.

Based on variable-length RISC technology, ColdFire combines the architectural simplicity of conventional 32-bit RISC with a memory-saving, variable-length instruction set. In defining the ColdFire architecture, Motorola incorporated a RISC-based processor design and a simplified version of the variable-length instruction set found in the 68k family.

The result is a family of 32-bit microprocessors suited for those embedded applications requiring high performance in a small core size. The ColdFire family provides balanced system solutions to a variety of embedded markets. Here are some of the basic philosophies that have guided all ColdFire designs.

When it comes to small, fully synthesizable processor cores, developments are on track with a publicly announced performance roadmap reaching 300 MIPS by the year 2001. Using compiled memory arrays and 100% synthesizable designs enables

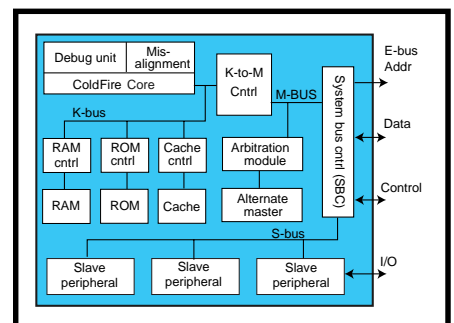


Figure 1—This generalized block diagram shows a custom integrated processor using a ColdFire core.

system designers to easily define CPU configurations.

Figure 1 depicts the standard ColdFire microprocessor configuration. The hierarchical bus structure and the modular architecture are apparent. You can add other logic, in the form of predefined macros, from Motorola's library or synthesize your own proprietary circuits.

MAXIMUM ARCHITECTURE

Fine-tuning a custom embedded processor for optimal price and performance requires some insight into the specific architecture's variables. The difficulty of this process is influenced by the sophistication of the silicon vendor's development environment as well as the system designer's ability to provide accurate, real-world application data for the target system.

For example, the system OEM may be able to provide information from a previous-generation system. The data can be a key piece of software that represents a critical execution path of the given application. If possible, the ability to extract the key software routines and recompile them for the target system makes the process much easier.

As an alternative, trace data captured from a previous-generation system can also provide critical information for sizing the processor's local memories (e.g., cache, RAM, ROM). These dynamic traces, whether captured from an earlier design or created by the application code running on a software simulator of the target system, are crucial for the price and performance optimization analysis.

PREDICTING PERFORMANCE

Although ratings for microprocessors are expressed in MIPS, this number often fails to accurately predict the performance of an embedded microprocessor system for a given application. Many times, these ratings need a "mileage you get may vary" disclaimer. Unless the effects of the memory subsystems are taken into

a) base CPI [cycles/inst] = summation {F(i) × ET(i)} + sequence-related pipeline stalls

b) effective CPI [cycles/inst] = base CPI + summation of memory factors + summation of system factors

c) effective CPI [cycles/inst] = base CPI + IC_miss × IF × IF_stall + OC_miss × REF × OP_stall

where the cache memory degradation factors include:

IC_Miss = Cache miss rate on instruction fetches (Miss/fetch)
 IF = Instruction fetches per instruction
 IF_stall = Time [cycles] the processor core is installed servicing an instruction fetch miss
 OC_Miss = Cache miss rate on operand fetches (Miss/OPFetch)
 REF = Operand references per instruction
 OP_stall = Time [cycles] the processor core is installed servicing an operand miss

d) effective CPI = base CPI + {(IC_Miss × IF) + (OC_Miss × REF)} × (2 + 11 + 0.6 × (12 + 13 + 14))

Figure 2a—Here's the simplified expression for the processor's performance measured by effective CPI. b—This generic expression defines performance as measured by effective CPI. c—This more detailed equation defines effective CPI performance for a processor with cache memory. d—And, this is the effective CPI equation for the ColdFir2 V.2 and V.3 processors.

account, these simplistic ratings can't accurately indicate performance.

Today, more precise performance estimates of a hypothetical or actual processor core can be made. By taking specific system and memory subsystem variables into account, this methodology provides a more accurate representation of completely different CPU configurations and architectures.

The predicted performance of a processor can be developed using an average-instruction-time methodology. In its simplest form, this cycles per instruction (CPI) metric represents the number of machine cycles per instruction and is calculated for a single-issue architecture as:

$$CPI \left[\frac{\text{cycles}}{\text{inst}} \right] = \text{summation} \{ F(i) \times ET(i) \}$$

where CPI is the average instruction time expressed in cycles per instruction, F(i) represents the dynamic frequency of occurrence per instruction, and ET(i) is the execution time for a given instruction i. By summing the product of relative frequency and execution time for each instruction type, the average instruction time for a processor executing any given instruction mix can be calculated.

Consider the definition of a base average instruction time (base CPI). Let the base CPI represent maximum processor performance strictly as a function of the instruction mix. Stated differently, this metric represents the processor's performance assuming the rest of the system (caches, memory modules, etc.) is ideal.

Figure 2a shows the base CPI where the summation product was previously defined and the sequence-related pipeline stalls include all pipeline breaks caused by the instruction sequence. You can calculate the base CPI by summing the product of the relative frequency of occurrence and execution time for each instruction type plus the sequence-related holds.

This base CPI provides a parameter to quantify the performance of a given processor microarchitecture. To convert this value into a more realistic measure of predicted system performance, you have to consider a series of degradation factors.

Let the effective average instruction time (effective CPI) represent this more realistic measure of performance. By quantifying the degradation factors associated with these other system components, the effective CPI can be calculated. As an example, the processor stalls resulting from cache misses typically represent the largest degradation factor in the effective CPI equation.

In Figure 2b, the calculated effective CPI is reached by summing the individual degradation factors. Let's

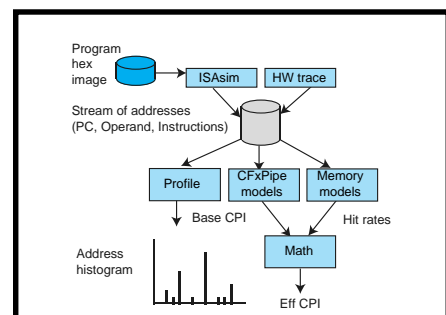


Figure 3—This diagram gives you an overview of the ColdFire performance-analysis methodology.

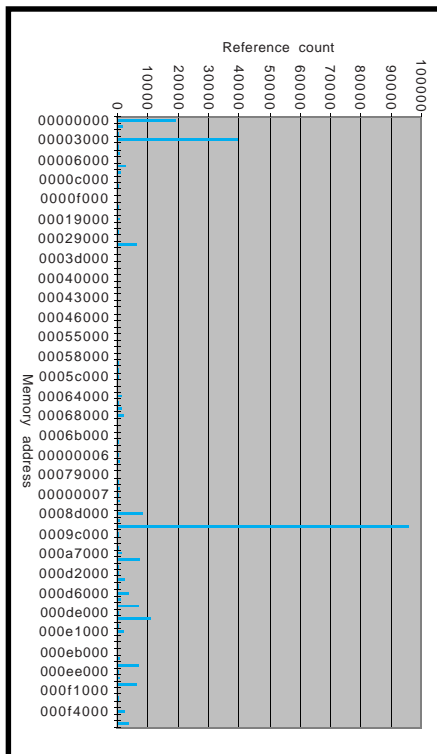


Figure 4—The operand address histogram is taken from a set-top box application.

define the memory subsystem factors as those associated with a cache memory, and assume the remaining system factors are negligible.

The effective CPI equation can then be rewritten as in Figure 2c. The first degradation term quantifies the CPI contribution due to instruction fetch cache misses, and the second term quantifies the operand reference cache misses.

The relative performance between two systems, x and y, can be expressed as:

$$\frac{x \text{ performance}}{y \text{ performance}} = \frac{y \text{ eff CPI}}{x \text{ eff CPI}} \times \frac{y \text{ cycle time}}{x \text{ cycle time}} \times \frac{y \text{ executed insts}}{x \text{ executed insts}}$$

where the first ratio defines the architectural factor, the second ratio is the technology factor, and the third ratio is the instruction set/compiler factor.

Using the system performance equation, you can analyze the relative performance of different generations of a microprocessor family, or compare different architectures. For benchmarks where the same binary code image is

executed on different designs, the relative performance equation reduces to the product of the architectural and technology factors.

MODELING TOOLS

Given CPI methodology, a number of tools have been developed to assist in this kind of performance analysis for the ColdFire architecture.

You can use a number of architectural models to analyze various factors within the effective CPI performance equation. These tools are typically high-level C language models of certain functions within the design and are driven with information from the ColdFire ISA simulator or trace data.

The ISA model is a C-language program that defines the expected results of execution of the instruction set architecture. By inputting a memory image file, the ISA model executes the program on an instruction-by-instruction basis, updating all program-visible machine registers and memory as required. This ISA model is instrumented to optionally output information on instruction fetch, operand addresses, and program counter values.

By executing the target application on the ISA simulator with the appropriate outputs enabled, a stream of data from the executing application can be input to one of the architectural models. This input data provides the required stimulus to the architectural models.

Processor pipeline models are used for base CPI analysis. There's also a program that gathers detailed statistics about dynamic opcode usage. Recalling the base CPI equation, this program provides the $F(i)$ factors associated with the various opcodes for the application.

ADDITIONAL ANALYSIS MODELS

The ColdFire cache model quantifies numerous performance parameters for various cache sizes, associativity, and organizations. It uses the stream of reference addresses generated by the simulator as input, and models the behavior of Harvard and unified caches of sizes from 512 bytes to 32 KB.

Additionally, the associativity can vary between two-way and four-way, and the operands can be mapped into copyback or store-through space. This

model can also include a RAM, mapped to a specific region, for heavily-referenced operands or code segments. Mapping the active region of the stack frame to this type of RAM is often effective.

A second model provides information for memory address profiling. Using the stream of reference addresses as input, this model profiles the memory access patterns to identify critical functions and/or heavily referenced operand locations. For some systems, such profiling helps you understand the required amount of RAM as well as which variables to map into this space to maximize performance.

Of prime importance is verification of the architectural models. So, at various times throughout the analysis process, the accuracy of the architectural models is validated.

The V.2 processor pipeline architectural model was initially verified by comparing predicted base-CPI values versus those directly measured from silicon. Reviewing measured base-CPI values versus those predicted by the pipeline model, the error was less than a 0.5% difference across a large set of embedded benchmarks.

The cache architectural models were validated against the design descriptions for several ColdFire MPU designs.

Another area of interest is the modeling of the $\{IF, OP\}_{stall}$ times. These degradation factors represent the pipeline stall that occurs on a cache miss. For the nonblocking streaming cache designs of the V.2 and V.3 cores, these terms are modeled as:

$$\{IF, OP\}_{stall} = (1 + t1) + 1.0 + 0.6 \times (t2 + t3 + t4)$$

Memory Configuration	Relative performance	Relative area
2-KB cache	1.00	1.00
+4-KB RAM	1.05	1.19
4-KB cache	1.19	1.11
+4-KB RAM	1.27	1.31
8-KB cache	1.52	1.32
+4-KB RAM	1.61	1.52
16-KB cache	1.98	1.79
+4-KB RAM	2.06	1.98
32-KB cache	2.71	2.71
+4-KB RAM	2.91	2.91

Table 1—Here's the relative performance and area for various ColdFire configurations executing a set-top box application

where the response time of the external memory for a line-sized fetch is specified as $t_1 - t_2 - t_3 - t_4$ when viewed from the microprocessor pins.

Using the equation in Figure 2d for the V.2 and V.3 designs, the relative error between the predicted and measured effective CPI

was less than 2% across a wide suite of embedded benchmarks.

Figure 3 summarizes the process. The architectural models are driven by trace data captured from existing hardware or from a compiled application executed on the instruction set simulator. The resulting streams of addresses and instructions are then input to the specific models.

The profiling tool determines any hot spots in the code or data areas that might be considered for placement

in local RAMs or ROMs. The pipeline model produces the base CPI performance metric for a given version of the ColdFire microarchitecture.

The local-memory models determine all the performance parameters associated with the cache, RAM, and ROM modules. The miss ratios are based on size, organization, and the dynamic stream of reference addresses. The base CPI and memory parameters are combined to produce an effective CPI value that provides an accurate

measure of predicted performance for a given configuration.

OPTIMIZATION EXAMPLES

To see how the performance analysis procedure works, consider the following real-world examples.

To begin, let's say you are implementing a digital set-top box. By instrumenting an exist-

ing 68k system, trace data is captured for two critical execution paths.

The challenge is to determine the appropriate amount of local processor memories (cache and possibly RAM) to optimize price and performance for a V.3 ColdFire design. When implemented in 0.35- μ m process technology, the V.3 core provides 70-Dhrystone, 2.1-MIPS performance when operating at 90 MHz.

The trace data is profiled to identify any potential hot spots that might benefit from placement in a RAM. The profile in Figure 4 shows several spikes representing heavily referenced operand areas.

The largest reference area is generally the system stack and the first candidate for mapping into a local RAM. Using the architectural models, the relative performance and area calculations across a range of cache and RAM configurations are given in Table 1. The reference design is a V.3 core with 2 KB of cache memory.

In Table 1, the relative performance ranges from 1.0 \times to 2.6 \times as a function of local memory configurations with a corresponding relative area of 1.0–2.9 \times . Depending on system requirements, the appropriate configuration can be selected, as shown in Figure 5.

In the second example, a customer provides a C-language benchmark that represented four execution paths in a servo control application. In this real-time application targeted for a V.2 core, absolute performance in response to certain interrupts was critical.

There was a fixed amount of time to service the interrupt and the algorithm implemented a number of digital

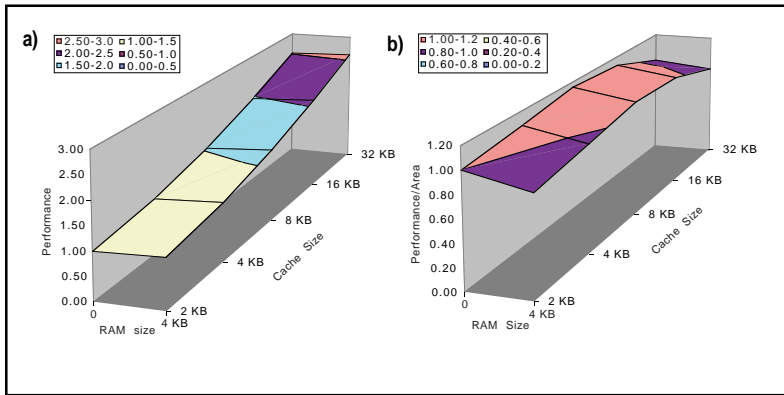


Figure 5a—This graph depicts the relative performance as a function of cache and RAM sizes. **b**—By contrast, this graph shows the relative performance per area as a function of cache and RAM sizes.

Configuration	Relative performance
CF2, no MAC	1.00x
CF2 + MAC with compiler-generated MAC instructions	1.45x
CF2 + MAC with hand-optimized MAC instructions	1.69x

Table 2—Depending on the servo control application, the relative ColdFire performance will vary.

filters. Given the signal-processing nature of the application, this analysis attempted to quantify the impact of the ColdFire multiply-accumulate unit (MAC). The optional MAC is tightly coupled to the basic execution pipeline and is designed to accelerate signal-processing algorithms.

Initial analysis indicated that the dynamic frequency of occurrence for multiply instructions (i.e., $F(\text{mul})$) was ~10%. Applying the MAC provides faster execution time for multiply instructions, reducing $ET(\text{mul})$.

Many implementations of digital filters can be optimized using MAC instructions directly. First, the bench-

mark code was compiled and executed on a V.2 core and its performance measured. This value provided the reference.

The code was recompiled using C-language macros to use MAC instructions for arithmetic calculations. The compiler-generated MAC assembly-language code was optimized by hand to provide an upper bound of performance. Table 2 shows the results.

The baseline core configuration included the processor complex with 8 KB of RAM. Including the MAC unit increased this area by only 11% but increased performance by 1.5–1.7x.

WHICH MEANS...

This analysis methodology provides a powerful tool, now system designers can balance processor performance, clock speed, and relative die size.

Given a highly configurable architecture, system designers now have access to the key silicon variables needed to create embedded processor solutions optimized for a given application. And, the result? Smart, intuitive, and user-friendly products. 📄

Joe Ciccello works as an advanced micro-processor architect for Motorola's Semiconductor Products Sector and was the chief architect for the MC68060 and the ColdFire family of microprocessors. With 23 years of experience, he is a veteran designer specializing in pipeline organization, control structures, and performance analysis. You may reach Joe at Joe_Ciccello-rzxs90@email.sps.mot.com.

Sylvia Thirtle is a principal staff engineer for Motorola's Semiconductor Products Sector, specializing in high-speed digital ASIC design. In her five years at Motorola, she's been involved in various design activities with ColdFire and is currently leading the design of the debug module for the next-generation development. You may reach Sylvia at Sylvia_Thirtle-r24495@email.sps.mot.com.

SOURCE

ColdFire

Motorola, Inc.

(512) 895-2134

Fax: (512) 895-8688

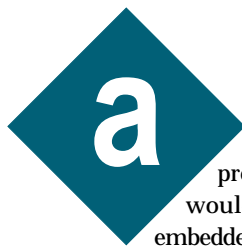
www.mot.com/ColdFire

Using Java in Embedded Systems

FEATURE ARTICLE

Vladimir Ivanovic &
Mike Mahar

If you're set on putting some desktop-Java functionality into an embedded system, chances are that you've had some sleepless nights. No more! That's what Vladimir and Mike Promise if you'll consider the available alternatives.



Although Java has properties that would be useful for embedded-system design, the versions of Java used in desktop systems just aren't suitable for embedded systems. There are some alternatives, but they do have drawbacks.

When considering the alternatives, it's important to consider issues like multithreading and debugging support. Regardless of whichever option emerges as the preferred form, two key issues must still be addressed by any embedded Java programming environment: how to provide determinism and how to interface to hardware.

JAVA IS GOOD

One of Java's strengths is its reasonably clean syntax that is strongly reminiscent of C or C++. So although it's a new language, it's familiar. Getting up to speed with Java is easy.

More importantly, Java is both object-oriented and strongly typed. Everything in Java is an object and there are no loopholes to circumvent Java's strong typing. Since the advent of C++, these features are considered essential in a programming language because they contribute enormously to the correctness of programs.

Anecdotal evidence bandied about in Java newsgroups and mailing lists

suggests that developers take less time to produce a working Java program than a program in C or C++. Debugging is also easier because Java has removed a prolific source of hard-to-find bugs, including those related to the incorrect use of pointers.

Example bugs include memory leaks and memory access errors (wild pointers, referencing freed memory, returning a pointer to a local variable, etc.). Java doesn't allow its pointer equivalent (i.e., object references) to be manipulated in the same way as pointers are in C or C++, and it provides automatic garbage collection.

Another strength of Java is its large reusable code base. In the standard distribution, Java supports threads, TCP/IP networking, and remote invocation. It even has a full set of classes for building GUI's.

Additional API's support a variety of needs, such as database access, communication, multimedia, a way to use GUI components, and security.

With Java's strengths as a language, a development environment, and a reusable code base, it's easy to see why developers—and not just embedded-system developers—are eager to put it to use.

DESKTOP JAVA DRAWBACKS

Unfortunately, as I mentioned, desktop Java has some drawbacks when used in embedded systems. Although Javawas originally intended for use in set-top boxes, it was first used in a web browser, which is a desktop application.

First, desktop Java is too big for embedded applications. Not only must the entire Java virtual machine (JVM) be present, but a Java interpreter or a just-in-time (JIT) compiler must be present as well.

On top of that, all the standard classes must be present. These take up to 8 MB on disk, more when loaded. Fonts take even more space.

The bottom line is that desktop Java needs on the order of 16 MB just to run, and the application needs are additional. Very few embedded systems have that kind of memory available.

Also, Java is too slow. Sun's first releases were usually more than 30x

slower than equivalent C code. Subsequent releases, which use JIT compilers, are significantly faster but still perhaps 5x slower than equivalent C. If you're used to squeezing out the last few cycles out of a processor, this is a heavy penalty to pay just to use Java.

But, the most important drawback of desktop Java is that it doesn't meet the constraints of most embedded systems. One such constraint is the requirement for real-time behavior (i.e., execution that's both predictable and bounded in duration).

Many embedded systems have severe real-time requirements. For instance, the collision-detection system on a jetliner has seconds in which to respond. Computation must finish in a certain amount of time, so execution has to be predictable.

Another constraint of embedded systems is their limited resources. Consumer devices, which may be manufactured in the millions, are very sensitive to cost, so designers tend to use the smallest processor and the smallest amount of memory possible to do the job. A programming language that's slow and uses up a lot of memory just isn't competitive with existing alternatives.

More importantly, Java doesn't possess the notion of an address. Embedded systems, almost by definition, are required to access hardware. Most often, that hardware is accessed by referring to a specific address. Because addresses aren't part of Java, you have to go outside the language to overcome this constraint.

Finally, desktop Java has some attributes that get in the way of successful use in embedded systems. These attributes may be useful and even necessary in desktop systems, but not in embedded systems.

For instance, Java is interpreted (the source of much of its slowness) and it is dynamic because it supports the downloading of new classes on-the-fly. Java is portable across many different systems because its source code is compiled, not to native code, but to bytecodes, an architecture-neutral format. Also, Java supports a comprehensive security model designed to prevent many kinds of attacks.

However, for embedded systems, which frequently exist in completely closed environments, portability and security aren't issues. Unless an embedded system is connected to a network, the ability to load new classes dynamically is useless.

These attributes of desktop Java prevent its use in embedded systems. And, the issues of performance, memory consumption, and poor real-time behavior make it hard to retarget the desktop version to an embedded system.

EMBEDDED ALTERNATIVES

What are the alternatives? How can an embedded-systems developer use the great features of Java without quadrupling the system's cost or writing piles of non-Java code?

Essentially, there are only three options: use a special-purpose JVM, use a JVM with a JIT compiler, or use compiled Java instead of some form of interpreted Java.

Many vendors have come up with specially tailored versions of Java that are a better fit for the needs of embedded developers. For instance, Sun offers PersonalJava for systems with 2-4 MB of memory and EmbeddedJava for smaller systems (Mentor Graphics' Microtec Division is a licensee of PersonalJava). Hewlett-Packard, NSI Com, Insignia Solutions, NewMonics, and others have similar offerings.

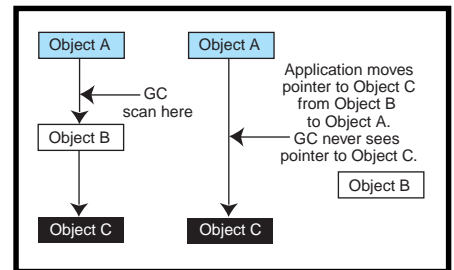


Figure 1—Object C is lost if the application changes pointers between garbage-collection scans.

Another approach, even in versions tailored for embedded use, is to use a dynamic compilation technique, typically a JIT compiler, to increase performance. But there are several tradeoffs involved.

First, JVMs with JITs have potentially longer start-up times because the JIT compiler has to compile Java bytecodes into native machine language before executing. Secondly, it's difficult to do a good job of optimizing native code while keeping memory consumption low. The more optimizations that are done, the larger and slower the JIT compiler becomes.

Several vendors provide knobs to tune the dynamic compilation process so you can choose on a case-by-case basis exactly what the performance, memory consumption, and start-up-time tradeoffs are going to be.

But, for some embedded applications, even a JVM with dynamic compilation is too slow and takes up too much memory. One option that is increasingly being considered is compiling Java directly to a native machine language, thereby eliminating both the JVM and either the interpreter or the JIT compiler.

Of course, the resulting application is no longer portable, but embedded developers typically don't care about portability. For a given design, their application needs to run on a single well-known hardware configuration.

The other attribute that compiled Java forces a developer to give up is the ability to load new classes on-the-fly. Because all the code is precompiled, there's no facility for dynamic loading of classes. Again, this issue probably isn't too serious for embedded-system developers, most of whom don't want random classes downloaded onto their system.

Listing 1—Since a compiler knows about every write to memory, it inserts a write barrier automatically.

```
object_a->next = object_c;
if(object_c != NULL)
    if(object_c->garbage_flags == WHITE){
        object_c->garbage_flags = GRAY;
        gc_make_gray(object_c);
    }
```

If you're willing to tolerate the lack of portability and the lack of dynamic class loading, you can still reap all the benefits of Java as a great language and keep the system small and fast—that is, if you can resolve the issues of determinism and low-level programming.

RESOLVING THE ISSUES

Any version of Java for embedded systems must first be deterministic and predictable. It also has to be able to access memory directly.

One bugaboo of embedded systems is ensuring real-time response. In the case of Java-based systems, the primary cause of nondeterminism is the garbage collector.

In desktop systems, it doesn't matter much that the JVM stops for several seconds to collect unused memory. But in an embedded system, several seconds can be the difference between correct operation and the loss of human life.

The biggest threat to an embedded operation is that most garbage collectors work in what's called stop-the-world mode. Usually, the collector is called only when an allocation fails because memory is exhausted. Therefore, allocation time is impossible to predict, and when the collector is running, no other processing is being done. This situation is unacceptable in a real-time system.

An obvious solution is to have the garbage collector run concurrently with the application so that the impact of garbage collection is spread around more evenly. This way, time-critical events are processed in a timely manner.

Ensuring real-time response still isn't enough to make Java useful for developing embedded systems. Because the added value of embedded systems is their specialized hardware, the embedded software must always be able to access or control the hardware, which requires an extension to the Java through a Java Native Interface (JNI) with several possible options or through a nonstandard extension of the Java language.

Sun's JNI permits portability across different JVMs on a particular processor

Listing 2— This code gives you an example of how the *Phys* package can be used.

```
import COM.mentorg.microtec.phys.*;
class m68561 {
    PhysByte this_uart;
    PhysByte Tsr, Tdr, Rsr, Rdr;
    int baseAddr;
    m68561(int base_addr) {
        baseAddr = base_addr;
        this_uart = new PhysByte(baseAddr);
        Tsr = new PhysByte(baseAddr + 0x8 * 4);
        Tdr = new PhysByte(baseAddr + 0xa * 4);
        Rsr = new PhysByte(baseAddr + 0x0 * 4);
        Rsd = new PhysByte(baseAddr + 0x2 * 4);
        // Initialize UART. Reuse this_uart object because registers
        // are only used once or twice
        this_uart.setAddress(baseAddr + (0x1 * 4)); //RCR
        this_uart.set(1); // Reset receiver
        this_uart.setAddress(baseAddr + (0x9 * 4)); //TCR
        this_uart.set(1); // Reset transmitter
        this_uart.setAddress(baseAddr + (0x19 * 4)); //PSR2
        this_uart.set(0x1e); // 1 stop, 8 bit
        this_uart.setAddress(baseAddr + (0x1c * 4)); //BRDR1
        this_uart.set(0x8c); // 9600 bps
        this_uart.setAddress(baseAddr + (0x1d * 4)); //BRDR2
        this_uart.set(0); //
        this_uart.setAddress(baseAddr + (0x1e * 4)); //CLKCR
        this_uart.set(0x1c); // Divide by 3, TCS out, TXC in
        this_uart.setAddress(baseAddr + (0x1f * 4)); //ECR
        this_uart.set(0); // No parity, no error check
        this_uart.setAddress(baseAddr + (0xd * 4)); //TIER
        this_uart.set(0); // No transmitter interrupt
        this_uart.setAddress(baseAddr + (0x15 * 4)); //SIER
        this_uart.set(0); // No serial interrupt
        this_uart.setAddress(baseAddr + (0x05 * 4)); //RIER
        this_uart.set(0x1e); // No receiver interrupt
        this_uart.setAddress(baseAddr + (0x1 * 4)); //RCR
        this_uart.set(0); // Enable receiver
        this_uart.setAddress(baseAddr + (0x9 * 4)); //TCR
        this_uart.set(0x80); // Enable transmitter
        this_uart.setAddress(baseAddr + (0x11 * 4)); //SICR
        this_uart.set(0x80); // RTS, DTR low
    }

    int pollReceive() {
        if (!Rsr.andByte(0x80)) {
            return Rsd.getByte(); //Check for break or error
        }
        return (-1) // -1 means no character
    }

    int receive() {
        // Wait for character to be called in a separate thread.
        while(Rsr.andByte(0x80)) {
            return( Rsd.getByte());
        }
    }

    void send(byte character)
    {
        // Wait for transmitter to be ready. Should probably be
        // in a separate thread.
        while Tsr.andByte(0x80)) {
            ;
        }
        Tsd.setByte(character);
    }
}
```

architecture, but it suffers from poor efficiency. An earlier version of JNI was more efficient, but it required the Java code to know the layout of an object in memory. In any case, it makes sense for an embedded system to offer a package specifically for accessing physical memory.

REAL-TIME GARBAGE COLLECTION

The Java language requires garbage collection of unused objects and there's no corresponding `delete` operator to go with the `new` operator. One advantage of garbage collection is that you can't have bugs in your memory allocation if the responsibility for detecting unused memory and reallocating it is automatic. The application simply clears a reference to memory to make it available for future use.

However, the advantages of garbage collection come with a price. Finding unused memory and freeing it can take a long time, causing critical deadlines to be missed in a real-time environment. A garbage collector for a real-time system must be predictable

and fast in addition to allowing high-priority threads to run. Unfortunately, most collectors fail on all three of these requirements.

Garbage collectors work oppositely of what their name implies. They find all the memory blocks that are in use and free up what's left over. There are many different algorithms for garbage collection, but most of them share the following steps:

- scan the local and statically allocated variables for pointers to the heap
- mark each memory object that can be reached from these pointers
- scan each marked memory object for pointers to the heap
- repeat steps 2 and 3 until no new pointers are found
- sweep the heap and free up any memory that is not marked

MOVING POINTER PROBLEM

A major problem with concurrent garbage collection is that while the collector is scanning the heap for pointers, the application is changing

those same pointers. In essence, the entire heap is a critical section.

Suppose an application is manipulating a linked list of three objects, as illustrated in Figure 1. Object A points to object B, which points to object C. The garbage collector scans object A for pointers, but has yet to scan objects B or C. The application then deletes object B by copying the pointer to C to object A.

The collector hasn't scanned object C, nor has it scanned object B, so it won't find any pointers to object C because it already scanned object A. When the collector completes scanning, it frees object C even though there's a live pointer to it in object A.

APPLICATION AND INTERFERENCE

Interference between the application and garbage collection is the only situation you have to worry about. Other accesses to the heap don't affect garbage collection. To make concurrent garbage collection work, the application must tell the collector every time it writes an object pointer to another

memory object. This is called a write barrier.

Write barriers sound expensive, but there are several ways to speed things up. Every allocated object on the heap has a flag word containing the state of that memory object. There are three possible states: black, gray, or white.

The collector knows a black-state object is live and has scanned it for pointers. The collector knows a gray-state object is live but has not yet scanned it for pointers. In the white state, the collector has not yet found a pointer to the object.

There are several variations on how a write barrier is implemented. Listing 1 is one example of a write barrier. The `if` statement is generated automatically by the compiler, so it's not necessary to put in write barriers by hand.

Every pointer assignment to the heap has two additional tests. Usually, programs manipulate the same pointers multiple times. The `garbage_flags` and `gc_make_gray` function calls occur only the first time an object is seen.

Subsequent stores find the object already marked `GRAY` and don't have to call the collector. Making a `WHITE` object `GRAY` isn't an expensive process, often taking less than ten instructions.

Once the garbage collector and the application are cooperating, it's possible to run the garbage collector as a separate thread of execution. The garbage collector's priority may have to be set differently depending on the characteristics of the application. The priority can be set low if the application spends a lot of time waiting for external events.

In fact, applications that are I/O bound or event driven may have better performance than explicitly freeing objects because the garbage collector can run while the processor is not otherwise busy, and still keep up with the demands of new memory. However, that's not always the case.

If an application has a mix of event, I/O, and compute-bound processing threads, the priority of the garbage collector can be set lower than the event and I/O threads and at the same priority as the compute-bound threads.

And, when all free memory is exhausted, the garbage collector's priority can change dynamically to take the priority of the thread that was trying to allocate memory.

The garbage collector can run until completion, and then the allocating thread can continue. It's also reasonable to have just two priorities for the garbage collector: one that is low for when free memory is plentiful and one that is higher for when free memory is exhausted, or nearly exhausted.

MEMORY ALLOCATION

Garbage collection isn't the only memory-management consideration in a real-time system. The allocation of memory must be fast and predictable. The system must allocate an object in nearly the same amount of time for every allocation.

Therefore, the memory manager can't maintain long linked lists of objects that must be searched every time memory is requested. The memory heap structure must be organized in a way that ensures predictability.

INTERFACING HARDWARE

The Java language doesn't have pointers to physical memory or any other built-in method for accessing specific memory addresses. This limitation makes it difficult to write device drivers or any other code that needs to talk to physical devices entirely in Java. Additionally, there may be existing modules written in C++ that you want to keep.

For this reason, the Java language specifies that certain methods may be declared native. Originally, native methods enabled high-performance functions to be performed in the native instruction set of the host machine without incurring the performance of the virtual machine.

Native methods can interface between Java and C++ in a compiled environment as well. The native declaration tells the compiler that the method is externally defined, so you can write this external method in C or C++.

Besides allowing native methods to be written in C or C++, the Java compiler has a switch (-xj for the Microtec Java compiler) that tells the compiler to emit two files for every class that has native methods. The first file is a C++ header file called Class.h that contains a C++ definition of the Java class. The second file is a Class.cpp file that contains a stub C++ program for every native method.

To implement the native methods, just edit the .cpp file and add code to the method stubs (see Photo 1).

PHYSICAL MEMORY PACKAGE

Because accessing memory directly is such a common request, Microtec included a package of classes called COM.mentorg.microtec.Phys, which enables you to create objects that access memory directly. There are three classes, one for each size of memory.

These classes are PhysicalByte, PhysicalShort, and PhysicalInt, and each class contains the following methods:

- Physicalsize(int address)
- set(size value)
- size get()
- int getAddress()
- setaddress()
- size and(size value)
- size or (size value)

The constructor for each of these objects takes an int argument that's the address in memory associated with this object. When a variable of type PhysicalInt, PhysicalByte, or PhysicalShort is declared, it takes one argument, which is the memory address at which you want the data to reside. For example, PhysicalInt myInt = new PhysicalInt(0x0100000C); creates a PhysicalInt variable and stores it at memory location 0x0100000C.

To set the data to something useful, use the set() method. For example, myInt.set(256); gives your myInt a value of 256.

Later, when you need to retrieve the data in myInt, use the get() method. int newInt = myInt.get(); will

make newInt contain the value (e.g., 256) of the data in myInt. If you need to find out the address of a variable in memory, try something like, int myAddress = myInt.getAddress();

You can change the address of myInt in memory with the setAddress() method. For instance, myInt.setAddress(0x0100000A); changes the address from whatever it was before to 0x0100000A.

If you want to perform a bitwise and or or with the data, they work the same way:

```
newInt = myInt.or(31);
newInt = myInt.and(31);
```

The functions take an argument, the number to and or or, to the data already in place and return the result.

These classes can be subclassed and any of the methods may be overridden to add additional functionality. For example, the and or or methods may need to be noninterruptable so they can be overridden with a method that disables interrupts during their execution. Listing 2 shows how to use COM.mentorg.microtec.Phys.

RECOMMENDATIONS

You've seen some of Java's advantages as a language for developing embedded systems, but you've also seen some of the nonobvious pitfalls of using a version of desktop Java in an embedded system.

Of the three options (special purpose JVM, JVM with a JIT, or compiled Java), compiled Java probably has the best set of tradeoffs. It's clearly the winner in raw CPU speed, and it has memory usage similar to a conventional language like C or C++. Although compiled Java isn't portable and doesn't immediately offer the ability to load classes dynamically, for many embedded systems, these drawbacks aren't issues.

If portability or dynamic class loading are needed, the next-best alternative is probably a special purpose JVM.



Photo 1—This screenshot demonstrates implementing the native method by editing the .cpp file.

Although a JIT compiler seems to offer the most straightforward way to add performance to Java, it runs the risk of taking up too much memory and causing unacceptable pauses while compiling a just-loaded class.

These alternatives resolve the problem of using Java in embedded systems. So, all the benefits of Java as a language and as a source for reusable code are available to both embedded and desktop application developers. ☒

Vladimir Ivanovic is the senior marketing engineer at Mentor Graphics' Microtec Division and specialize in Microtec's VRTX real-time operating and developing systems and Java products. He has taught computer courses for more than 12 years at Northeastern University and at the University of California, Berkeley, extension. You may reach him at vladimir_ivanovic@mentorg.com.

Mike Mahar has worked in the software development industry for more than 20 years and has been involved in

developing compilers, assemblers, debuggers, and integrated development environments. He has also worked on software development tools and contributed to the design of several RISC processors. You may reach him at mike_mahar@mentorg.com.

REFERENCES

- K. Arnold and J. Gosling, *The Java Programming Language*, Addison-Wesley, Reading, MA, 1997.
- R. Jones and R. Lins, *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*, Wiley & Sons, New York, NY, 1996.

SOURCES

Mentor Graphics Microtec Division
(800) 950 5554
(408) 487-7000
Fax: (408) 487-7001
www.mentorg.com/microtec/java

Hewlett-Packard
(800) 452-4844

(650) 857-1501
www.hpconnect.com/embeddedvm

Insignia Solutions
(800)848-7677
(510) 360-3700
Fax: (510) 360-3701
www.insignia.com/embedded

NewMonics
(515) 296-0897
Fax: (515) 296-4595
www.newmonics.com

NSI Com
(212) 717-9615
Fax: (212) 734-4079
www.nsicom.com

EmbeddedJava, Java Native Interface, PersonalJava

Sun Microsystems
(650) 960-1300
www.java.sun.com/products/embeddedjava
www.java.sun.com/products/jdk/1.1/docs/guide/jni
www.java.sun.com/products/personaljava

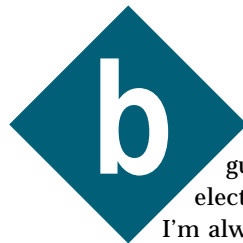
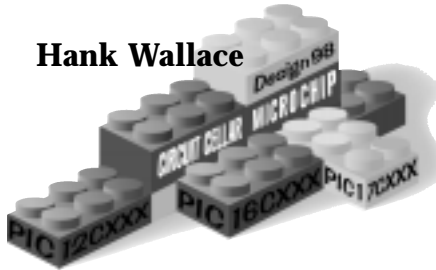
Music at Your Fingertips

Guitar Effects via Remote Control

Hank loves to play guitar, and like many guitarists, he thinks that using pedals to control the effects is too restrictive. What's a stage-raging musician to do? As an embedded-systems designer, Hank opted to put the music in the hands of the artist.

FEATURE ARTICLE

Hank Wallace



Being a longtime guitar player and electronics addict, I'm always searching for the latest technology to enhance my hobby. And, one of the most frustrating tasks for a guitar player is controlling sound while playing live.

Sound is typically controlled with an array of foot switches on the stage. However, this arrangement limits the musician's movement, and a stationary musician is certainly not conducive to the general affectations and mannerisms of modern music. Concentrating on foot switches also robs brain cycles from the activity at hand and detracts from the show.

That's why I began searching for an effective, inexpensive way to control my guitar processors and amplifiers without foot switches. After several imaginative attempts that were too costly or impractical, I finally found a simple, low-cost method for remotely controlling equipment configurations.

Figure 1 shows an overview of my guitar effects controller system. The transmitter interprets the guitar player's touch on the three isolated metal surfaces on the instrument and translates the information into ultrasonic data bursts that are sent down

the guitar cable to the receiver with the usual guitar audio.

The low cost, low power consumption, and rich features of today's microprocessors make this system possible. I chose to base the two parts (transmitter and receiver) of my guitar effects controller system on the PIC12C508. Figure 2 shows the layout of both parts.

TRANSMITTER

As you see in Figure 1, a transmitter is mounted under the guitar's phone jack to sense the guitarist's touches on the three metal pads attached to the pick guard. Touchpads provide simple, inexpensive operation.

The pads are aluminum tape and use the resistance of the guitarist's body between the pad and ground to trigger a transmission. The strings are grounded, and the fretting hand is almost always touching the strings, providing a ground path.

I didn't use switches because it's next to impossible to mount them on a guitar without drilling holes—a taboo to many guitarists. The touchpad design requires no permanent changes to the instrument that might affect its future collector's value.

The data bursts are sent to the receiver at an inaudible 50 kHz. Any frequency over 15 kHz may be used, but 50 kHz lowers costs by allowing simple two-pole filtering to be used. Lower frequencies would enable you to use existing wireless systems that typically cut off at 15 kHz.

In Figure 3, the transmitter schematic shows the general-purpose NPN transistors (Q1-Q3) that buffer the three touch pads (J1, J4, and J5). Filtering on the base of each transistor (C1, C3, C5) removes any ambient noise, including 60 Hz.

Transistor collectors are connected to the three inputs of the PIC12C508 to awaken the processor from sleep mode on pin change GP0, GP1, and GP3. The internal weak pullups on these pins are enabled.

The reference for the touchpads is not ground but the +3-V supply of the transmitter. The ground of the transmitter is actually the microprocessor's V_{CC} . This arrangement works fine because the 50-kHz output of the

transmitter is AC-coupled to the guitar audio path.

Because the +3-V V_{CC} signal is the only DC connection between the transmitter and the instrument, internal pullups can be used on GP0, GP1, and GP3.

With one hand on the strings and a finger on the touchpads, the guitarist's body conducts enough current to turn on a transistor and wake up the PIC12C508. The processor does a 120-ms debounce delay, qualifies the three inputs, and produces a 50-kHz pulse train on output GP2.

The response varies according to the combination of pads activated. If the pads are A, B, and C, the seven possible combinations are A, B, C, A+B, A+C, B+C, and A+B+C.

Imagine a guitar player accidentally touching one of the pads during a screaming lead break and immediately switching sounds from Van Halen shred to George Benson mellow. Embarrassing to say the least, especially in front of fans who think nothing of piercing their own, not to mention someone else's, flesh. The ramifications on church musicians like me can be just as ominous.

To prevent this, the A+B+C combination locks the transmitter output against inadvertent changes. Touching the three pads together toggles an internal lockout flag preventing any transmissions until the A+B+C combination is touched again.

The datastream consists of 3-ms, 50-kHz bursts, with 1 ms between each burst. To enhance noise rejection at the receiver, two bursts is the minimum transmitted. Thus, the transmitter can send between two and seven bursts with a maximum total length of 27 ms.

Figure 3 shows that the data is capacitively coupled to the audio path through C4. My Fender Stratocaster has an impedance through the pickups of 2.5 k Ω to ground.

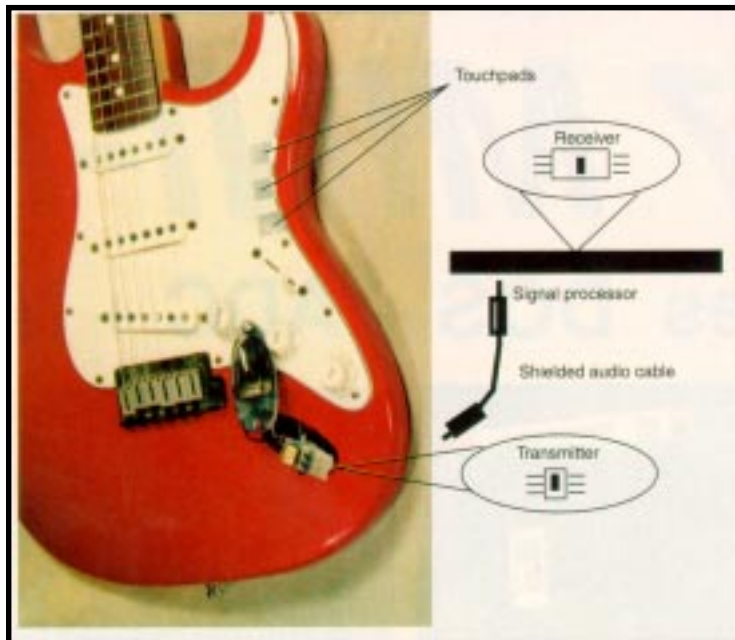


Figure 1—The transmitter is installed under a jack plate of the guitar. The few wires are either connected to the audio jack or routed through a preexisting hole in the body toward the touchpads on the pick guard. The pads are aluminum tape wrapped under the pick guard where the connection is made. No drilling or other permanent modifications are made to the instrument.

The 100-k Ω series resistor (R1) lowers the signal level to the minimum required, and also protects the microprocessor output from electrostatic damage. The 1-k Ω series resistor (R2) raises the output impedance of the amplifier so the data is not swallowed even if active electronics exist in the modified guitar.

The 50-kHz data burst has a DC component that is half V_{CC} for the duration of the square wave, or 1.5 V. As a result of the DC shift across the coupling capacitor (C4), each data burst causes pops and clicks when transmitted. Also, ringing is caused by the inductive effect on the DC pulse by the guitar's coil pickups.

To solve this problem, output GP4 switches from V_{CC} to ground during a burst while GP2 switches from ground to a square wave with a half- V_{CC} average level. The RC summing network on GP2 and GP4 subtracts the DC-induced transient from the data waveform.

This switch gives the impression that the transient was generated from a bipolar swinging source and thereby eliminates the pops and clicks. The DC resistance through the paths has a ratio of two for the 3/1.5-V ratio of the DC pulse levels.

To cancel the RC transients, the time constants of each path remain identical.

TAKE NOTE

The data output of the transmitter must be terminated in a low impedance like a guitar pickup. Otherwise, the capacitively coupled negative-going transients can damage the micro's outputs even if a pin is set to output mode.

To lower the parts count, I used the internal RC oscillator mode of the PIC12C508. Therefore, a communications protocol that tolerates the resulting wide timing variation over temperature and supply voltage had to be used.

The receiver only counts bursts with an overall gross timeout parameter, allowing heavy noise-elimination filtering can be done in the receive program.

Until activated by the user's touch, the transmitter micro remains asleep. The entire circuit runs off a dime-sized lithium battery, and average power consumption, including the sleep spec of the micro and leakage of the transistors, amounts to a few microamps. The estimated battery life is two years.

Because guitar manufacturers leave precious little room for additions inside their instruments, size is critical. With this in mind, the surface-mount board is single sided and the outline is smaller than a postage stamp.

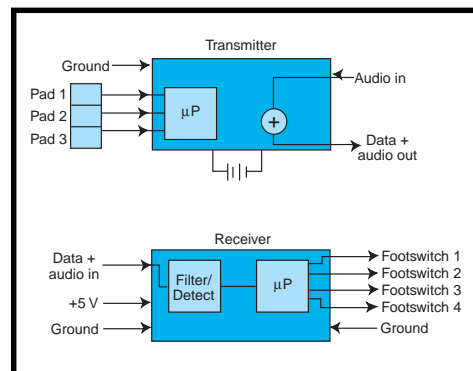


Figure 2—These block diagrams of the transmitter and receiver illustrate their simplicity.

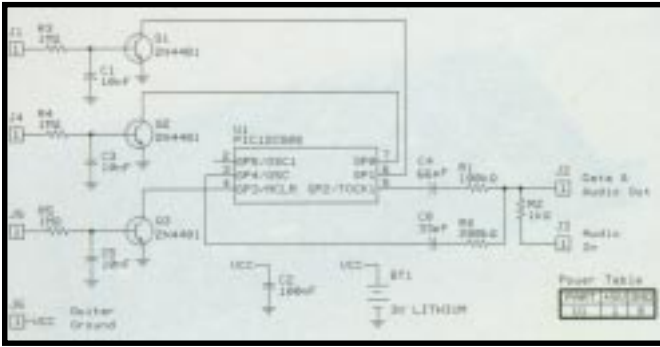


Figure 3— The schematic of the transmitter shows that it could hardly be simpler. Running from a 3-V lithium battery, the micro is in powersave mode until awakened by a touch on one of the pads. This circuit can be rendered in printed circuit format about the size of a postage stamp.

RECEIVER

The receiver is mounted inside a commercial DSP-based guitar effects processor. After decoding the data burst, the receiver determines which touchpad(s) were activated and connects the processor's foot switch terminal(s) to ground for 250-ms simulating foot-switch operation.

Also based on a PIC12C508, the receiver has an LM358 op-amp high-pass filter front end to separate the guitar audio from the data burst (see Figure 4). Two high-pass sections with a gains of over 200 bring the data burst up out of the dirt. The LM358's gain rolls off at 100 kHz—another benefit of this design.

Guitar cable capacitance and low pickup impedance help attenuate the data. The 50-kHz data is sent to the guitar processor for removal by its anti-aliasing filter, so it's virtually inaudible.

Once amplified, the data burst is applied to an amplifying rectifier that

consists of a PNP transistor (Q1). The following NPN transistor (Q2) discharges the filter capacitor (C2) when a burst is present, and that capacitor's signal drives the micro on GP2.

The program performs additional low-pass noise filtering (similar to switch debounce), counts the received pulses, and activates one or more of the output transistors for 250 ms as needed. The output transistors connect to the foot-switch jack in the guitar processor. Thus the receiver outputs and foot switches (if any are used) are wire-ORed, as seen in Photo1.

Powered by the processor's +5-V supply, the receiver doesn't need any special power-management techniques.

ENCORE

The receiver may be enhanced to include MIDI support in place of the four foot-switch drivers. Then, it can control any MIDI-capable instrument, like a synthesizer or sequencer.

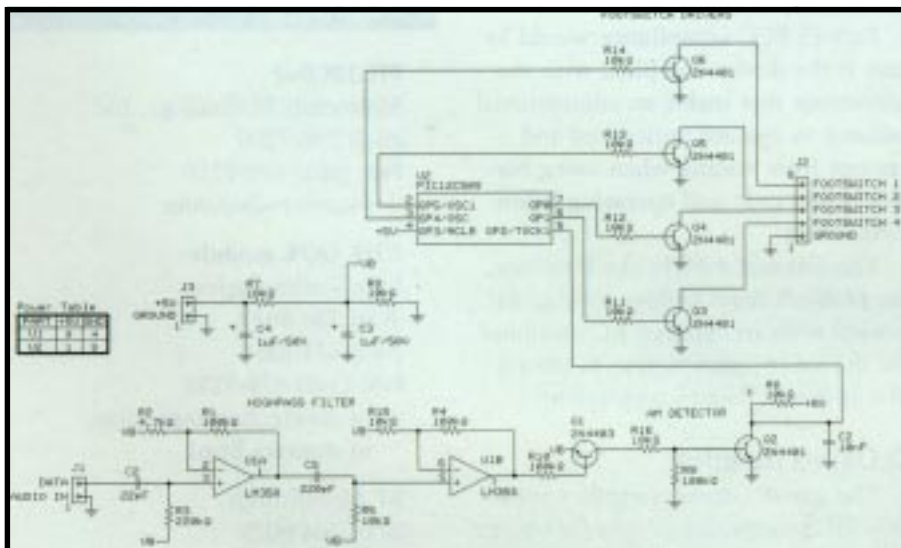
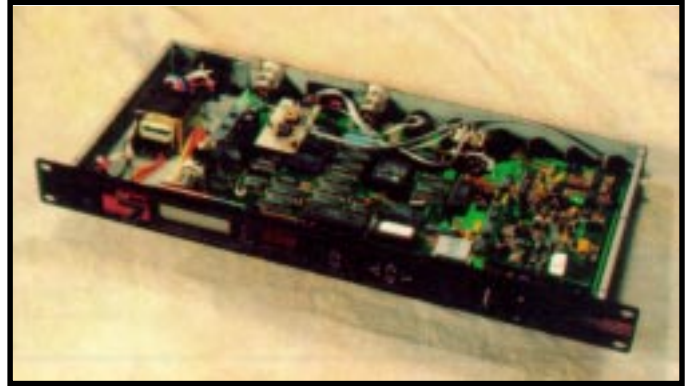


Figure 4— The schematic of the receiver shows the analog filtering, AM detection of the 50-kHz pulses, and processing by the micro. The signal processor'd footswitch inputs are activated by four open-collector drivers.

Photo 1— The receiver is installed in a commercial guitar-effects processor, stealing +5 V from the internal power supply.



By packaging the receiver in an external foot-switch controller, the device can be made compatible with lots of equipment. However, including the receiver in the signal processor permits the welcome deletion of the foot-switch box from my stage setup.

The transmitter and receiver software could also be enhanced to perform MIDI continuous-controller functions. Simply holding a finger on one of the touchpads would control the level of effects. Using a processor with more wakeup-on-pin-change inputs (e.g., the '16F84) provides more possibilities.

Of course, communication from transmitter to receiver can be performed over an RF link using some of the UHF OOK modules. The functions would be available without touching the audio signal path—a must for demanding guitarists and anyone desiring this benefit while recording. Changing the databurst frequency to 15 kHz, still practically inaudible, enables use of a commercial wireless unit.

Part-15 FCC compliance would be easy if the device complied with the restrictions that enable an unintentional radiator to operate unlicensed and exempt from testing when using battery power only and operating below 1.706 MHz.

The internal 4-MHz clock causes the problem here. Unless another PIC is used with an external RC oscillator, the device requires testing to ensure that it meets Part-15 regulations.

CLOSING NUMBER

The guitar effects controller shows how PIC micros can effectively replace digital and analog alternatives. As the replacement for a circuit based on the CD4093 quad NAND Schmitt trigger,

the PIC12C508 costs about the same but requires less space and provides seven functions instead of one.

The estimated parts cost is about \$3.50 for the transmitter and \$5 for the receiver. I have to wonder what the world's coming to when I can buy a 20-MHz microprocessor for the price of a cheap op-amp. ☒

Hank Wallace is the president of Atlantic Quality Design. When not annoying family and neighbors with extremely loud tests of new guitar gadgets, he designs embedded-systems hardware and software for more traditional clients. You may reach him at www.aqdi.com.

SOFTWARE

Software for both the transmitter and the receiver is available via the Circuit Cellar web site.

SOURCE

PIC12C508

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

UHF OOK module

Linx Technologies
(800) 736-6677
(541) 471-6256
Fax: (541) 471-5251
www.linxtechnologies.com/m_contact.html

RF Monolithics

(800) 704-6079
(972) 448-3700
Fax: (872) 387-8148
www.rfm.com

I/O SUBSYSTEM BOARD

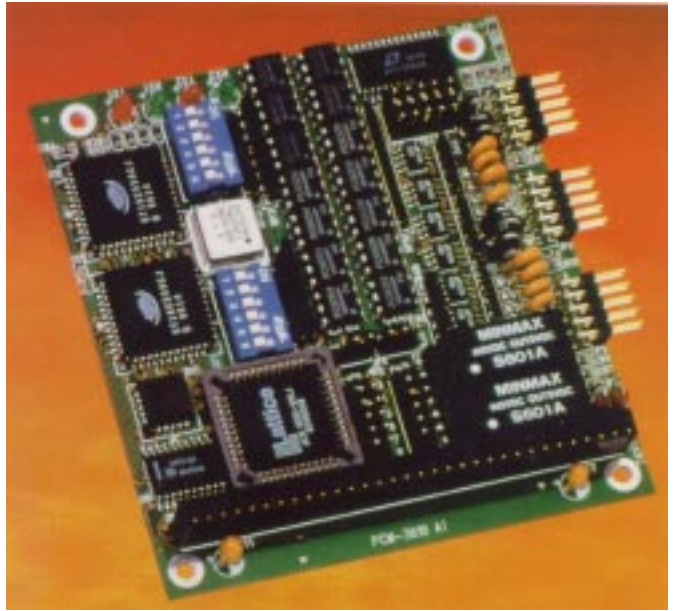
The **PCI-ADADIO** I/O subsystem board is a PCI-bus expansion board that provides analog inputs (ADC), analog outputs (DAC), digital I/O, and timer/counters. In short, the APCI-ADADIO includes everything required to interface with most sensors and transducers, monitor switches and provide digital outputs, and perform onboard timing for purposes such as controlling scanning or generating PC interrupts.

The analog input function features a 12-bit ADC, available as 16 single-ended or eight different channels, link selectable for two unipolar or two bipolar voltage ranges. The board handles input frequencies up to 100 kHz single channel or 10 kHz channel-to-channel. Two precision analog output channels are provided by a 12-bit DAC, with a typical settling time of 10 μ s. These channels are link selectable for one unipolar or two bipolar ranges.

The board includes 16 bidirectional TTL-level digital I/O lines, organized a four nibbles, a 1-MHz clock, and a programmable 8254-compatible device offering three 16-bit timer/counters. One channel is dedicated to triggering periodic conversion of the ADC. The other two are for general-purpose use, such as generating periodic interrupts to control channel scanning. A software strobe or external triggering can also activate the ADC.

The board comes complete with example C source code to help calibrate the board. Also supplied are Windows NT 4.0 drivers for use with high-performance 32-bit OSs.

Arcom Control Systems
(816) 941-7025
Fax: (816) 941-0343
www.arcomcontrol.com



SERIAL I/O MODULE

The **PCM-3610** is a dual-port serial interface module in a PC/104 form factor. It features two independent fully isolated serial channels with RS-422/-485 interface. Channel two also includes an RS-232 interface.

Electrical isolation of up to 500 VDC from both the system power supply and the serial interface signals provides important protection for the system from ground loops, externally induced power surges, and other hazards present in industrial environments. Additionally, line-side surge circuitry provides protection for other devices in the RS-485 network.

Using the industry-standard 16C550 asynchronous communications chip, this module is fully DOS and Windows compatible when used at the standard COM1 and COM2 addresses. The module is capable of data rates up to 115 kbps and transmission distances of up to 4000' when the RS-422/-485 interface is used.

The module also has RS-485 interface circuitry with automatic direction control. This eliminates the need to modify software drivers to manage the switching between send and receive modes.

The PCM-3610 sells for **\$186** in OEM quantities.

Versalogic Corp.
(800) 824-3163
(541) 485-8575
Fax: (541) 485-5712
www.versalogic.com



DSP WITH ON-CHIP FLASH MEMORY

The **TMS320F240**, the first DSP with on-chip flash memory, is ideal for motor-, motion-, and process-control applications. Motor designers can program quickly to flash memory and then transfer that code to a more cost-effective ROM-based DSP for volume production.

The DSP is pin- and code-compatible with TI's 'C240 ROM-based DSP of the TMS-320C24x generation, and it includes a 10-bit ADC with an 850-ns typical conversion time and a dedicated event manager that supports multiple PWM channels and dead-band logic.

Two other flash-based DSPs—the **TMS320F241** and the **TMS320F243**—integrate both flash memory and the industry's only control area networking (CAN) bus. With the CAN interface, these two devices



will support complex industrial applications that require control over multiple motors and intersystem communication. The TMS320F243 also integrates an expansion bus for additional memory.

The TMS320F240DSP is available in a 132-pin plastic quad flatpack (PQFP) and is priced at **\$15.51** each in 10k units. The 'F241 is packaged in a 68-pin plastic leaded chip carrier (PLCC) and a 64-pin PQFP and is priced at **\$12.73** each in 10k units. The 'F243 is packaged in a 144-pin thin quad flatpack (TQFP), and it costs **\$14.39** each in 10k units.

Texas Instruments
(800) 477-8924, x4500
(972) 995-2011
Fax: (972) 995-4360
www.ti.com/sc/docs/dsps/dcs/dcshome.htm

FLAT-PANEL CONTROLLER BOARD

Apollo's CDS545 flat-panel controller board is an ISA-bus direct interface that supports flat-panel resolutions up to 1280 × 1024 and noninterlaced CRT monitors with resolutions up to 1024 × 768. Hardware Windows acceleration includes a 32-bit graphics engine, three operand ROPs, color expansion, hardware line drawing, and hardware cursor. True-color and Hi-color display capabilities support resolutions to 640 × 480. Colors are converted up to 64 shades of gray for monochrome panels.

The display voltages generated onboard include power-up sequencing, panel V_{cc} , LCD bias voltage, and 12-V backlight voltage. The board includes 1 MB of display memory and can support 5- and 3.3-V panels. The board also runs EL and plasma flat panels.

Applications include user interfaces for commercial, industrial, and scientific products. Sample price is **\$240**.

Apollo Display Technologies, Inc.
(516) 654-1143
Fax: (516) 654-1496
www.apollodisplays.com



Nouveau PC

PC/104 CPU BOARD

The **GW2400** is a PC/104 '486 CPU board based on the AMD Elan SC410 embedded '486 CPU. The board features up to 100-MHz operation, 16 MB of DRAM, and 8 MB of flash memory.

It comes with DOS in flash memory and includes a full set of PC peripherals such as IDE, floppy, three serial ports, parallel port, and AT keyboard controller. It also contains several features for industrial applications such as serial port console redirect, watchdog timer, user-programmable LED, and battery-backed real-time clock.

The GW2400 leverages off the high integration of the AMD Elan SC410 to bring a full featured, low-cost '486 board to the PC/104 bus. The board was designed to handle rugged environments and features soldered-in memory.

The 33-MHz GW2400 with 2-MB DRAM and 512-KB flash memory is priced at \$395 in OEM quantities.



Gateworks Corp.

(805) 461-4000 • Fax: (805) 461-4001
www.gateworks.com

PCI WATCHDOG TIMER CARD

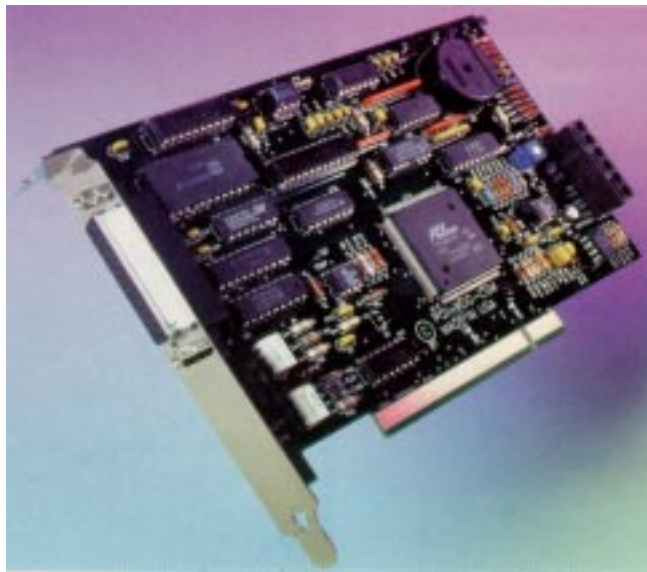
The **PCI-WDGCSM** is a PCI watchdog timer card that continuously monitors critical PC functions. When a fault occurs, the card automatically generates outputs that initiate corrective action or generate alarms. For Windows 95/98 users, the card can supply an interrupt prior to watchdog timeout (programmable from 4 μ s to 30 min.) that will cause a user-provided interrupt service routine to perform a graceful shutdown of Windows so that application files do not risk corruption during restart.

The PCI-WDGCSM offers a building-block approach to board-level functionality. Six low-cost options (\$10-20 each) enable users to specify a multi-function card that is tailored to their particular status-monitoring needs. This feature keeps the price of the PCI watchdog timer card low by eliminating unneeded functionality. The six options include a power monitor, temperature monitor, computer temp-

erature measurement, four functions (e.g., change of state, timeout buzzer, optoisolated outputs), one or two computer-controlled digital outputs in lieu of optoisolated outputs, and fan-speed control.

Each PCI watchdog timer card comes with two 3.5" floppy diskettes. The Product Diskette provides a setup program, DOS drivers, and sample programs in Pascal, C, and QuickBASIC. The second floppy includes numerous drivers and utilities, including PCI/ISA, Visual Basic, Windows NT, and other drivers and utilities.

The PCI-WDGCSM sells for **\$175-260**, depending on which status-monitoring options are selected.



ACCES I/O Products, Inc.
 (619) 550-9559
 Fax: (619) 550-7322
www.acces-usa.com

Nouveau PC

Real-Time PC

Ingo Cyliax

Embedded RT-Linux

Part 3: Networking

Why is Linux so popular? It could be the networking. No, not the schmoozing nice-to-meet-ya kind of networking! Ingo's talking about the down-and-dirty details of protocol stacks, link-level devices, network file systems...

Embedding Linux into a fairly small system using a floppy or RAM disk-based file system is a method that opens many possibilities for basing embedded systems on Linux. But, Linux's popularity could also be a result of its extensive networking support.

Linux includes a little of everything, and it's all free if you download it from the Internet. Of course, most of us just buy a distribution CD, which typically comes royalty free and with sources. Either way, it's a nice change from commercial OSs.

Although Linux is used in many embedded projects, probably the most popular embedded application is the terminal server or Internet router.

A terminal server is an Internet router that accepts modem connections (usually 64-128 ports) from PCs and routes packets (using the mode line) over Ethernet or WAN connections to the Internet.

An Internet router is a generic version of the terminal server but with one or more WAN connections to the Internet and one

or more Ethernet connections. Based on a routing table, it routes IP packets from one interface to another.

You might not think of these devices as embedded systems, but most terminal servers and routers don't have a console

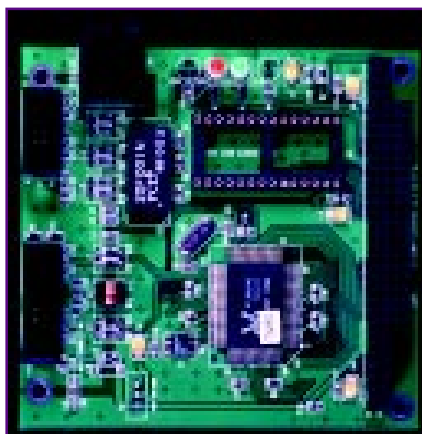


Photo 1—The PC/104 card from VersaLogic basically just plugs in and works with Linux. I had to change the IRQ using the supplied configuration program because it conflicted with the IRQ of the serial port.

and they live in a rack or wiring closet. Some are even managed remotely from across the country using web-based interfaces. Terminal servers and routers tend to be very transparent to users (i.e., as far as the user is concerned, things just work).

Because Linux makes possible other embedded projects that require network support, there's no reason why it can't be used here. So, let's look at what Linux has to offer with respect to networking.

There's so much to write about that I won't have enough time to cover programming under the socket API (the basic programming interface for Linux and other Unix and Unix-like OSs). But an overview can be found my column in *INK 98*, which covers the basics of TCP/IP networking.

Instead, I want to concentrate on the networking facilities and features available in Linux. This information should help you decide whether Linux is a feasible platform for your next embedded network-aware system.



Photo 1—The PC/104 card from VersaLogic basically just plugs in and works with Linux. I had to change the IRQ using the supplied configuration program because it conflicted with the IRQ of the serial port.

PROTOCOL STACKS

The most popular protocol stack in Linux is the TCP/IP stack, which is the traditional protocol stack needed to communicate with other hosts on the Internet. The TCP/IP stack in Linux is a high-performance implementation that takes advantage of all the tricks of the trade.

The programmer's interface to the protocol stack in Linux is the standard socket API, so it's easy to port network applications to Linux. In fact, most applications that use the socket API just compile and work under Linux.

At the link layer, the TCP/IP stack interfaces with a variety of link-level devices such as Ethernet, serial connections, and even wireless LANs. I'll talk about some of the available devices a little later.

Linux also supports Novell networking, known as IPX. IPX support in Linux provides the tools to configure IPX interfaces and manage IPX routing. Support for IPX comes in two flavors—minimal and full.

Most applications don't need full IPX support. Minimal support IPX has freely available tools and enables Linux to support Novell clients and servers, of which several kinds have been implemented commercially.

If you manage Apple computers, here's some good news. Linux also supports AppleTalk, EtherTalk (AppleTalk protocols directly on Ethernet), and encapsulated AppleTalk. Encapsulated AppleTalk tunnels AppleTalk over IP packets, so they can be routed by TCP/IP-only routers. AppleTalk support means you can share printers and disks between Linux and Apple computers.

Windows/DOS networking (basically NetBEUI and NetBios) is also supported by Linux. With session management block (SMB), you can export Linux file systems and printers to Windows machines as well as access those services from a Windows machine.

All available network stacks use the socket API, enabling application programs to use the networking resources. This situation isn't surprising if you know that the socket API was originally developed to permit access to different types of network protocol implementations. It's just that TCP/IP is the most popular stack supported with the socket API.

Programming with the socket API for a non-TCP/IP protocol is application- and network-implementation dependent. It'll take some work to port a native Windows or Apple application to Linux's protocol stack. Luckily, you can take a peek at the source code of many applications that already run under Linux to find out how.

Having the kernel sources for all of these networking implementations comes in handy because you can find out exactly how the network protocols work. With the original implementation of these protocols (e.g., AppleTalk under MacOS or NetBIOS under Windows/DOS), you couldn't look at the actual implementation. You had to rely on the documentation provided.

That's one of the biggest motivations for using Linux or other OSs with sources readily available, especially in mission-critical applications where you have to know what's going on in your product and be able to maintain it.

Because Linux is an open architecture, it's also possible to implement nonstandard networking protocols. Perhaps, you're using a nonstandard protocol for some previously developed proprietary architecture. Linux provides a good framework for porting your protocol stack to a standard OS. A Linux box could serve as an application gateway to a proprietary network implementation making it TCP/IP accessible.

LINK-LEVEL DEVICES

Along with the protocol stack, you need link-level networking devices. Recall from

my INK 98 article that link-level networking devices are the devices and device drivers that enable protocol packets to be sent over the physical wire (RF). Linux also provides a large selection of network drivers.

Linux supports just about every common Ethernet network interface controller (NIC) out there, including ISA-bus and PCI-bus-based controllers as well as 10- and 100-Mbps cards. For example, I use Linux's standard ne2000-based driver to control a PC/104 Ethernet card from VersaLogic (see Photo 1).

Linux has support for Ethernet cards on PCMCIA adapters and can support Intel- and PCIC-based PCMCIA socket implementations. Although this support is primarily for notebook based computers, PCMCIA adapters are available for some embedded-systems controllers. PC 104-based PCMCIA controllers are available from Real Time Devices, VersaLogic, and Ampro, as well as some other companies.

Using a PCMCIA controller in a PC/104 stack might seem strange at first, but there are many cards available in PC-card format that aren't available in PC/104 format. And, cards available in both formats (like modem cards and Ethernet cards) are usually less expensive in PC-card format.

On top of that, PC cards consume little power and are hot swappable. Of course, Linux easily supports hot-swapping PC cards.

Finally, there's a driver for the DLINK parallel-port Ethernet adapter. This adapter plugs into a standard PC parallel port and provides access to Ethernet. The performance isn't the best. After all, the parallel port can't transfer data at the maximum Ethernet speed of 1 Mbps. But, this setup works when there's no bus available to plug in a standard Ethernet card.

Besides popular Ethernet cards, Linux also supports a few Token Ring cards. Token Ring is still used in mainframe-era network implementations where your workstation talks to a mainframe (mostly IBM or Amdahl) application or database.

If you want run your network over a serial or even parallel port cable, Linux provides serial line IP (SLIP) and PPP drivers. SLIP is the oldest and simplest link layer protocol to run over serial lines.

It sends the IP packet on the serial line with a special byte to signal the packet

boundaries. There's also an escape mechanism so you can send the frame separator within a packet without confusing the driver.

PPP, on the other hand, is a complete protocol with state machines and so on, which runs beneath the IP layer. It has options for negotiating connection parameters between two hosts and enables more than just the IP to be transferred across the serial line.

For example, AppleTalk and Novell protocols can be transmitted over a PPP link. Even though PPP is more complex, it's the serial line protocol of choice, and almost all Internet service providers use it.

You could use SLIP and PPP to let your embedded Linux application communicate with a Windows machine. This is done with a null modem cable and a special null modem driver (for the Windows machine) available from the Internet.

If you configure your Windows machine to act as a dial-up server, then your embedded system connects to it and can communicate. Now, your web browser running on a Windows machine interfaces with a web-based GUI in the embedded application without actually putting the embedded system on an Ethernet-based network.

Besides serial lines, Linux also has a parallel port driver—parallel line IP (PLIP)—which runs over a LapLink cable. A LapLink cable is like a null-modem cable for parallel ports. Originally, it was used to enable laptop computers to exchange files with desktop systems. This application used a proprietary protocol and software.

PLIP is an open implementation enabling two computers to communicate using the TCP/IP suite over a LapLink cable (see Figure 1). Because just about every PC-compatible computer has a PC parallel port interface, this is a handy way of networking between a two-PC system.

One application that makes Linux popular for networking is its support for various WAN networking standards. Linux supports several sync serial cards that are needed for communicating over T1 (E1 if you're in Europe) with other routers.

Also, Linux has extensive support for ISDN when used with both synchronous serial and asynchronous interfaces. You can find Linux machines connected to WAN connections, when they're configured to act as routers or firewalls.

I've covered almost all the wired link-layer network devices, but I also want to mention that Linux has drivers for a few wireless LAN cards. The two supported types are WaveLAN and NetWave. Photo 2 shows a NetWave card, but several companies resell both types of cards under a variety of names.

However, supported cards are older models that aren't 802.11 compatible. So, WaveLAN can only talk to WaveLAN cards and NetWave cards can only talk to NetWave cards.

The 802.11 is an IEEE standard that's supposed to clear this up and enable all 802.11-compliant cards to talk to each other. The newer WaveLAN and NetWave cards support 802.11 now, but current Linux drivers don't.

Did I miss anything? Probably. I've only overviewed of what typically ships out with many Linux distributions. Many drivers and cards aren't in the distribution, but are directly supported by vendors. If your vendor doesn't provide Linux drivers and support, their competitors most likely do.

Many network drivers are even available in sources. Having sources for your OS and drivers enables you to build mission-critical systems that can be maintained by your company. Several companies use Linux in their embedded systems and include sources in every unit shipped.

It might sound crazy, but a system that is self-contained and documented in source code isn't such a bad idea. Consider, for

example, an elevator controller, which is likely to have a fairly long product life.

Should the controller break down, need to be upgraded, or need to have new hardware features integrated, it's easy to make changes to the system—even if the original development team at the company that supplied the system has gone on to bigger and better things or if the company has gone out of business.

I wonder if source code is included for mission-critical systems such as life-support and communications on the international space station. Our experiences with the Russian space station have shown that being able to innovate on the spot can be a good thing.

Having the source for device drivers also provides you with a good example of how to start writing a driver for your spiffy worm-hole interface card.

NETWORK APPLICATIONS

I mentioned that protocol stacks for TCP/IP, AppleTalk, and Novell are typically included with Linux. Many network applications are included with most Linux distributions as well.

Let's start with NFS, the network file system. NFS was developed by Sun Microsystems in the '80s and its protocol specification was made public so that other workstation vendors' implementations could talk to Sun's servers. Today, NFS is the most-used network file systems for Unix and Unix-compatible systems. Many non-Unix systems support NFS as well so they'll be compatible with Unix systems.

NFS can be run over UDP or TCP. UDP is simpler to implement, enabling even the smallest embedded systems to implement NFS services. TCP is more reliable and robust and offers, in some cases, better performance than UDP.

There are two parts to NFS: the server and the client. The NFS server runs on a machine that attempts to make some or all of its file systems available (exported) via NFS. NFS clients permit hosts to mount the NFS, exported by a server, and the read/write files it contains, on this mounted network volume.

One nice feature of NFS is that it's stateless. If either the client or the server crash, the file system can be remounted and all of the file-transfer traffic can continue where it left off. The server and client don't need to synchronize after a crash.

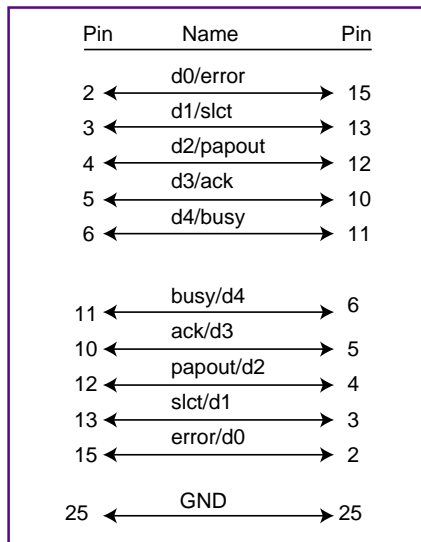


Figure 1—These are pinouts for a PLIP cable, which is sort of a null-modem cable for the PC parallel port. You can use this setup to attach a computer that doesn't have a bus or network interface to the 'Net.

Listing 1—This tiny web-server implementation is written in Task Control Language(Tcl). It implements a GUI for a DES encryption demo. The DES hardware is accessed with an external program (hwdes) that this script executes.

```

set hits 0
set done 0

proc htdecode {line} {
    set resp ""
    foreach field [split $line '&'] {
        lappend resp [split $field '=']
    }
    return $resp
}

proc connect {fd addr port} {
    global hits
    incr hits
    if {[gets $fd line] != -1} {
        if {[string length $line] != 0} {
            set request [split $line]
        }
    }
    while {[gets $fd line] != -1} {
        if {[string length $line] == 0} {
            break;
        }
        set header [split $line]
        if {[lindex $header 0] == "Content-length:"} {
            set len [lindex $header 1]
        }
        else {
            set len 0
        }
    }
    set fields ""
    if {[lindex $request 0] == "POST"} {
        while {[gets $fd line] != -1} {
            if {[string length $line] == 0} {
                break;
            }
            lappend fields [htdecode $line]
            set len [expr $len - [string length $line]]
            set len [expr $len - 2]
            if {$len < 0} {
                break;
            }
        }
    }
    set key ""
    set text ""
    set cmd "encrypt"
    foreach field [lindex $fields 0] {
        if {[lindex $field 0] == "text"} {
            set text [lindex $field 1]
        }

        if {[lindex $field 0] == "key"} {
            set key [lindex $field 1]
        }
        if {[lindex $field 0] == "button"} {
            set cmd [lindex $field 1]
        }
    }
    set answ ""
    if { [catch] {
        set pfd [open ". /hwdes \"$cmd\" \"$key\" \"$text\" r]
        while {[gets $pfd line] != -1} {
            set answ $answ$line
        }
        close $pfd
    }

```

Listing 1—continued.

```

} ] != 0){
    global errorInfo
    set answ $errorInfo
}
puts $fd "HTTP/1.1 200 OKKey d0Key"
puts $fd "Expires: Mon, 01 Jan 1970 00:00:01 GMT"
puts $fd "Pragma: nocache"
puts $fd ""
if {[lindex $request 1] == "/decrypt"} {
    puts $fd "<HTML><TITLE>Hardware Decrypt</TITLE>"
}
else {
    puts $fd "<HTML><TITLE>Hardware Encrypt</TITLE>"
}
puts $fd "<BODY>"
if {[lindex $request 1] == "/decrypt"} {
    puts $fd "<FORM method=\"POST\", action=\"encrypt\">"
}
else {
    puts $fd "<FORM method=\"POST\", action=\"decrypt\">"
}
puts $fd "<FONT size=\"+1\"><B>"
puts $fd "<TEXTAREA name=\"text\", rows=8, cols=35>$answ</TEXTAREA>"
puts $fd "</B></FONT>"
puts $fd "<P>Password:"
puts $fd "<INPUT type=\"password\", name=\"key\">"
if {[lindex $request 1] == "/decrypt"} {
    puts $fd "<INPUT type=\"submit\", name=\"button\", value=\"decrypt\">"
}
else {
    puts $fd "<INPUT type=\"submit\", name=\"button\", value=\"encrypt\">"
}
puts $fd "</BODY>"
puts $fd "</HTML>"
close $fd
}
set fd [socket -server connect 4321]
vwait done

```

Besides implementing NFS, Linux also comes with Samba, which is a Windows network implementation. Photo 3 is a screenshot of my Windows NT box mounting the hard disk of my notebook running Linux—over a wireless LAN, no less.

Linux has a variety of e-mail applications, including sendmail (the standard Unix mail exchange agent) and several mail interfaces, as well as several post-office protocol (POP) implementations popular with ISPs. Linux also implements e-mail over UUCP which is an older batch mode serial line protocol that can be used efficiently over dial-up modems.

UUCP is interesting because it's a file transfer and remote execution system. A host queues up batches of transactions it wants to perform on a remote host. Whenever the host dials up, these batches are exchanged and the connection is terminated.

The remote host then executes the commands in the transaction asynchronously to the connection and, if there's a response to be sent back, queues it up to be sent back. It's the ultimate store and forward system.

UUCP can also compress all the traffic over the communication link. Such efficiency helps when the connections between hosts are expensive and have low bandwidth.

Also, the connections don't need to be real time. Commands are submitted and the response is sent back later. UUCP is how e-mail and Usenet were transmitted in the days before the Internet. UUCP can be used for remote data loggers or devices where channel utilization is important, or implemented to enable systems to communicate over slow satellite links.

Linux has standard nameserver applications for the domain name system (DNS).

With DNS, hosts find out how to map a hostname or domain (e.g., www.ezcomm.com) to an Internet address xx.xx.xx.xx. The DNS client is the software that looks up the domain name and maps it to the address while the DNS server (or name-server) implements the portion of the database that does the mapping.

Each domain needs a name-server that contains a database of records that tells how names are mapped to addresses. Because many ISPs and organizations use Linux machines as the nameserver for their domain, Linux includes the nameserver. Although the nameserver probably isn't too important for an embedded device, the client library, called the resolver, can be.

If you want to use web-based services, full-featured remote printing is also supported. I mentioned that Samba and AppleTalk can connect with Mac and Windows-based printers, but Linux also supports Unix printing using lpd.

Also, GhostScript (a freely available PostScript-compatible interpreter) can be used to translate PostScript to the raster images used by many printers. It's possible to implement PostScript-compatible printer devices that are accessible via Windows, Unix, and Mac systems. GhostScript has drivers for many popular inkjet and laser printers already.

The web server included is used by several major ISPs. It's robust and has many features necessary for hosting web sites. It may be overkill for most embedded applications, but it's there if you need it.

More interestingly, Linux distribution also contains scripting languages like Perl/Tcl and Python. In either of them, it's possible to implement a small embedded web server to implement web-based GUIs.



Photo 3—Here's a screenshot of the Windows NT machine in my basement office mapping my hard disk on my notebook. The notebook has a wireless LAN card and is currently on the kitchen table. Linux also enables me to mount my Windows 95 partition on my notebook and make it available over the network. So now I can map a Linux-based volume, which has a Windows 95 partition mounted on it, all from Windows NT over a wireless LAN.

Listing 1 shows how I implemented a small web server for the interface to a DES implementation running on a FPGA board (featured in my *INK* 99 column).

There is even a Java implementation for Linux, although you'll have to download Sun's run-time class library to make it work. With Java, you can implement small embedded GUIs and web server fairly quickly.

Typical Linux distributions include everything but the kitchen sink, and if you can't find a feature you need, check the 'Net. But don't worry, you don't have to take everything. You can choose a mail program and sendmail and nothing else if that's all you need for your embedded applications.

NETWORK STARTUP

A desktop system typically needs several applications and services provided by Linux distributions. It'll probably use a PPP driver, the TCP/IP stack, and perhaps Netscape as a web browser. Oh, did I forget to mention that Netscape now offers a free Linux version of their web browser?

What if you only need the socket API and TCP/IP stack so your application

Listing 2—This minimal startup script of code is all you need to start networking in an embedded system. At the very least, you need to set the Internet address of the interface with the `ifconfig` command and provide a default route using the `route` command.

```
[
  if config eth0 {ipaddr}
  route add default {gateway}
  your_application
]
```

program could directly communicate with it? That's easy—just build a kernel. Most likely, TCP/IP networking is already enabled in the kernel config.

You could also enable whichever link-layer device driver you want in the kernel config. Just build the kernel and prepare a boot image like I described last month.

Listing 2 shows what is needed in a `.profile` or `/linuxrc` startup script to start networking in Linux.

```
ifconfig initializes the network driver and makes it available to the TCP/IP stack. The {ipaddr} argument is the IP address of the embedded device. The second line adds the default route needed to get to hosts outside the network that the embedded system is connected to. {gateway} is the address of the router on the attached network that knows how to send traffic on.
```

You have to make sure that `ifconfig` and the `route` program are available on the root file system image. If you configure the kernel to make some of the network external modules, you need to include those modules in your disk image and load them before enabling the network.

Typically, you can use the `insmod` program to accomplish this step. Having module support for device drivers is particularly nice when using PC cards because you never know what kind of card can be inserted into the PCMCIA socket.

ANY QUESTIONS?

What doesn't Linux do? Not much. With networking stacks and applications, it's easy to solve network-based problems with Linux.

About the only drawback is that Linux needs a 32-bit protected-mode processor to run. The lowliest i386ex is fairly cheap these days, so that's not too bad. Even so, work continues on nonprotected-mode implementations.

Speaking of drawbacks, once again I've run out of space long before I've run out of ideas. But, hopefully you've seen that Linux just might be the OS of choice for any Internet or Ethernet devices you're considering. *RPC.EPC*

Ingo Cylix has been writing for INK for two years on topics such as embedded systems, FPGE 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 cylix@derivation.com

SOURCES

RT-Linux
rtlinux.cs.nmt.edu/~rtlinux/

IPX Networking

Novell, Inc
 (801) 222-6000
 Fax: (801) 861-3933
www.novell.com

NetWare Linux software

Caldera, Inc.
 (888) 465-4689
 (801) 765-4999
 Fax: (801) 765-1313
www.caldera.com

PCMCIA controllers

Ampro Computers
 (408) 360-0200
 Fax: (408) 360-0222
www.ampro.com

Real Time Devices USA
 (814) 234-8087
 Fax: (814) 234-5218
www.rtdusa.com

Versalogic
 (800) 824-3163
 (541) 485-8575
 Fax: (541) 485-5712
www.versalogic.com

Fred Eady

In the Face of Medusa

Part 1: Developing Reliable Control

Fred believes any monocomplex embedded system will turn to stone when faced with this NASA ground support unit's host of pumps, valves, and cameras. So, he sent up PicStic to meet the challenge. Has Medusa met her match?

Sometimes it makes good sense to augment the power of a '386 or '486 embedded system with additional peripheral processing. Sure, you could design a system solution using just the 'x86 or Pentium platforms, and most of the time, that would be the best solution. But, sometimes it's not.

By using smaller and less expensive peripheral processors, to support a good 'x86 design, you can cut costs and complexity. Placing a subordinate processing platform in a specific job role offloads cycles from the main embedded processor and breaks the code into more manageable pieces. If the peripheral processor scheme is well thought out, a gain in overall system productivity is possible.

Think of it this way. Your 'x86 is busily cranking away on numerical calculations when an interrupt or clock event signals that it's time to perform some I/O. Instead of sending a quick command to a peripheral processor to move a motor or turn on a valve, you stop your number crunching

to take care of it from the 'x86 firmware or hardware.

Now that you've initiated the process from the 'x86 system, you also have to make sure it completes successfully. Another burden on your already busy 'x86 CPU, and you can't continue your calculations until you finish the I/O operation.

The ultimate answer is to buy some expensive multitasking, multithreading OS

and pile on the expensive CPU, megs of memory, and I/O hardware. You could do that or...

MARK IS IN THE BUILDING

Once again, I managed to get my hands on some real fight hardware from my friend Mark, "the Orbiter machinist." Seems there's intelligence needed for a ground support unit that squirts water into petri dishes to grow plants. Take a look at Photo 1 and you'll know why it's named Medusa.

But, Medusa's not the whole story. Intelligence is also needed to move a CCD camera over the petri dishes to take pictures of the growing and feeding process. As with most experiments of this type, there's a myriad of pumps and valves that need attention, too.

A perfect solution for this bsby is an embedded '386 or better embedded system. By the way, this baby also needs to have its temperature checked and its diaper looked at as well. Can a

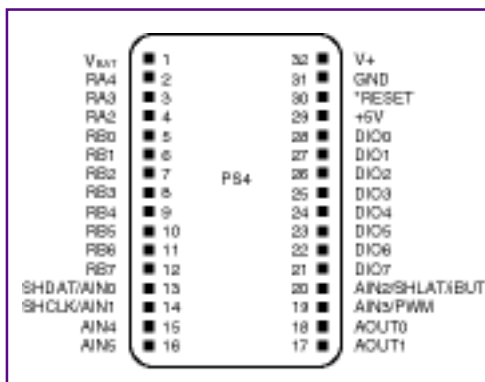
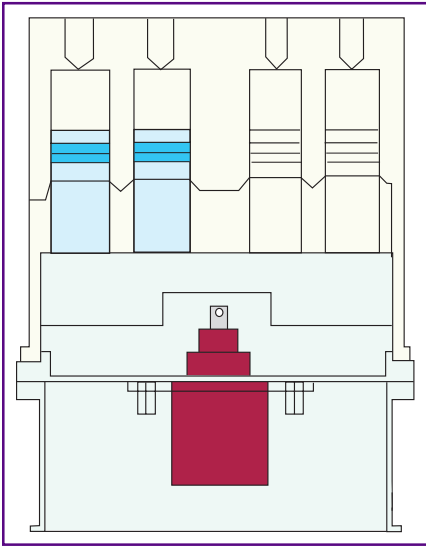


Figure 1—Here's a look at the 32 pins of a basic PicStic-4.



single embedded platform, all dressed up in the proper hardware, provide the perfect solution? Well, sorta.

ENEMY IDENTIFICATION

The first problem is that any mono-complex embedded system is a single point of failure. If the embedded 'x86 system croaks or jams up, the whole experiment may fail unless an astronaut or scientist comes to its aid in time. Besides, we all know those astronauts are just in it for the thrill ride. The scientists do the real work, right?

The second problem I see is a lot of expensive add-on hardware to accommodate the control of the valves, pumps, and camera drive. And don't forget taking temperature readings and monitoring humidity. All of this is compounded by more complex 'x86 firmware that's needed to monitor and failsafe the devices being controlled. Complexity is directly proportional to cost. You do the math.

I'm sure you've all heard stories about how the NASA folks lean toward redundancy to improve the safety factor. Well, the same premise can be applied to the experiments that fly too. But, it would be impractical to place numerous embedded 'x86 systems within our Medusa experiment.

First of all, the price would be prohibitive and the space allotted for the experiment is insufficient for that much hardware. By the way, there's no local power company supplying the power, either.

So, how can we make this experiment perform with a minimal 'x86 system using smaller peripheral processors? My answer: the Micromint PicStic.

SHIP IN THE DISTANCE

I liken the PicStic to a ship in the distance. It may look like a dinghy from afar, but as you come closer to its hardware and get to know its power, it arrives in port as an aircraft carrier with you as the captain.

The PicStic is small, lightweight, inexpensive, and easy to implement. You can choose from several variants depending on your application's requirements. For Medusa, I chose the PicStic-4Q.

The 4Q is a rectangular solid with its longest side measuring 1.5", and it takes up about 0.57421875 in.³. Power consumption is less than 75mW, and as long as the astronauts are comfortable, the 4Q module is, too.

If harsh environments are expected, you can obtain 4Qs with extended industrial operating temperature ranges. The device is housed in a protective cover with the hardware features accessible via 32 dual-inline pins. The pinout is shown in Figure 1.

Not only is the 4Q rugged and economical, but it's also highly capable and easy to program. You can program your 'x86 embedded firmware in C, BASIC, or assembler, right? Same for the 4Q.

This implies a minimal learning curve for any embedded programmer. As you might have guessed, this module is based on Microchip's '16F84.

The '16F84 is one of those EEPROM-based microcontrollers that needs no special erasing lamps to clear the program and data memory areas. The '16F84 core is teamed up with an I/O coprocessor to provide a wealth of I/O pins and built-in functionality.

Instead of writing every little bit of code to perform a simple switch read, the 4Q's firmware contains such routines. You simply call them as you would in a desktop-based BASIC program. All of the switch debounce software and input code is already built into the PBASIC call.

FLYING THE 4Q

Now, let's apply the 4Q to the Medusa experiment. Medusa's only function is to nourish the life in the petri dishes. The actual laboratory experiment has Medusa feeding three sets of eight petri dishes.

The inner workings of Medusa consist of a series of syringes containing liquefied nutrients that are delivered to the petri dishes by pressure applied from the shaft of a linear stepper motor. A cross section of the Medusa is shown in Figure 2.

Medusa is the first module of our experiment in which we will embed a 4Q. The linear stepper motor is driven using an Allegro UNC5804B. The UNC5804B is a BiMos II Unipolar stepper-motor translator/driver. The logical depiction of the UNC5804B is shown in Figure 3.

This IC provides complete control and drive for a four-phase unipolar stepper-motor. The PicStic-4Q need only supply STEP, DIRECTION, and ENABLE signals to the UNC5804B to move the linear stepper, which in turn puts the squeeze on the syringe plungers.

Each step of the HSI linear stepper is one half-thousandth of an inch, so a small amount of nutrients are supplied at each

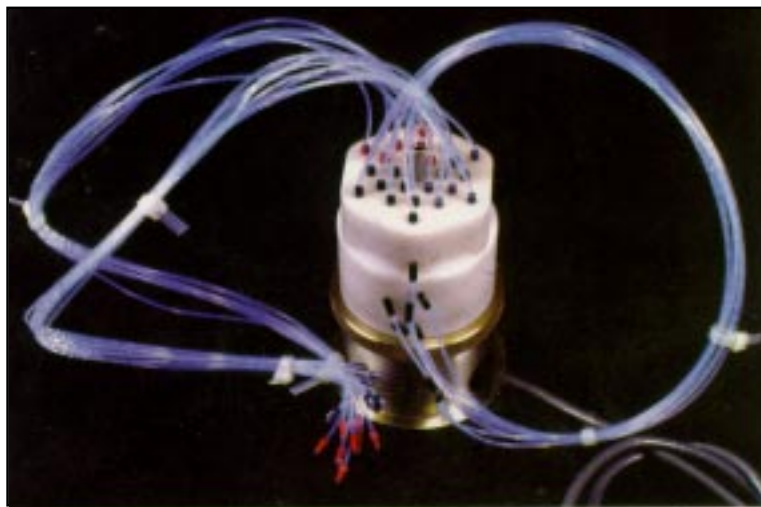


Photo 1—It's rather ugly, but at least you can look at it without turning to stone

plunger move. The object is to feed the experiment without drowning it.

Seems like a waste to only use three I/O lines on the module, but I haven't finished telling you about the other things the 4Q attached to Medusa needs to perform.

Again, NASA is synonymous with redundancy. Redundancy implies safety, and safety improves the possibility of success. I pointed out that a single point of failure exists when using a single processor complex to do the entire job. I also sug-

gested that peripheral processing was one way to eliminate a single point of failure situation.

The 4Q comes equipped with a battery-backed real-time clock, and there's also a built-in serial communications function. The embedded 'x86 system is also capable of keeping experiment time and communicating with peripheral serially. We can use these attributes to build a did-I-get-fed safety factor.

The 'x86 is ultimately the boss here. Although the 4Qs have their own intelligence, the 'x86 controls and monitors

their grammatical movements. It's safe to say that the 'x86 embedded board is able to reset any of the 4Qs as well as command them to perform preprogrammed functions at the predetermined times.

What if the 'x86 experienced an I/O component failure? What if the firmware running on the 'x86 hung?

My first thought is that the 'x86 onboard watchdog timer would reset the hung embedded complex and things would resume as normal. Watchdog timers are great for software hangs, but I haven't seen one yet that can fix hardware.

So let's assume the 'x86 hardware is down. What are the 4Qs to do? After all, they can't work without the direction of the big daddy 'x86, right? Wrong. The solution is to use the hardware incorporated into the 4Q to enable each one to continue to function until the 'x86 problem is repaired.

It's a given that catastrophic failure of the 'x86 hardware would result in the loss of critical data and, ultimately, experiment failure. But if the 4Qs are programmed to continue their assigned tasks according to mission time, and buffer any gathered data during the 'x86 downtime, depending on how long the 'x86 hardware was inoperative versus how much data the 4Q could store, data loss wouldn't be total.

The 4Qs can also be programmed to poll each other to see if an unsuccessful data transfer to the 'x86 occurred for other modules in the experiment. This is akin to how the mission computers on the Orbiter vote each other out if a computational discrepancy occurs.

If all the 4Qs involved with Medusa can speak to each other and not speak to the 'x86, it's logical to assume that the 'x86 is offline. I fight with this concept in my mind, but the 4Qs can also be programmed to attempt to reset the 'x86 embedded system if a consensus of them decided to do so.

I see a never-ending loop in that idea, but it can be implemented as long as the proper controls are put into place. The key to success here is to synchronize the module's clocks with the clock of the 'x86 at experiment startup. If the 'x86 system doesn't issue a feed command within the specified time limits, then the module responsible for driving the linear stepper motor assumes command and executes the process.

Any resulting feed operation status that should have been transmitted to the host 'x86 is stored in the 4Q, just in case the 'x86 was busy at feeding time or was unable to recover from a crashed condition. The inverse of this scenario is that the module performing the feed function fails.

The bad news is that the experiment would fail because there would be no backup for the 4Q. The good news is that the probability of this module failing is minimal. All of its parts have proved reliable in a variety of real-world applications.

Besides applying pressure to Medusa's plungers, the same 4Q could be configured to measure temperature and humidity as well. The 4Q is also equipped with a pair of ADC's. Depending on the accuracy and number of inputs required, you can choose from a 4-channel 8-bit or a 2-channel 12-bit ADC.

Each ADC is based on a 5-V full-scale value, so sensors and interfaces must be chosen accordingly. A good choice here is the HyCal monolithic IC humidity sensor model IH-3602. This sensor measures relative humidity as well as temperature and is packaged in a six-pin TO-5 package.

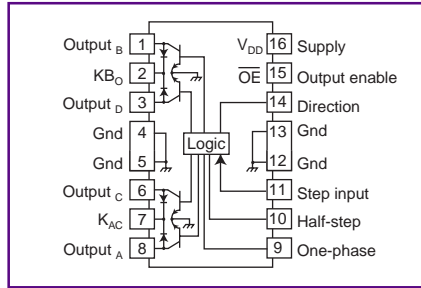


Figure 3—A few years back, I wrote an application and hacked some hardware to do what this little IC does.

Temperature is measured using a thin film platinum 1000-Ω RTD. The temperature sensor can be used to provide temperature compensation or to measure temperature independently.

The real plus is that this sensor interfaces directly with the 4Q. A nominal 0.8—4-V linear output for 0-100% relative humidity is provided when the sensor is powered by 5 VDC.

Temperature is measured by detecting a change in the RTD resistance. This is done by supplying a small constant current to the RTD and reading the voltage across the platinum resistance.

ROLL THE CAMERA

A second peripheral processor rotates a stepper motor attached to an arm that holds a CCD camera. In this application, the module must be able to home the camera arm using an infrared interruption sensor, turn on IR flood LEDs, and activate the CCD camera.

Moving the stepper motor involves the same procedure that I used earlier to extend the linear stepper motor driving the syringes. The algorithm is:

- home the camera arm at the beginning of experiment time
- wait for a picture command from the 'x86 embedded system
- activate the IR flood LEDs
- activate the CCD camera
- wait for end-of-picture command
- deactivate the CCD camera
- deactivate the IR flood LEDs
- move the camera arm to next target
- execute from step two until the end of the experiment

The 'x86 system is equipped with a frame grabber. So, the camera module executes the mechanical motion and lighting procedures, freeing the 'x86 to spend its cycles obtaining the resulting image.

Although not included in the algorithm, the module is also capable of recalibrating the camera arm at any predetermined time deemed necessary by the experiment scientists, without interrupting the 'x86 processing thread.

The 'x86 can also authorize a recall operation via command to the 4Q in charge of the camera stepper motor. 'x86 failsafes for this 4Q are identical to the Medusa 4Q, and the inclusion of an IR sensor to monitor the IR LEDs is included in the camera-driver module firmware.

As far as the 4Qs are concerned, their destinies are clad in firmware. You could control the remaining valves and pumps with the 'x86 hardware, but there's plenty of 4Q cycles left.

Another strength associated with using peripheral processor complexes is that, because each peripheral complex responds to commands, functionality can be spread across all of the peripheral processors regardless of their primary job.

In other words, if there's enough program code resource and processor cycle time in the camera module to toggle a

valve and I can send a toggle valve command to that 4Q, then I can put the valve control functionality into that 4Q.

Thus, I can load balance my peripheral processors and still conserve cycles on the main 'x86 board. Basically, I end up with a simple half-duplex serial network. The 4Qs don't speak until spoken to, unless a failure of the 'x86 is suspected.

So I can hang as many modules as necessary, and whenever I need to, on this experiment. I both conserved 'x86 cycles and implemented a cost-effective distributed-computing environment. The offloading of 'x86 CPU cycles to the 4Qs enables use of a lower cost 'x86 embedded board, which next to the experiment hardware, is the most expensive piece of computing equipment in the experiment.

THE CENTRAL SITE

Now that the module's tasks have been defined, let's focus on the 'x86 system. Using the 4Q peripheral processors means I can choose a low-cost '386 system. My choice is the Octagon Systems 4010.

The 4010 is an 80C386CX running at 25 MHz. It has the standard AT-compatible BIOS and uses standard PC-type peripherals like floppy and hard drives. Ideally, the fewer moving parts the better, and the 4010 is equipped perfectly. There's 512 KB of flash memory and 2 MB of DRAM, which can operate just like their mechanical cousins.

And, just in case they're needed the 4010 can also accommodate a 2.5" hard disk or standard 3.5" floppy drive.

The serial ports of the 4010 will be busy handling the peripheral processors, so a means of communicating the collected data, other than the standard serial ports, is necessary. I just happen to have an Octagon 5500 Ethernet card that plugs into the 4010 card cage to transfer data out of the Medusa experiment.

THE COUNTDOWN

We have some one-of-a-kind hardware fresh from the lab, along with some powerful little devices called PicStics. The idea is to allow the PicStics to operate under control of a '386 embedded-processor complex. Additionally, the PicStics can take over their respective roles if the '386 system is busy doing other things.

Failure of any component during the experiment time is unacceptable. So, the

hardware I chose is robust and reliable. The 4010 has an MTBF of 11 years. I've worked with Microchip PICs for many years now and can count on one hand the non-customer-induced failures I've seen.

The system concept is described and the jobs are well defined. The next step is integrating the '386 and the Medusa experiment hardware with the peripheral processors. Once that's accomplished, the Ethernet conduit must be affected so the data can be put onto a more meaningful platform for the scientists to evaluate (e.g., an Ethernet-capable laptop).

Next time, I'll show you how to integrate the PicStic-4Qs and the Octagon Systems 4010 into a synchronous system to support the requirements of the Medusa experiment. I'll also take you through the steps to effect an Ethernet interface between the 4010 and a scientist's laptop.

I realize this article has been off the beaten path as far as some folk's definition of embedded is concerned, but I doubt that you're a typical cookie-cutter professional. The 4Q is a good fit for this application and proves again that it doesn't have to be complicated to be embedded. APC/EPC

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

SOURCES

UNC584B

Allegro Microsystems, Inc.
(508) 853-5000
Fax: (508) 853-7861
www.allegromicro.com

PIC16F84

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

PicStic-4Q

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

80386 4010

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

IH-3602

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

FEATURE ARTICLE

Gordon Dick

The PCL3013 Step/Servo Motor Controller in Action

If you need a high-performance step/servo motor controller, check in with Gordon. Along with its unique approach to program memory (data is written to preregisters), the PCL3013 offers so many interesting features, you won't want to miss out.



It's getting harder and harder for the average person to mess around with stuff! Most folks won't build a PC board just to test this new micro or that new controller if the device is only available in a surface mount package.

Oh sure, evaluation boards are available, but that changes a \$10 ding in the wallet into a \$100 dent. And even if you go to the trouble of building a PCB, it takes some skill to solder a surface-mount device by hand. The day is approaching when, unless it's part of your job, you won't be able to tinker with whatever device is currently new and hot.

The PCL3013 is a high-performance step/servo motor controller with an exciting array of features and it was a newly introduced device when I began this project four years ago. Like most step/servo motor controllers, it's intended to be used with a host micro directing its operations. It keeps track of the time-critical items and does any required calculations, freeing up the host to do other things.

The PCL3013 features control of step motors or pulse input servo motors,

linear or S-curve acceleration, microstepping, stepping rates up to 4.9 Mpps, and out-of-step detection. It also offers a Motorola or Intel interface, an 8- or 16-bit data bus, 12 different origin returns, and interrupts that signal various internal events. With all of these features, it's no surprise that this device has a large number of pins, as illustrated in Figure 1.

Solutions to many problems I previously encountered seemed to be at hand with this device. Let's see how it addresses them.

DOING IT RIGHT

First off, let's say I have this device—thanks to the folks at Kollmorgen for the samples—that I'm anxious to get working. Should I try to haywire it together?

Or, maybe there's an adapter unit out there, something that lets me solder my device to it and then have wire-wrap pins to work with. Such adapters are available for a variety of surface-mount styles, but not this one because of the lead spacing. The lead spacing of this device is metric because it is made by Nippon pulse, a Japanese device manufacturer.

I care how it looks and it'll probably save me time in the long run to do it right the first time, so I opted to make an adapter. That turns out to be more work than I thought. (How many times have I jumped into a project and said that?)

MADE A PCB LATELY?

Like everyone in the industry, we threw out our tape and donut supply quite a while back. All our PCBs are made using a CAD PCB design package.

When I complained to one of my colleagues about how easy it was to make a PCB in the good old days and how hard it is now (since I don't use powerful PCB CAD packages that often), he told me about EasyTrax. It's available for free on the 'Net and it's easy to learn. In about three or four hours, I was done with the tutorial, and in another three or four hours, I had my adapter layout complete.

For many of the PCBs we make, there is no longer any photography involved. A negative is made in a

laser printer on a transparency. Once I have the negative, it isn't long before I have a decent-looking adapter board waiting for a device, pull-up resistors, and wire-wrap pins to be installed.

How often do you solder a surface mount IC to a PCB? I don't do it often. Examining my first attempt under a microscope reveals it's not perfect. A few leads may not be connected. After a touchup or two and a vigorous wash in alcohol, it looks all right. The pull-up SIPs and the wire-wrap pins can now be installed. I'm getting close to wiring!

BUILD THE PROTOTYPE

Most of the hardware for the prototype mounts to the wire-wrap perf board but not my just-completed adapter board. The wire-wrap pins on its metric grid don't match the perf board's inch grid. A suitable hole has to be made in the perf board and the adapter board glued over the hole.

I don't know when I've taken on a project that involved so much in the way of background work before I could get to any of the fun stuff. By this time, I was getting itchy to put this chip through a few of its paces.

The completed prototype is shown in Photo 1, and you see the schematic in Figure 2. The elements in the schematic are similar to other intelligent step motor controllers.

A clock source was required. In this case, I had an oscillator can that was the correct frequency, so I used it. I could have used a slower clock signal from the 'HC11 board.

I chose a mature step-motor driver chip to keep the support circuit simple. A reset circuit and some LCDs with a driver complete the schematic.

BUT DOES IT WORK?

Usually, there's a certain amount of pain associated with getting any project working. Maybe it's wiring mistakes that have to be corrected or timing problems that have to be dealt with. Or in some cases, a servo amp wants to oscillate.

Amazingly, none of that happened. I can't explain it.

Maybe I'd already paid my pain quota with all the hassle I had getting to this point.

The monitor program I'm using for the 'HC11 allows reading and writing memory or I/O without generating new code. This feature is useful when connecting a new I/O device.

To verify that communication with the PCL3013 is OK, I first attempted a read of the status word. The data contained in the read back of the status bytes appears to be correct, given that I tied some signal lines high that wouldn't be used. That's a good sign.

A write attempt is next. A general-purpose I/O pin is told to be an output and then try writing it 0 and then to 1. That works. I try it with a different I/O pin and that works, too. I'm pretty confident now that the data bus connection between the 'HC11 and the PCL3013 is functional.

INTERFACE DETAILS

It's common for a peripheral I/O device to have times when it's busy and won't read or write data. The PCL3013 is similar in that you should wait 200 ns after sending a read register command before letting the micro try to read. The 813-ns micro bus cycle provides the necessary time delay.

The PCL3013 needs 16 bytes of I/O space, which is

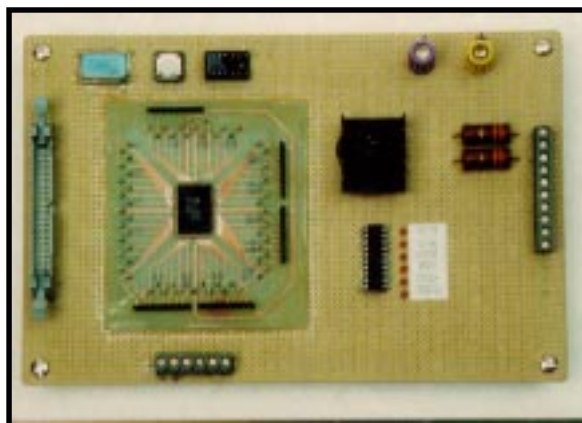


Photo 1—The box header allows connection to the 'HC11. The screw terminals on the right-hand side are for the motor/encoder, and the screw terminals on the bottom left are for the manual pulser. Power for

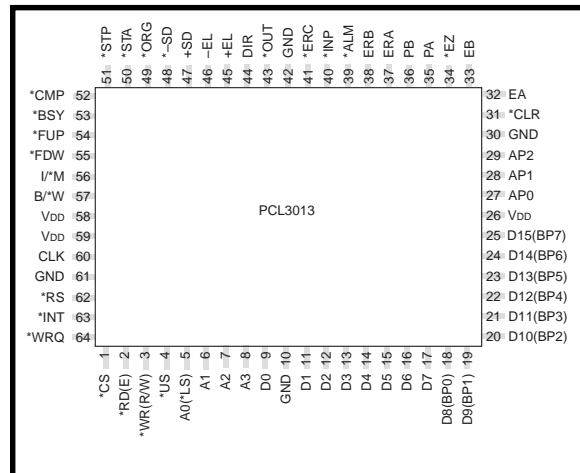


Figure 1—To cram in all the features, the PCL3013 needs 64 pins.

quite a bit more than the I/O devices I've worked with until now. The I/O space on the development micro is partitioned into 800h blocks, so the I/O space requirement was not a problem.

The device interfaces easily with Motorola and Intel micros: there is an I/*M control line (read *M as the complement of M). And, it supports 8- or 16-bit data buses: there is a B/*W control line.

WHAT CAN THE PCL3013 DO?

A lot. I wasn't able to test everything, but more on that later on.

This device has a large number of dedicated I/O lines to support more than the typical switches associated with a motion stage. For example, slow-down switch inputs help prevent the high-speed end-of-travel crashes that happen even when end-of-travel limit switches are present.

The amount of general-purpose I/O varies, depending on the bus width of the micro. Since the 'HC11 bus is 8 bits wide, the 8 bits that would otherwise be used to support a 16-bit bus become general-purpose I/O. There are three other general-purpose I/O lines.

Now it's time to generate some code and exercise this device a little. But first, I want to say a word about notation.

When referring to registers in this device, they are specified by number (e.g. R1 for register 1). Many registers have pre-registers that hold data until a

start command is issued. Then the data is transferred to the working register.

First, decide what mode you wish to operate in. Four bits in the operation mode buffer are dedicated to mode. In this case, I wanted to do some simple jogging just to get started. The PCL-3013 jogs when in continuous mode 1, and the direction is defined by another bit in the operation mode buffer.

Initially, I tried loading only R1 (FL pulse rate register) with the low-speed stepping rate and R4 (multiplication factor register) with a suitable multiplier. This technique works, but only out of reset. If it was stopped, it wouldn't start again until it was reset.

The manual cautions you to keep the high speed stepping rate R2 (FH pulse rate register) larger than R1. Since I wasn't using R2, I didn't think it needed to be loaded. But as soon as I loaded a valid number into it, the unit worked fine.

With data in both R1 and R2, I was able to use two of the seven start commands. Sending 10h to the command register results in low-speed jogging at the rate determined by R1 (and R4), and sending 11h results in high-speed jogging at the rate determined by R2 (and R4). The 'HC11 code used to do this jogging test is shown in Listing 1.

Once the 'HC11 has written into the appropriate registers, I used the monitor program to send start and stop commands. Although there are three different stop commands, for this test, the only valid one is 9h, which produces an immediate stop.

A small note regarding code is appropriate here. For the features of the PCL3013 discussed in this section, code was created and the feature exercised. Because the feature is of more concern than the code, I only provide you with one example (see Listing 1). In the following section, I'll discuss features that were not exercised.

If a value is loaded into R3 (acceleration/deceleration rate register), you can make starts using 13h. Now the motor accelerates linearly with the acceleration time determined by a formula involving R1, R2, R3, and the 19.66-MHz clock.

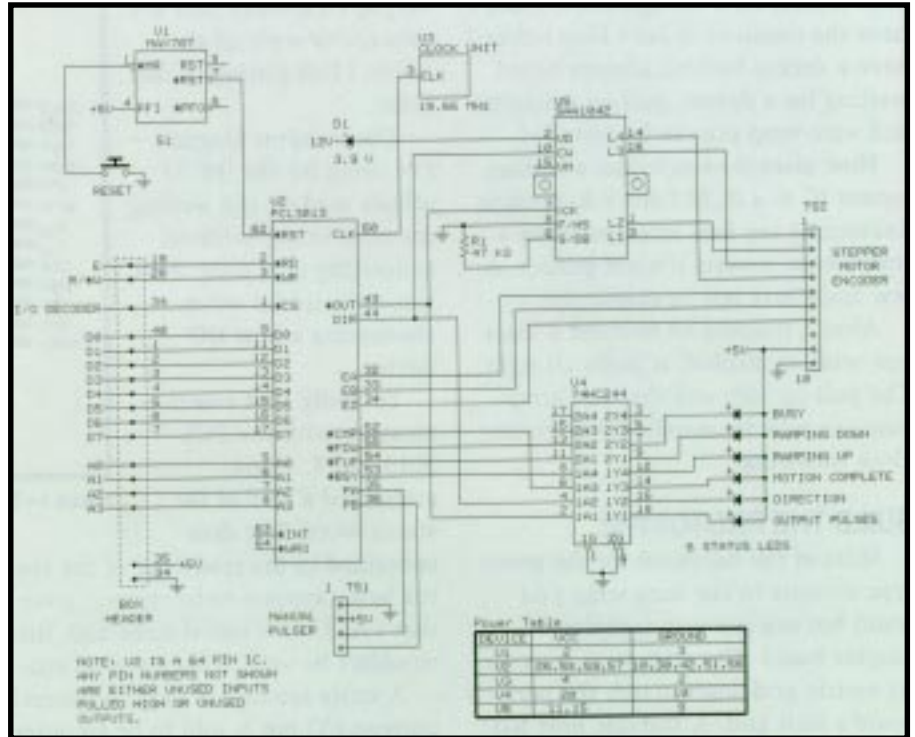


Figure 2—Some very sophisticated control of a step motor may be achieved with this circuit. The circuit is simple because the PCL3013 contains the complicated control features.

By using fairly large value in R3 and the existing numbers in R1, R2, and R4, I was able to get an acceleration time in the 2=s range. When the motor started, it was clearly accelerating. Similarly, when the Ah stop command was used, the deceleration was obvious.

This unit has the ability to do S-curve acceleration as well. The choice between linear and S acceleration is made with a bit in the control mode buffer.

When a comparison was made using the two types but keeping the same acceleration time, I couldn't tell the difference by just watching and listening to the motor start and stop. The difference has to be more noticeable when the motor is driving a motion stage.

Two flavors of relative moves are available: preset mode 1 and 2. In preset mode 1, the relative move distance is loaded into R0 (output pulse register) as an unsigned number from 0 to 268,435,455. The move direction is established by bit 4 of the operation mode buffer.

In preset mode 2, the relative move distance is again loaded into R0 but this time as a signed number from -134,217,728 to 134,217,727. Bit 4 of

the operation mode buffer has no effect on this mode.

Using either of the relative move modes, you can issue multiple start commands, causing multiple moves as would be expected when making relative moves. Actual position is kept in R9 (up/down counter register), and it too can be configured to use signed or unsigned values by toggling bit 28 of R6 (environmental condition register 1). As I said, this is a powerful device and it has lots of registers.

To make absolute position moves, use preset mode 3. If multiple start commands are issued after a move, they are ignored as expected since the device is in the desired position after the first move command.

If your application requires time delays, the PCL3013 can do that too by selecting the timer mode in the operation mode buffer. The description of this feature in the manual was difficult to understand at first because nowhere is any mention made of producing a time delay.

The PCL3013 produces a time delay based on the contents of R0 and the current stepping rate as determined by R1, R2, and R4. For example, if a low-speed start command is used, the step

ping rate is determined by R1 and R4, so the delay time is based on that rate and the number in R0.

When I first looked at the manual for the PCL3013, I was surprised that there didn't seem to be any program memory. Other devices I have used could store complete motion programs of various lengths. Eventually I learned that the PCL3013 features a different approach with similar results.

Data for the next move can be written into preregisters while the present move is in progress. On completion of the current move (which can be signaled via one of a large collection interrupt sources or determined by polling one of the many status bits), a start command can be issued for the next move. Since the necessary data is all present, only the start command needs to be written. And, you can do one better if you wish.

If bit 4 of the control mode buffer is set, the unit automatically starts the next operation if it is in "settled" status. To be settled, the registers for the next move must have had valid data written to them at some point, and a *stprt* command must have been issued prior to the current move finishing. If those conditions exist, the PCL3013 continues on to the next move as soon as it finishes the present one.

Of course, the host micro still has to keep track of what's happening in the PCL3013. For example, it can't write any data other than what will be used next. The PCL3013 won't stack a collection of data to be used for several moves in advance.

Would you like to microstep? The PCL3013 is good at that, too. Select the number of microsteps per step anywhere from 1 to 256 by entering the number you want minus 1 into bits 24 to 31 of R7 (environmental condition register 2).

I didn't use a microstepping drive to test this. Using a 100-step/rev motor and setting up for 10 microsteps per step, a move distance of 100 produced 10 revs. This calculation would be correct if a microstepping drive using 10 microsteps per step fed the motor. The only thing I found a little strange is that the PCL 3013 keeps track of full steps rather than microsteps.

There are two 28-bit comparator registers, R10 and R11, which can be used in various comparison scenarios to implement tasks. For example, you can compare R10 against R0 or R9. Or compare R11 against R0 or R9. Or compare R10 and R11 against R0 or R9.

The selection between R0 or R9 is made with bit 22 of R7. By setting bits 20 and 21 in R7, you can change the low- and high-speed stepping rates when the comparison condition is met. I tested this by doing the on-the-fly speed changes. It works fine.

The comparison condition is specified in bits 16-19 of R7. Here's where you indicate that the comparison is equal to, less than, or greater than the specified counter value.

You also indicate here if the comparison uses just R10, just R11, or both. If you wish, you can produce a hardware interrupt as a result of a comparison condition being satisfied.

Of course, I saved the best for last. In continuous mode 2, pulses from an external encoder can be used to cause the motor to step. When I read about this feature, my first thoughts were about slaving one system to another or electronic gearing.

Oddly, the manual discusses this feature under the manual pulser input heading. I tested it using a manually rotated encoder to cause motor movement. If the pulse rate from the external encoder exceeds the high-speed stepping rate established with R2 and R4, operation becomes erratic. But, the manual warns that this will happen.

A variation on this feature is implemented with preset mode 4. Here, an absolute position is moved to by rotating an external encoder. No matter which way the encoder is rotated in this mode, the motor moves toward the absolute position specified as long as the encoder is producing pulses.

Listing 1— This code tests the ability to operate the PCL3013 in continuous mode 1. After executing the code, start commands 10h and 11h and the immediate-stop command 9h may be issued from the monitor program.

```
;* EQUATES SECTION
I01      = 0x980B ;I/O buffer bits 0-7
I02      = 0x980A ;I/O buffer bits 8-15
CMD      = 0x980F ;command buffer
Write_PR1 = 0xC1  ;write to R1 preregister command
Write_PR3 = 0xC3  ;write to R3 preregister command
Write_PR4 = 0xC4  ;write to R4 preregister command
        .AREA   Jogging (ABS)
        .MODULE Jogging
        .ORG    0x1040
;load R1 to set the low stepping rate
ldx #I02          ;point X at the I/O buffer
ldd #I00          ;load D with desired F1 stepping rate
stab 1,x         ;write the LSB
staa 0,x         ;write the MSB
ldx #CMD          ;point to the command buffer
ldaa #Write_PR1  ;write the 'write into pre-reg 1'
staa 0,x         ;command to PCL3013
;load R4 to set the multiplier
ldx #I02          ;point X at the I/O buffer
ldd #0x012B      ;load D to get the desired multiplier
stab 1,x         ;write the LSB
staa 0,x         ;write the MSB
ldx #CMD          ;point to the command buffer
ldaa #Write_PR4  ;write the 'write into pre-reg 4'
staa 0,x         ;command to PCL3013
;load R1 to set the high stepping rate
ldx #I02          ;point X at the I/O buffer
ldd #400         ;load D to get desired FH stepping rate
stab 1,x         ;write the LSB
staa 0,x         ;write the MSB
ldx #CMD          ;point to the command buffer
ldaa #Write_PR2  ;write the 'write into pre-reg 2'
jmp 0xC000       ;go back to the monitor program
```

WHAT ELSE?

Again, the answer is “a lot.” There were a number of features I didn’t test, but I do want to tell you about them. There are several modes involving the origin: origin-return mode 1 and 2, origin-escape mode, and origin-search mode. To test them, you need a stage with limit switches and home switches. This unit doesn’t seem to home the stage to an encoder marker pulse, which is the most accurate method, I believe.

A zero-return mode lets the motor return to zero without writing a zero in R0—similar to a go-home command.

Here’s a great feature: another set of encoder inputs in addition to the manual pulser inputs I referred to earlier. This encoder would likely be mounted directly to the motor.

This feature allows a comparison of the number of pulses sent to the motor and the number of pulses produced by the encoder. If a preset deviation loaded into R8 (environmental-condition register 3) is exceeded, a hardware interrupt is produced and pulses to the motor are stopped.

The one-pulse output mode is self-explanatory. It may be useful when some condition at the I/O port is monitored by the host and then used as the basis for sending commands to the PCL3013 to repetitively move by a single step.

This unit can drive a servo amplifier, too, but it has to be the type that expects a pulse train as input. There are several features and inputs associated with this feature that I was unable to test without such an amplifier. But certainly this is yet another indication of the flexibility of this part.

MORE THAN ENOUGH

Although I didn’t discuss all the features here, you’ve now been introduced to most of them. My aim wasn’t to provide a tutorial on how to use this part but rather to give you a solid indication of its capabilities. I hope I’ve accomplished that.

It takes awhile to come up to speed on this unit because of its complexity, but it’s worth it if you need a leading-edge motor controller. ☛

Gordon Dick is an instructor in electronics technology at the Northern Alberta Institute of Technology, Edmonton, Alberta, Canada. He is a member of the American Institute of Motion Engineers and is the first Canadian to obtain the Certified Motion Control Specialist (CMCS) designation. You may reach Gordon at gordond@nait.ab.ca.

REFERENCE

Nippon Pulse Motor Co., *Programmable Single-Chip High-Speed Pulse Generator*, User Manual for PCL3013.

SOURCES

EasyTrax
AP Circuits Ltd,
(403) 250-3406
www.apcircuits.com

PCL3013
Kollmorgen Corp.
(800) 77-SERVO
Fax: (540) 731-0847
www.kollmorgen.com

DEPARTMENTS

68

MicroSeries

76

From the Bench

80

Silicon Update

MICRO SERIES

Joe DiBartolomeo

TPU

A Coprocessor for Timing Functions

Part
1
of
4

Timing is everything, more so for embedded systems. So we don't lag behind, Joe provides us with the basics of the timer/counter functions found on various popular microcontrollers before introducing the time processor unit.

W

hoever coined the phrase "timing is everything" wasn't referring to embedded systems, but for embedded-system designers, no truer words have been spoken. The control of timing and counting functions is a basic requirement for most embedded systems.

Because timer and counter functionality is found in every microprocessor and microcontroller, you'd expect to find a great variety of timer and counter implementations. But, these functions have become standardized around a few basic configurations. This series deals with one timer/counter module, a time processor unit (TPU) that differs significantly from the pseudo-standard.

Figure 1a shows the basic implementation of the timer/counter function. A timer/counter register (TCR) that can be loaded by the micro is incremented (or decremented) based on an event.

In the counter mode, TCR is incremented every time there's a transition on the external pin. In the timer mode, TCR is incremented by the external or internal clock source. An interrupt request is issued when TCR rolls over.

The TPU is not like the standard timer/counter found on most micros. Its architecture includes a microengine that runs TPU microcode and also includes an execution unit, a register

set, and a bus structure, essentially making the TPU a coprocessor.

The CPU sets up the TPU to perform a timing function. The TPU then runs the microcode associated with the CPU-requested timing function. Once set up, the TPU runs autonomously, greatly increasing the micro's throughput.

Motorola, which provides the TPU on many of its micros, also provides preprogrammed, canned functions that simplify complex timing tasks (see Table 1). These functions are stored in the micro's ROM, mask sets A or G in the 68332.

The 68332 TPU has 16 channels that can independently run any of the canned functions listed in Table 1. Several complex timing functions exist that require TPU channels to be linked together.

The most intriguing TPU feature is that it can run user microcode. The programmer has control over TPU resources, which give the embedded-system designer a lot of flexibility.

Tasks that are difficult to implement with the standard timer/counter are easily accomplished with the TPU. After all, the TPU is a coprocessor and should be thought of as such. Why limit its use to timing functions?

TIMER/COUNTER BASICS

Before I get into the TPU's details, let's look at how the timer/counter function is implemented on several popular microcontrollers—the 68HC705J1A, the 8051, and its derivative, the 80552.

The 68HC705J1A has a three-stage timer/counter, shown in figure 1b. At the heart of the timer is an 8-bit ripple counter, TCR, that's driven by a di-

vided-down system clock known as an E-clock. After 1024 E-clocks, the timer overflows and sets the TOF bit, which the micro can pole or enable an overflow interrupt.

The output of the 8-bit ripple counter feeds the input of a 7-bit ripple counter, and any of the last four bits of the ripple counter can be set to generate an input based on the value of bits RT0 and RT1. This feature permits much finer timing than would be expected of a ripple counter. The timer's final stage is a computer-operating-properly (COP) function or watchdog.

A more advanced timer/counter is found on 8051 microprocessors. The 8051's two timer/counter registers, T0 and T1, are controlled by the timer/counter mode control register (TMOD) and the timer/counter control register (TCON), as Figure 2a shows.

TMOD gates the timer/counter, selects the counter or timer function, and sets operating modes (i.e., 16- or 8-bit reload). TCON has interrupt flags for the timers TF1 and TF0, which are set by hardware on overflow and cleared by hardware when the interrupt is serviced. TCON also sets interrupt parameters and has two bits that run or stop the timer/counters.

In the timer function, the timer register is incremented every machine cycle. Because the 8051 has 12 clocks per machine cycle, the timer register is updated at $1/12$ of the clock frequency.

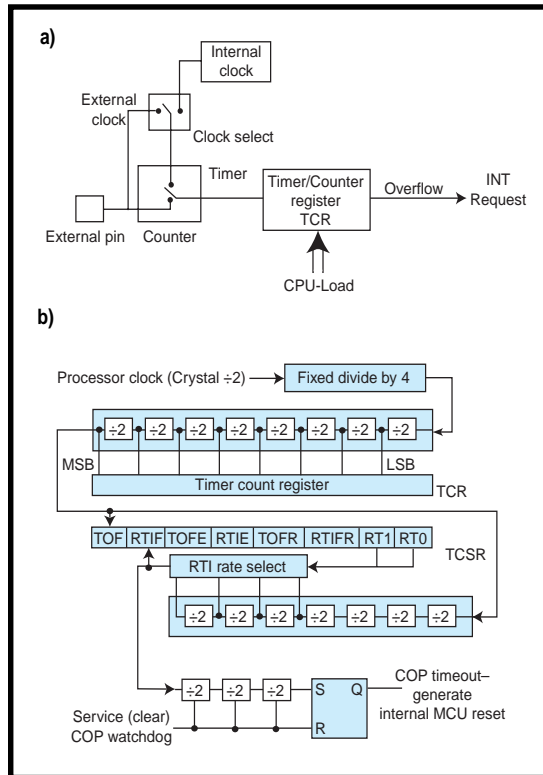


Figure 1a—This timer/counter configuration is the basic architecture for most timer/counters found in microprocessors and microcontrollers. b—The timer/counter found on the Motorola 68HC705J1A consists of a 15-bit ripple counter and a COP watchdog.

Of the timer's four modes of operation, modes one and two are the most common. The 16-bit timer register in mode one issues an interrupt and sets the TF flag when the count rolls over from all 1s to all 0s. If the timer interrupt is enabled, the CPU is interrupted. In mode two, the timer register is 8 bits (TL) and on overflow it generates an interrupt request, sets the TF flag, and reloads TL with the content of TH.

In the counter function, the T0 (T1) register is incremented whenever a 1 to 0 transition occurs on the external pin. It takes two machine cycles to recognize

a 1-to-0 transition, so the maximum count rate is $1/24$ the clock frequency.

One of the most common enhancements to basic timer/counter architecture is the addition of capture and compare registers (see Figure 2b). The 80552 is one microcontroller that contains several capture and compare registers.

When a transition occurs on the capture

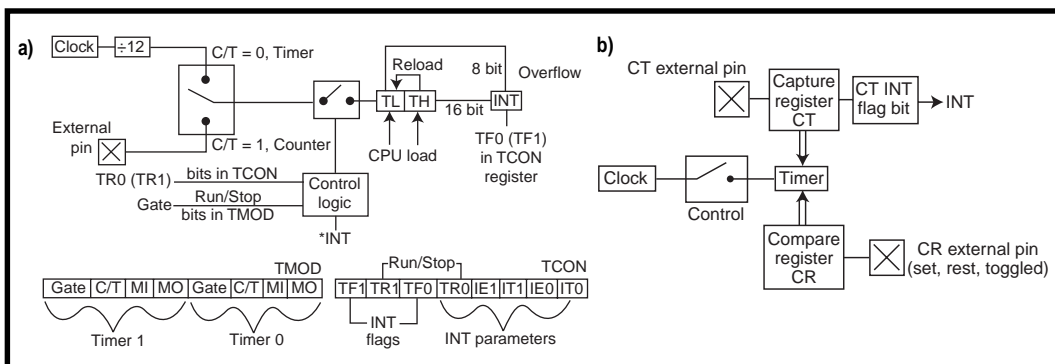


Figure 2a—Here's a block diagram of the timer/counter found on 8051 microcontrollers along with its control register bit fields. b—The addition of capture/compare registers greatly increases the functionality of the basic 8051 timer/counter. This block diagram shows the capture/compare register setup from the 80552.

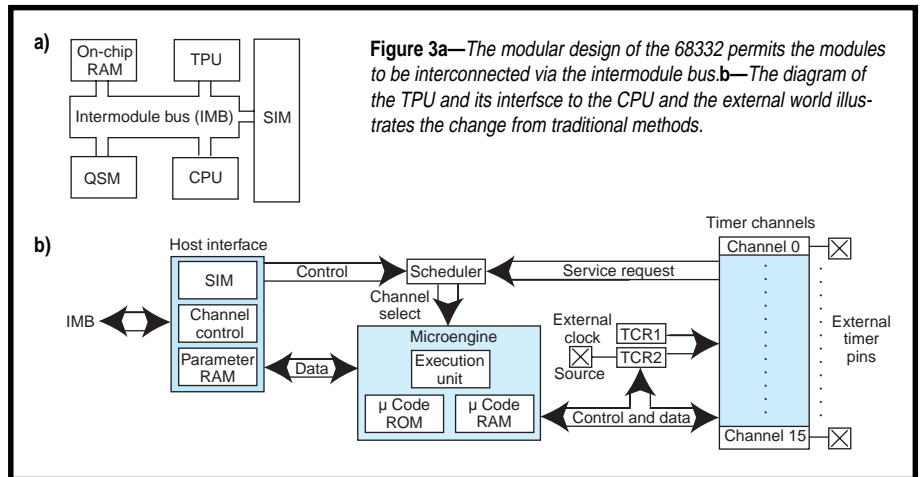


Figure 3a—The modular design of the 68332 permits the modules to be interconnected via the intermodule bus. **b**—The diagram of the TPU and its interface to the CPU and the external world illustrates the change from traditional methods.

register's input pin, the contents of the timer are captured. A compare register sets, resets or toggles the compare register's output pin whenever the content of the timer and compare registers match. Capture and compare registers add much functionality and flexibility to the basic timer/counter.

Say you want to measure the frequency of a repetitive signal using an 8051. The task looks simple, and it is, except that it requires a great deal of microcontroller resources. One timer/counter captures transitions of the incoming signal and the other measures time, so both 8051 timers are used.

Name	Code	Description
Mask Set A		
DIO (Discrete I/O)	\$8	Allows the TPU channel pin to be used as a digital I/O
ITC (Input Transition Counter)	\$A	Captures the value of a specified timer/counter on a (or a specified number of) transition(s)
OC (Output Capture)	\$E	Generates a single transition, single pulse, or square wave on the channel pin
PMA/PMM (Period Measurement With Additional/Missing Transition Detection)	\$B	PMA measures the period of an input signal, while PMM is a special 23-bit period measurement that indicates a missing transition (i.e., missing tooth on sensing wheel)
PPWA (Period PW Accumulator)	\$F	Accumulates using a 16- or 24-bit sum, either the period or the pulse width of an input signal
PSP (Position-Synchronized Pulse)	\$C	Generates pulses of variable length based on a reference timer/counter
PWM (PW Modulation)	\$9	Generates a PWM signal on the channel output pin
QDEC (Quadrature Decode)	\$C	Passes the CPU position and direction data by decoding two out-of-phase signals. Requires two adjacent TPU channels
SM (Stepper Motor)	\$A	Generates PWM output that can be synchronized to PWM signals running on other channels. For signal channel, use PWM function
Mask Set G		
COMM (Multiphase Motor Commutation)	\$9	Generates the phase commutation signal for a variety of brushless DC motors
FQD (Fast Quadrature Decode)	\$6	Passes the CPU position and direction data by decoding two out-of-phase signals. Requires two adjacent TPU channels
FQM (Frequency Measurement)	\$C	Counts number of input pulse on TPU pin for a user-defined time
HALLD (Hall Effect Decode)	\$8	Decodes signals for Hall-effect sensor. This function is primarily used with the COMM function in brushless motor apps
MCPWM (Multichannel PWM)	\$5	Uses externally gated multiple channels to generate complex PWM signals
NITC (New Input Capture/Transition Counter)	\$A	Upon occurrence of a (or specified number of) transition transitions, captures the value of a specified timer/counter
PTA (Programmable Time Accumulator)	\$F	Accumulates a 32-bit sum of the high/low or total time of an incoming pulse
QOM (Queue Output Match)	\$E	Generates single or multiple match events based on a user set offset table
TSM (Table Stepper Motor)	\$D	Provide acceleration/deceleration control of stepper motor with programmable step rates, based on user set tables
UART (Universal Asynchronous Receiver/Transmitter)	\$B	Provide standard UART function; upto eight simultaneous UART channels running as 9600 bps can be implemented

Table 1—These preprogrammed (canned) functions are provided on the Motorola 68332 (mask sets A and G). The 16 channels of the TPU can independently run any of these functions. Several functions require more than one TPU channel.

Or, you can use only one timer and capture just the transitions on the 8051 interrupt lines. When the first transition occurs, the interrupt routine starts the timer. The interrupt routine then stops one timer when the second transition occurs. However, this setup requires you to set the highest priority to this interrupt, otherwise the timer will start too late, run too long, or both.

Starting and stopping the timer also means that it can't be used for other functions. Therefore, the simple task of measuring frequency causes severe programming constraints. And, if the duty cycle was required, you'd most likely need external circuitry.

On the other hand, using two capture registers to measure the frequency of an incoming signal is quite simple. The signal to be measured is fed to the external pins of both capture registers, which are set to capture the same transition edge.

Initially, the first capture register is disabled, so when the first transition occurs, the second register captures the timer value and interrupts the CPU. The CPU then disables the second register and enables the first capture register. When the next transition occurs the second register captures the timer value, and it's a simple matter of subtraction to get the frequency of the incoming signal.

Because the capture registers hold their values, the CPU has some time before it must process the captured times. Also, the timer is free running and available for other functions. If a third capture register is used to detect the opposite edge transition, then the pulse width and duty cycle can easily be found.

The point here isn't that it's better to have capture registers, Rather, timing

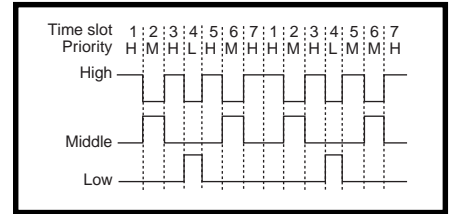


Figure 4—The scheduler priority scheme ensures that the high-priority channels, H, get the greatest access to the execution unit, while ensuring that low-priority channels, L, do not get locked out.

functions use a great deal of system resources, and when complex timing tasks are required, even capture/compare registers aren't enough.

This is the genesis for the TPU module. It enables extremely complex timing functions to be handled simply and with little CPU overhead.

TIME PROCESSOR UNIT

For this series, I'm using Motorola's 68322, a 32-bit microcontroller with a modular design. The 68322 modules are essentially stand-alone subsystems as you see in figure 3a. The queued serial module (QSM) handles communications such as RS-232, three-wire serial SPI, and high-speed QSPI.

Systems integration modules (SIM) handle chip selects, clocks, bus sizing, and the like, reducing the amount of external glue logic required. The TPU, of course, handles timing functions. The modules are interconnected via the intermodule bus (IMB) and run semiautonomously to increase system throughput.

TPU HARDWARE MODEL

As you see from the TPU depicted in figure 3b, it's quite a departure from the timers/counters I've just discussed.

The heart of the TPU—the microengine—executes the TPU microcode. In emulation mode, the microengine executes user microcode stored in onboard RAM. In nonemulation mode, (for lack of a better term), the microengine executes the microcode stored in the 332's ROM (i.e., the functions listed in Table 1).

Once initialized, the microengine runs with no CPU intervention. Parameter pass-

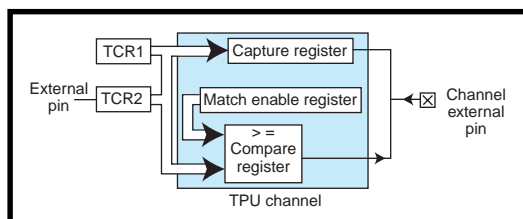


Figure 5—Each TPU channel has three registers (capture, match, and compare) along with an external pin. The TPU channel can be synchronized to one of the internal counters or an external clock.

ing between the CPU and TPU is accomplished via dual-port (i.e., parameter) RAM. The microengine also has registers that enable the TPU to perform basic arithmetic, which isn't possible on the standard timer/counter. I discuss the microengine in more detail later in this series.

The scheduler determines which of the TPU's 16 channels is serviced by the microengine, based on the priority assigned to each channel by the CPU. The priority levels are H for high, M for middle, and L for low. If two channels have the same priority level, the lower channel number is serviced first.

As you see in Figure 4, the scheduler's cycle is divided into seven time slots—four for high priority, two for middle-priority, and one for low-priority channels. This way, high-priority channels get the services they require and low priority channels are not locked out. I'll discuss scheduling more when I go over microcoding.

The timer channels can be synchronized to either of the two internal timers, TCR1 and TCR2. Or, you can synchronize them to an external clock source on the external clock pin. The TPU's 16 identical channels can be used individually or linked together.

Each channel contains a 16-bit capture register, a 16-bit greater-than or equal-to compare register, a 16-bit match-enable register, and six words of parameter RAM (see Figure 5). The exceptions are channels 14 and 15, which each have eight words of parameter RAM to use when passing parameters to and from the CPU.

TPU PROGRAMMING MODEL

As I mentioned, the TPU can run factory-programmed microcode out of ROM in nonemulation mode, or user microcode out of onboard RAM in emulation mode. I'll leave emulation mode for later, but I want to cover nonemulation mode now.

In nonemulation mode, the TPU can perform any of the canned functions on any or all of its channels. Table 1 lists the functions that come preprogrammed on the '332. Although other processors have their own mask sets, Table 1 shows the two mask sets, A and G, available on the 68332.

TPU programming is straightforward, but some care is required because several registers have to be set up and some registers are shared by several channels. Figure 6 shows the TPU register map and the registers' bit fields.

		15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0								
Module config. register	TMCR	Stop	TCR1P Prescaler	TCR2P Prescaler	EMU	2TCG	STS	SUPV	PSCK	0	0	Interrupt arbitration ID													
	TICR	0	0	0	0	0	Channel interrupt request level	0	0	0	0	Channel base vector		0	0	0	0								
Interrupt enable register	CIER	Chan 15	Chan 14	Chan 13	Chan 12	Chan 11	Chan 10	Chan 9	Chan 8	Chan 7	Chan 6	Chan 5	Chan 4	Chan 3	Chan 2	Chan 1	Chan 0								
Channel function select registers	0	Channel 15				Channel 14				Channel 13				Channel 12											
	1	Channel 11				Channel 10				Channel 9				Channel 8											
	2	Channel 7				Channel 6				Channel 5				Channel 4											
	3	Channel 3				Channel 2				Channel 1				Channel 0											
Host sequence registers	0	Channel 15			Channel 14			Channel 13			Channel 12			Channel 11			Channel 10			Channel 9			Channel 8		
	1	Channel 7			Channel 6			Channel 5			Channel 4			Channel 3			Channel 2			Channel 1			Channel 0		
Host service registers	0	Channel 15			Channel 14			Channel 13			Channel 12			Channel 11			Channel 10			Channel 9			Channel 8		
	1	Channel 7			Channel 6			Channel 5			Channel 4			Channel 3			Channel 2			Channel 1			Channel 0		
Channel priority registers	0	Channel 15			Channel 14			Channel 13			Channel 12			Channel 11			Channel 10			Channel 9			Channel 8		
	1	Channel 7			Channel 6			Channel 5			Channel 4			Channel 3			Channel 2			Channel 1			Channel 0		
Interrupt status registers	CISR	Chan 15	Chan 14	Chan 13	Chan 12	Chan 11	Chan 10	Chan 9	Chan 8	Chan 7	Chan 6	Chan 5	Chan 4	Chan 3	Chan 2	Chan 1	Chan 0								

Figure 6—Notice that the bit fields are in registers shared by several channels. When programming a bit field, take care not to inadvertently alter other bit fields.

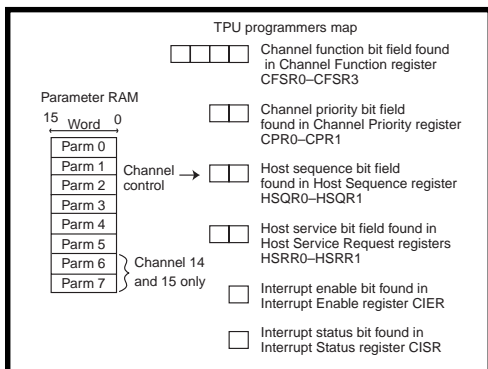


Figure 7—The programmer must set up every bit field and parameter RAM prior to running one of the canned functions.

First, you need to globally initialize the TPU, which involves writing to the module configuration register (TMCR) and the interrupt configuration register (TICR). The TMCR sets up clocks and prescaling and sets arbitration levels for accessing the IMB. Most importantly, it selects whether the TPU is in emulation mode.

The TICR sets the interrupt request level, which is the same for all TPU channels, and arbitrates the interrupts between 68332 submodules on the IMB. The TICR also sets the base address for the interrupt vector. Global initialization must be performed for both modes of TPU operation.

The individual channels must be initialized through a seven-step process. The channels can be initialized in any order, and Figure 7 shows the programming map of a timer channel.

First, disable the channel by clearing its two-bit priority field in the channel priority register CPR0 or CPR1. Next, load the channel-function code into the appropriate channel-function register CFSR0-CFSR3 (see Table 1 for function codes). After that, load any parameters required by the function into the channel's parameter RAM.

The host sequence bits are set in the host sequence register, HSQR0 or HSQR1. These bits help specify the operation of the time function, and their meaning depends on the function selected.

The host service requests bits in the host service request register, HSR0 or HSR1, are set up next. They determine the type of service the CPU is requesting (e.g., initialization, run). Their meaning depends on the time

function selected. The CPU requests service by writing any one of three nonzero values into the appropriate HSR two-bit field. Only the TPU can clear this bit field, and it does so when it completes the requested service.

The CPU can monitor the HSR two-bit field to determine when the TPU completes the service, but it's better to enable the interrupt on the channel. For the CPU to interrupt when the TPU completes a requested service, the appropriate bit in the channel interrupt enable register (CISR) must be set.

The channel priority bits are set as high (11), middle (10), or low (01). A nonzero value turns the channel on; a value of 00 turns the channel off.

I know the 16-channel setup is a bit tedious, but using a template for new applications makes the job go faster. The up-front effort is worth it when you see how the TPU handles complex timing and counting tasks with ease.

Next time, I'll look at how to handle a quadrature encoder, DC motor speed control, and position measurements, all using the canned TPU functions. ☒

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

REFERENCES

- T. Harman, *The Motorola MC-68332 Microcontroller*, Prentice-Hall, Englewood Cliffs, NJ, 1991.
- Motorola, *Time Processor Unit Reference Manual*, TPURM/AD, 1993.
- Motorola, *Central Processor Unit*, 1990.
- J. Sibigroth, *Understanding Small Microcontrollers*, Motorola, 1995.

SOURCES

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

FROM THE BENCH

Jeff Bachiochi

Can You Feel the Beat?

temperature-compensated crystal oscillator (TCXO), the DS32KHz. Touting accuracy of ± 1 min. per year $0-40^{\circ}\text{C}$ and ± 4 min. a year $-40-85^{\circ}\text{C}$, the TCXO requires no calibration. Operating voltage is 2.7–5.5 V, making it useful in those 3-V projects.

The device has a separate backup-battery input so it continues running even when system power is removed. Current consumption at 3 V is typically $1\ \mu\text{A}$. The DS32KHz is larger than your typical 32-kHz crystal and comes in a half-inch-square surface-mount (ball grid array style) package.

Don't go looking for this TCXO in the 65-cent bin at your local Radio Shack. You'll need to fork out about ten times that amount. But when you need higher accuracy than you can typically get, the Dallas part is golden.

Why am I leading off with this? Last month, I hinted at some gotchas I wasn't able to see until I actually ran code on Atmel's AVR part that I designed with. I wondered how many of you would catch what I missed.

Last month's project used an AT-90S2323 with a 32-kHz crystal. The datasheet states 0–10 MHz. Although it will operate with slow oscillator inputs, the internal amplifier won't drive a 32-kHz crystal (the Atmel apps engineers assured me some newer parts will include the proper amplifier).

So, where does that leave a design that works well on paper, simulates fine in the simulator, but doesn't function in reality?

EENIE, MEENIE...

For highly accurate timing, the DS32KHz TCXO could be used as the oscillator input to the AT90S2323 micro, thus assuring low power con-



Yeah, you can use a crystal for your time base. But, how accurate is it? Standard tolerances are ± 20 ppm at 25°C . Twenty parts per million seems pretty good—or is it?

Well, $60\ \text{s} \times 60\ \text{min.} \times 24\ \text{h} \times 365$ days is about 31.5 million seconds a year. If the crystal can be off by 20 ppm, that's 20×31.5 , which is 630 s or 10.5 min. (about 1 min. a month)!

In an NEMA box exposed to the sun or freezing temperatures, it can be two or three times as inaccurate. In an environment above the full industrial temperature $-40-85^{\circ}\text{C}$ ($-40-185^{\circ}\text{F}$), it ends up being off by over an hour a year—hardly accurate, and that's based on finding a manufacturer that produces a crystal that meets industrial specs.

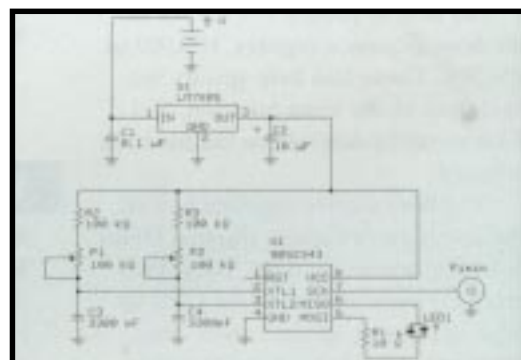
Time is now on your side. Dallas Semiconductor has released a new



With a son taking drum lessons,

it's no surprise that Jeff wants to create a metronome. And small wonder that he needs an accurate time base for it! Listen in to learn how a new crystal oscillator helps out.

Figure 1—When using the internal oscillator of the '2343, the crystal inputs can be used as additional I/O. But, changing the direction of an I/O bit alternately shorts out the attached capacitor C3/4 (outputting 0) and allows it to charge (configure pin as input). The charge time, adjustable via the potentiometer P1/2, is based on the number of times the pin is read before it becomes a logic 1.



Listing 1—This test program helped me determine the timing parameter for various external capacitor and resistor combinations.

```
.device AT90S2343 ;Prohibits use of nonimplemented instructions
.include "2343def.inc"
;* Global Register Variables
;* Code
rjmp RESET          ;Reset Vector
rjmp INTO           ;External Interrupt Vector
rjmp TMRO_OVF      ;Timer 0 Overflow Vector
;* Main Program Register Variables
.def TEMP =r16
.def CNTR =r17
;* Code
RESET: ldi  TEMP,$DF          ;value for stack pointer
      out  SPL,TEMP          ;put it there
      ldi  TEMP,$00          ;value for PORTB xxx00000
      out  PORTB,TEMP        ;put it there (no pullups on inputs)
      ldi  TEMP,$0F          ;value for PORTB direction xxxI0000
      out  DDRB,TEMP        ;put it there
      ldi  TEMP,$01          ;value for no prescaler
      out  TCCR0,TEMP        ;put it there
BEGIN: sbi  DDRB,PB4          ;config as output (to short out cap)
      clr  TEMP
      out  TCNT0,TEMP        ;start at zero
      in   TEMP,TIFR
      cbr  TEMP,TOV0
      out  TIFR,TEMP
      cbi  PORTB,PB3
      cbi  DDRB,PB4          ;config as input (to allow cap to charge)
CHECK: sbis  PINB,PB4         ;skip next if PB4 (charging crossed high
                              ;threshold)
      rjmp CHECK             ;input high so keep timing
      in   TEMP,TCNT0        ;save the timer0 value
      in   CNTR,TIFR
      sbrc CNTR,TOV0         ;skip next if no overflow in timer0
      sbi  PORTB,PB3         ;set PB3 (overflow indicator)
      rjmp HNIB              ;go on
CHECK1: cbi  PORTB,PB3        ;clear PB3 (no overflow)
HNIB:   mov  CNTR,TEMP        ;get saved timer value
      swap CNTR              ;get upper nybble
      andi CNTR,$0F          ;mask off upper (was lower)
HNIB1: breq  LNIB            ;branch if CNTR is zero
      sbi  PORTB,PB0        ;else toggle PB0 (upper nybble count)
      cbi  PORTB,PB0
      dec  CNTR              ;reduce the count
      rjmp HNIB1            ;and check again
LNIB:   mov  CNTR,TEMP        ;get saved timer value again
      andi CNTR,$0F          ;this time get rid of upper nybble
LNIB1: breq  DONE            ;branch if CNTR is zero
      sbi  PORTB,PB1        ;else toggle PB1 (lower nybble count)
      cbi  PORTB,PB1
      dec  CNTR              ;reduce the count
      rjmp LNIB1            ;and check again
DONE:   rjmp BEGIN           ;that's all, so start over
TMRO_OVF: ;not being used
      reti
```

sumption and no software changes. But, the more cost-effective choice for last month's tiny timebase might be a software change that enables you to use a 4.194304-MHz crystal.

This frequency crystal divides down to the same half-second tics that the 32-kHz crystal would have if the internal oscillator had enough drive. With the higher crystal, you can use a divide-by-1024 prescaler to get a timer clock of 4096 Hz. The 8-bit timer rolls over (interrupt) at a rate of 16 Hz. Now, every eighth rollover is 0.5 s—the same time base used in last month's code.

THE BEAT GOES ON

Now that we corrected the design error from last month, let's look at a technique for using a digital input as an analog input with a potentiometer. The most recent project I'm working on is for my youngest son, Kristafer. He started music lessons this month and is in need of a metronome.

The fact that he chose the drums underscores this need. After all, if the percussion section can't keep the beat, the fat lady will never want to sing.

There are two input controls to the metronome. The first is a tempo or beats/minute control, and the second is a mode or beats/measure control. The mode control allows counting by one, two, three, four, five, or six.

These counts fire off a piezo device that makes a beep, and lights an LED, providing both audible and visible indicators. The first count always uses a different tone and color (via a multicolored LED) to indicate the downbeat or first beat in a measure. The tempo control adjusts the tempo of the beats from largo (40 beats/min.) to presto (208 beats/min.).

A/D inputs would make this task easy. The typical connection has the ends of a potentiometer across V_{CC} and ground with the wiper connecting to an analog input of the micro. Assuming the ADC is referenced to V_{CC} , it would convert the wiper's voltage into a value equal to relative position of the pot's wiper.

Because the ADC's converted values are steps, what you have is a pot's analog position converted into a number of (switch) positions equal to the reso-

lution (or number of steps) of the ADC. Usually, the more steps (i.e., the higher the resolution) the harder it is to position the wiper to a particular step.

Without an ADC, this situation becomes more difficult. Since we can only read two input voltages (high and low) via a digital input, a different approach must be used to determine the potentiometer's wiper's position.

One way is to use the time it takes to charge a capacitor from zero (a low input) up to the input's high threshold switching level (V_{ih}), usually about $0.6 V_{CC}$ or 3 V at a V_{CC} of 5 V. Figure 1 shows how this is done. One end of the capacitor connects to ground while the other end goes to an I/O pin.

The pot used as a variable resistor connects between the I/O pin and a series resistor to V_{CC} . The series resistor prevents the I/O pin from shorting to V_{CC} when the pot is at minimum resistance (short circuit).

TESTING 1, 2, 3, 4

Because of all the unknowns here (especially the exact voltage necessary

to switch from logic low to high), determining what capacitor and resistor to use isn't simply mathematics. But, I need to start somewhere.

The shortest overflow of timer0 using the internal 1-MHz R/C oscillator will be about 256 μ s. So, I'll try a R/C time constant of that length using a 100 k Ω potentiometer. The capacitor value needed will be about 0.002 μ F, or:

$$C = \frac{t}{R} \left(\frac{2.56 \times 10^{-4}}{1 \times 10^5} \right)$$

I always write short programs to test certain operations that I don't feel comfortable using for the first time. This is a good case in point. Listing 1 shows an overview of the routine.

To start, set the data on the I/O bit connected to the external R/C circuit to 0. Begin with this pin configured as an output. The 0 data output on this pin shorts the capacitor to ground.

Now, clear and start the timer and reconfigure the I/O pin as an input. The voltage across the capacitor will

begin to rise as the capacitor becomes charged through the potentiometer.

When the rising voltage crosses the threshold for a logic 1, the digital input will be seen as high. The code jumps out of its tight loop (waiting for this to happen), immediately reads the timer's count, and checks the overflow flag. Now, the timer count is relative to the time it took the input voltage to charge up to the logic high threshold.

Adjusting the potentiometer varies the charging time (and timer count). I need a way to see what the count is. If the device had a hardware UART, I probably would have sent the ASCII value out a port. But there's no hardware UART on the AT90S2343, and I don't want to spend time debugging a software UART routine.

So, I used some output bits for a quick indication. I toggle PB0 once for each count of the timer's upper nibble value and then PB1 once for each count of the lower nibble's value. A third output bit mirrors the timer overflow flag.

The scope traces in Photo 1 give you an idea of how this quick routine



works. After playing around with capacitor values I had in my parts cabinet, I chose a couple of 0.0033 μF caps with 100-k Ω pots. These values gave me almost a full count of 03-F6h.

For the input that's being used as a switch, I need six switch positions. I use the upper nibble values of 00-3Fh as 1, 40-5Fh as 2, 60-7Fh as 3, and so on, with anything over C0h as 6. For the tempo input, I use the count directly. To keep the math minimal, the tempo value is not in counts/min. but the reciprocal, counts \times period (a period equals 10 ms).

A ONE, AND A TWO...

Getting this metronome's heart beating takes a single interrupt. The timer0 interrupt is based on a noprescaled 8-bit rollover. Every 256 μs , the timer0 interrupt flag redirects the execution to an interrupt routine.

The timer0 interrupt routine counts interrupts up to 39. If 39 isn't reached, the routine exits with a RETI (to re-enable the global interrupt on exiting).

When it reaches 39, the interrupt branches to a second loop that increments TCNT up to the value of TEMPO (the periods/beat counter). This branch is ended with a RET (not RETI) and leaves the global interrupt disabled (important point, as you'll see later).

The timer0 interrupt ends up incrementing TCNT every $256 \mu\text{s} \times 39$, which is 9.984 ms (~ 10 ms).

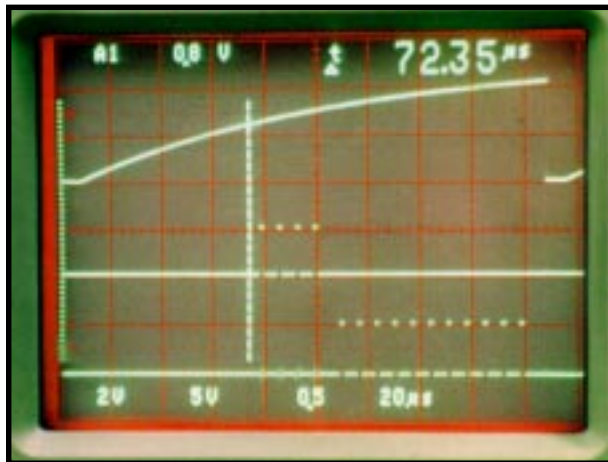


Photo 1—C3 changes to the point where PB4 sees a logic 1 (72.35 μs). The timer count is shown as pulses (value of upper nibble) on PB0 and (value of lower nibble) on PB1.

Between interrupts the main execution does very little. When TCNT is 1, execution does a read of the tempo potentiometer and updates TEMPO if needed.

TEMPO, the interrupt's outer loop reload count, determines how many 10-ms periods will be counted until it declares that one beat has occurred and clears TCNT. To prevent exiting this branch and doing the same thing over again, a BRIE (branch if global interrupts are enabled) is executed.

Remember how the timer0 interrupt exited the inner loop (count up to 39) with a RETI. I use BRIE to branch to itself, which holds execution until 39 is reached and the outer loop increments TCNT, finishing the interrupt routine with a RET.

When TCNT is 2, the mode pot is read. The mode pot uses a lookup table to divide its position into the values 1-6. Changes to the mode pot (beats/measure)

clear the mode count MCNT immediately. MCNT is then incremented, and BRIE is used to hold execution again.

When TCNT is 3, the mode count determines which LED is enabled. The red LED is enabled for all but the first count of MCNT. An MCNT of 1 enables the green LED. If the mode value is 1, MCNT will not increment past 1 and every beat is a (green) downbeat.

Not much happens until TCNT reaches 10, in which case both LEDs are disabled. Notice that the bicolor LED has two leads. The color is determined by the polarity of the potential across the leads, and by placing the

LED (and series resistor) across two output bits, I can change the color by raising one and lowering the other bit.

Again, not much happens until TCNT reaches the value of TEMPO. TCNT is now cleared and the whole process repeats itself for the next beat.

POCKET PACKAGING

The metronome is housed in a small PacTec enclosure (see Photo 2) with a 9-V battery compartment and a belt clip that holds the box onto the music stand. Now if he'd only practice! ☹

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

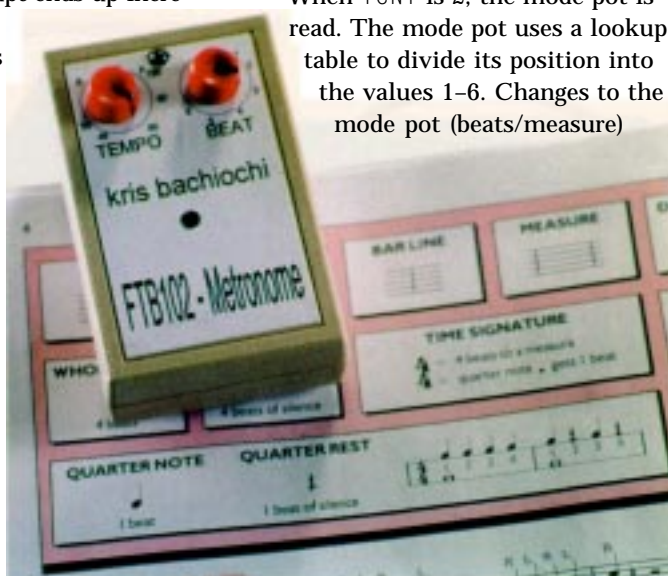


Photo 2—Kris's personalized metronome clips onto his music stand, thanks to the belt-clip option available with this PacTec enclosure. A push-button power switch located under the clip turns power on when the unit is clipped onto the music sheets.

SOURCES

DS32KHz TCXO
Dallas Semiconductor
(972) 371-4448
Fax: (972) 371-3715
www.dalsemi.com

AVR AT90S2343
Atmel
(408) 441-0311
Fax: (408) 436-4200
www.atmel.com

HM-9VB-BC
PacTec
(215) 365-8400
Fax: (215) 365-4420

Wires, Wires Everywhere

The RF Solution

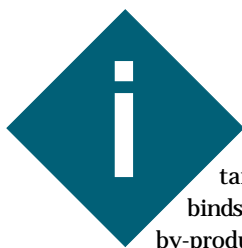


Tom's had a wire problem for years. But, with

some embedded-world technology, he may have found the fix at last. He's hoping Micrel's complete radio receiver in a single chip will bring him a cable-free future.

SILICON UPDATE

Tom Cantrell



I hate wires. The tangled web that binds is an unfortunate by-product of the modern computer age. Keyboard, mouse, CRT, speakers, printer, modem, game ports, scanner, and more all call for their own hookup, not to mention the requisite power connections.

Hope lies with reducing if not the quantity, at least the confusion, of all these cable connections. But, progress has been agonizingly slow.

For instance, USB has been on-deck for more than two years (see "Oh Say Can USB?" *INK* 74). Yet, despite liberal seeding with USB-enabled motherboards and chips, progress has gone at a snail's pace. Part of the problem was a soft-

ware driver vacuum while waiting for Windows 98, and some would argue that the air is still thin.

The other problem is sheer inertia. I recently got a new PC that came with a regular mouse and keyboard, leaving my USB ports to gather dust like most everyone else's. The only bit of hope is that I did notice a few square feet of shelf space devoted to a handful of USB gadgets at the local computer shop.

Ironically, after all the PC huffing and puffing, it may be the appearance of the Apple iMac that gives USB the push it needs. I recently saw a Mac-oriented mail-order catalog with a page full of USB gadgets (mice, trackballs, disks, printers, etc.) for this new baby.

But even as USB pokes along, the powers that be turn their attention to the next big thing in the form of IEEE 1394 (i.e., Firewire). In an article I wrote for *Computer Design* ("Firewire Getting Hot," October '97), I referred to 1394 as the "RCA jack of tomorrow." It's the holy grail of convergence, a single cable that purports to connect every A/V gadget we own.

Of course, 1394 will have to cross the same barriers as USB and then some. Besides hardware inertia and lack of software, it's being challenged by that all-too-common malady of creeping featuritis (i.e., the ink on one spec is barely dry before someone decides more tweaking is in order).

Worse yet, the fear of having their art reduced to easily copied 1s and 0s has Hollywood and their armies of lawyers involved in torturous and time-consuming negotiations over the

nitty-gritty of copy protection. Last I heard, not only will authorized 1394 gear be

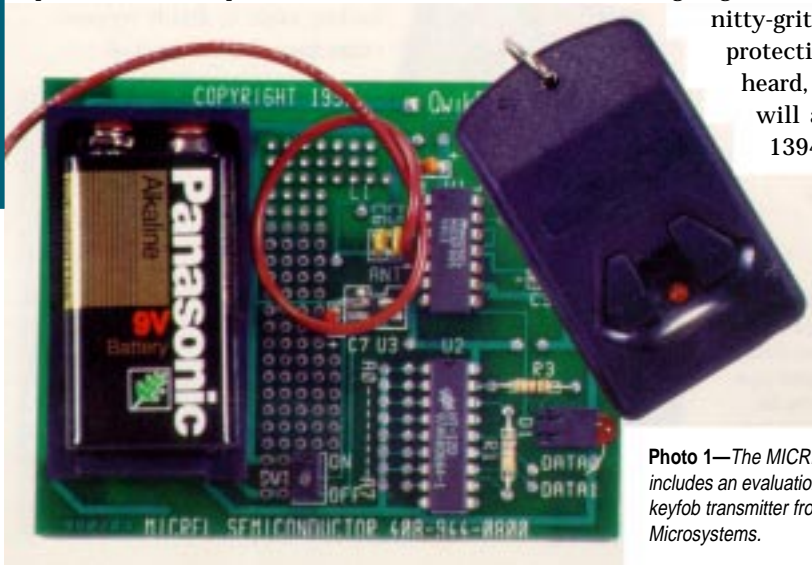


Photo 1—The MICRF001 EV kit includes an evaluation board and keyfob transmitter from Ming Microsystems.

subject to a gag rule when it comes to copying, but it'll also rat on any unauthorized gear that plugs in.

Unlike the PC world, where even the smallest innovation seems to require consensus from Redmond to Washington DC, the embedded world is blessedly agile. Wireless TV remotes, phones, car locks, and head-phones are all common and welcome additions to our daily lives.

Free from the politics, inertia, and jockeying of the PC market, embedded wireless technology proceeds at a silicon rather than human pace. A recent chip announcement from Micrel demonstrates what I mean.

MICROFUN

The MICRF001 (I don't know if Micrel intended it, but I find myself saying "microphone") is a complete radio receiver on a single chip. As stated in the datasheet, you don't have to be an "RF expert" to design it in (or, fortunately for me, to write about it).

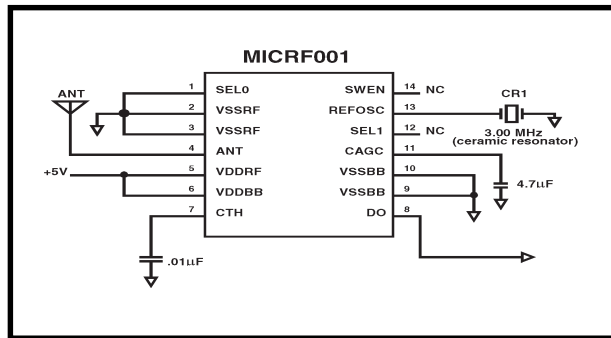
Again quoting the datasheet, the chip is a true antenna-in, data-out monolithic device. As Figure 1 shows, building a complete receiver around the MICRF001 requires as little as adding a crystal (possibly getting away with an even lower cost ceramic resonator) and two capacitors.

The specs reflect targeted applications like keyless entry, security systems, garage door openers, and so on. Operating in the 300- to

440-MHz (UHF) frequency band, the chip can typically receive data at up to 4.8 kbps over 100 m. The exact data rate depends on the RF frequency selected and one of four selections made by jumpering the SEL0 and SEL1 pins.

For instance, if the frequency is 418 MHz, datarate options are 4.8, 2.4, 1.2, and 0.6 kbps. As for range, actual results depend heavily on antenna design (more on this later) and, of course, the presence or absence of interference and ob-

Figure 1—What's it take to get on the air with the MICRF001? Little more than a clock reference (crystal, resonator, or external input) and a couple of capacitors.



stacles. Performance also depends on transmission characteristics such as the presence or absence of a preamble, minimum pulse width, and such (more on this later, too).

Figure 2 shows all the components of a classic radio receiver design on-chip. Starting at the ANT (antenna) input, the raw RF (ftx) is downconverted to a lower intermediate frequency (fif) using a mixer in conjunction with a programmable synthesizer also known as the LO (local oscillator, flo).

The downconverted data is amplified, subjected to automatic gain control (AGC), and passed to the demodulator section where it's filtered and sliced into good old 1s and 0s (i.e., baseband) for delivery out the data out (DO) pin.

I just described a classic superheterodyne (SH) receiver. The MICRF001 can work in an even simpler super-regenerative (SR, or homodyne) mode that dispenses with the need for the LO because conversion is direct to

baseband from RF without an intermediate step.

The MICRF001 works with on-off key (OOK) modulation in which the transmitter simply turns the RF carrier on and off, rather than modulating its amplitude (AM), frequency (FM), or phase (PM).

SH and SR schemes each exhibit relative advantages and disadvantages. An SR transmitter doesn't call for especially high transmit-frequency accuracy (e.g., the transmitter can use a cheap LC oscillator). So, the SR transmitter is usually only appropriate for applications where receiver frequency can be manually tuned.

In contrast, SH setups require accurate timing as well as crystals or especially accurate SAW resonators. The benefit is that you can dispense with the need for manual tuning.

Micrel has managed to combine the best of both the SH and SR worlds. For example, the device can be configured

in sweep or fixed modes by jumpering the sweep-enable (SWEN) pin appropriately. Sweep mode varies the LO symmetrically to broaden the RF bandwidth permitting operation with drift-prone LC-based transmitters.

Of course, being less selective about what's received implies more susceptibility to interference. But if there's no need to achieve accurate timing, as when a crystal-based reference already exists in the system, conventional fixed mode is OK.

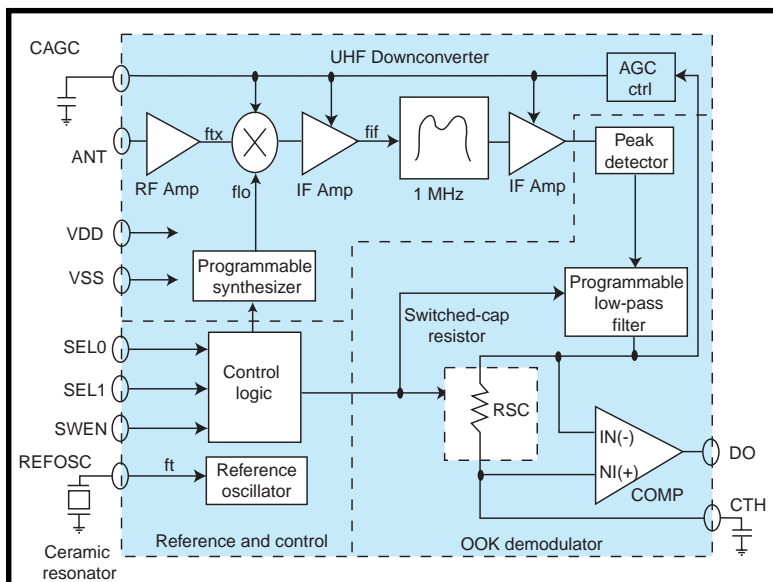


Figure 2—The MICRF001 integrates all the components needed to grab RF from an antenna connected to the ANT pin and deliver digital data out the DO pin.

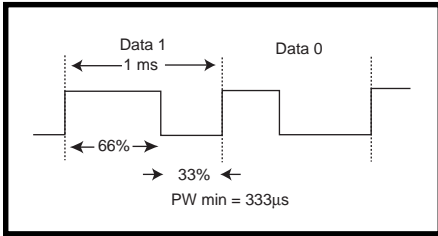


Figure 3—The way data is coded impacts performance with minimum pulse width, a key parameter. One popular way to avoid narrow pulses, which are hard to distinguish from noise, is to use a 33/66% PWM scheme.

TUNING IN

Getting on the airwaves design-wise is a relatively simple (at least conceptually) four-step process, followed by testing and tweaking to optimize range. But the devil is in the details, so let's step through the process focusing on the areas where gotchas might arise.

The first step is establishing the basic timing, which is usually a cart or horse decision. If the MICRF001 listens to an existing transmitter, that unit's frequency determines the clock required on the REFOSC input, which is stepped up internally. The datasheet shows the defining equations (e.g., how working with a 315-MHz transmitter calls for 2.4092 MHz on REFOSC).

If you have the luxury of choosing the transmitter frequency (i.e., tuned LC), you can start with a standard value (3.000 MHz) for REFOSC and work backward. This technique can save you the cost associated with a non-standard clock source and the hassle of having to deal with any fractional errors in subsequent calculations.

Another timing decision involves jumpering the SEL0 and SEL1 pins. Though the configuration inherently dictates the baseband (i.e., out the DO pin) data rate, it's not simply a matter of choosing the fastest or most convenient digital connection. Rather, you have to find a baseband filter bandwidth that depends on the minimum pulsewidth, not the data rate.

Here, for the first (but not the last) time, the issue of the data coding comes into play because it impacts the relationship between data rate and pulse width. As an extreme, NRZ coding, which represents a 1 or 0 as a corresponding level, won't work because a 0 is equivalent to silence. Instead, a

PWM type coding is required, in which each bit consists of high and low levels.

Consider typical codings like the 33/66 shown in Figure 3 and 50/50 (e.g., Manchester coding). Even though the base-band data rate may be the same, the former has a shorter minimum pulse width (33% vs. 50%) calling for a higher filter bandwidth.

Because more bandwidth lets in more noise and reduces range, 50/50 schemes are preferred and although 66/33 are OK, something like 90/10 should be avoided. The datasheet contains equations and tables to calculate the optimal SEL0 and SEL1 setting depending on transmit frequency and minimum pulse width.

IT SLICES...

The second step in getting on the airwaves is selecting the slicing time constant via the capacitor on the CTH pin. The slicer cuts the data into 1s and 0s and consists of a comparator with threshold determined by the voltage on the CTH pin.

By feeding the demodulated data through an RC low-pass filter comprising an on-chip resistor RSC and the external capacitor on CTH, the voltage is developed. The voltage represents the average voltage of the data signal over a period of time (the slicing time constant) against which the instantaneous voltage of the data signal is compared.

Consider the extremes of zero and infinite slicing time. At zero slicing

time, the voltage on CTH exactly follows the data. At infinite slicing time, the voltage on CTH remains at zero. In both cases, the comparator has nothing meaningful to compare against.

Figure 4 shows three example slicing time settings ($v(2)$, $v(3)$, and $v(4)$) and illustrates what's going on. Once again, the choice of coding and further protocol comes into play.

For instance, by sending a long preamble or repeated transmissions, a relatively long time constant (e.g., 50 ms) produces a nice even level for the comparator to work against. Of course, the downside is that it takes relatively longer to move a given amount of data.

In contrast, if there's little preamble or no repeated transmissions, a shorter time constant (5 ms) is required to get the comparator input ramped quickly. The problem is that the steep slope creates pulse-width distortion that impacts range.

Thus, the overall goal is to choose the longest time constant that is consistent with protocol and decode time constraints. The datasheet indicates that a reasonable rule of thumb is a slicing time constant equal to about five bit times.

Step three is configuring the automatic gain control (AGC) via the CAGC's pin capacitor. The idea is to center the dynamic range of the system around the local ether noise level.

Setting the attack/decay time constant with the capacitor on the CAGC

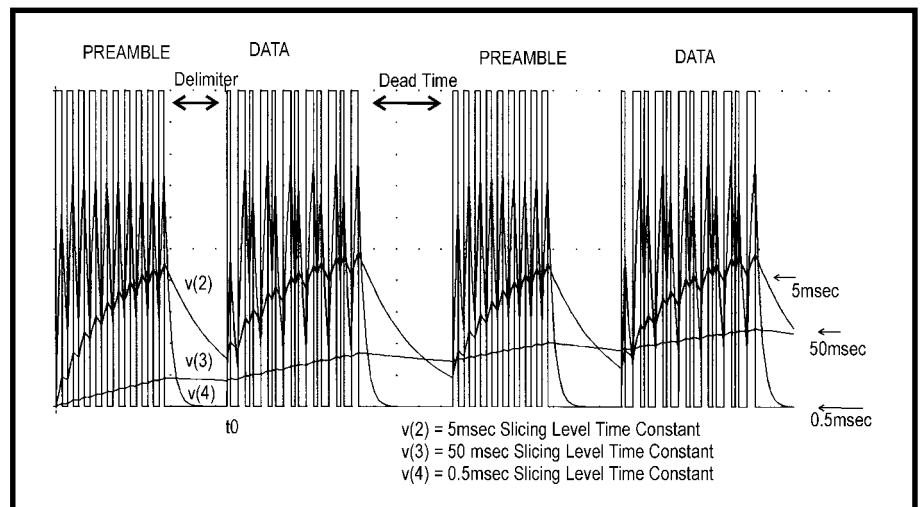


Figure 4—Configuring the optimal slicing time constant is critical and depends on both protocol (i.e., presence or absence of preamble or repeated transmissions) and throughput requirements.

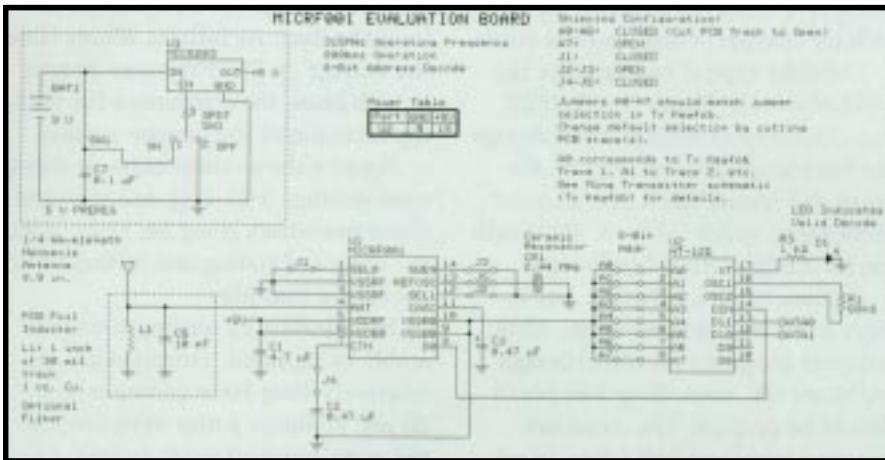


Figure 4—Configuring the optimal slicing time constant is critical and depends on both protocol (i.e., presence or absence of preamble or repeated transmissions) and throughput requirements.

pin is similar to slicing time, in that you trade-off a smooth gain curve with minimum ripple for a fast response. For instance, applications where the receiver is constantly powered, there's a lot of preamble and the decode time is leisurely give the AGC lots of time to adjust, so a long time constant can be used.

The final step to getting on the air is raising the antenna, and the documentation does a good job of shedding light on this rather black art. In short, the simplest and best-performing setup is a quarter-wave length (e.g., inches = 2808/ftx in megehertz, or about 6–9", depending on transmit frequency) piece of wire (monopole) connected directly to the ANT pin.

Less cumbersome options include coils of wire (helicals) and PCB loops, although range is typically cut to 60 and 30 m, respectively.

Possible enhancements include LC filtering to counter interference from machinery located near the receiver or, at the very least, a resistor offering a DC path to ground affords some input protection from large EM spikes. The antenna can also be located remotely via transmission line, with the caution that an impedance-matched coupling is necessary.

HERE COMES DE CODE

Final integration involves hooking the RF subsystem to a decoder. A number of dedicated chips are available from the likes of Motorola, National,

Holtek, and Microchip. Or, you can roll your own using an MCU.

In principle, it's relatively easy to design your own transmitter because the basic functionality involves gating the RF on and off. However, keep in mind that once you start talking, rather than listening, the FCC is going to insist on oversight. For all but the highest-volume apps, it may be wiser to purchase preapproved commercial units from outfits like Ming Microsystems, Abacom, Radiometrix, or DVP.

The easiest way to get started with the MICRF001 is to pick up the EV kit shown in Photo 1. It consists of an EV board (see Figure 5) combining the MICRF001 with a Holtek decoder, a keyfob transmitter from Ming Microsystems, and all the requisite docs.

The kit offers just enough functionality to evaluate performance and perform range testing, a low-tech exercise that boils down to: press a button on the keyfob, see if the LED on the board lights up, move a step away, repeat.

All in all, the MICRF001 proves RF technology doesn't have to be complicated or expensive (\$3 in volume), thus enabling widespread adoption into new apps.

Although technically, the MICRF001 could work in a wireless keyboard or mouse, FCC restrictions on the 300-400-MHz band rule it out. However, a Micrel app note does show how to use the MICRF001 as the second stage in a 900-MHz wireless modem.

I hope chips like the MICRF001 get the message to the PC powers that be: Wires? We don't need no stinking wires. ☐

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

MICRF001

Micrel Inc.
(408) 944-0800
Fax: (408) 944-0970
www.micrel.com

Decoders

Holtek Microelectronics, Inc.
(Digi-Key)
(800) 344-4539
Fax: (218) 681-3380
www.digikey.com

National Semiconductor
(408) 721-5000
Fax: (408) 739-9803
www.national.com

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

Motorola
SPS Division
(602) 952-3433
Fax: (602) 952-3359

Transmitters

Abacom
(416) 236-3858
Fax: (416) 236-8866

DVP, Inc.
(818) 541-9020
Fax: (818) 541-9423
www.dvp.com

Ming Microsystems
(818) 912-7756
Fax: (818) 912-9598

Radiometrix/Lemos Int'l, Inc.
(508) 798-5004
Fax: (508) 798-4782
www.radiometrix.co.uk/contact/usa.htm

PRIORITY INTERRUPT

'Net Worth



W

When I'm in the booth at the tradeshow, I can always count on a few people asking my advice on starting a business. I guess it must be one of those age things. In any case, I try to avoid coming across as completely bitter and warped. I don't tell them that lawyers and government meddling have made entrepreneurial ventures more of a bureaucratic hassle than a rewarding enterprise these days. I know most of these guys aren't even thinking about company medical insurance and 401K plans yet. They've got a great product idea and they just want to capitalize on it, come hell or high water. So, I smile with my aged wisdom and give them a little advice.

A lot has changed in the 20 years since I started a business. Back then, I just threw \$50k in the pot and believed that if my product idea was sound and I followed the standard business model (albeit limited by the amount of my investment capital), I would succeed. At least that used to be the formula.

Unfortunately, I don't think that model is true anymore. The commercialism of the Internet has completely shredded the old model and dictated some interesting prophecies. In my opinion, entrepreneurial performance in the future will depend less on product competition and more on the business model employed to promote it—specifically, the Internet. For the entrepreneur who understands this new business model, the Internet offers a level playing field where intellect has an opportunity to succeed against overwhelming financial competition.

The Internet is changing the relationship between consumer and producer. Don't think this realization doesn't have some traditional TV-centered big-name companies running scared. The Internet is not, as some have viewed it, just a new marketing channel and faster advertising medium. It's a whole new industry! It's an industry that incorporates personal choice, personal freedom, and personal control in the buying process. Online product promotion has to entice, educate, and entertain.

You don't have to take my word for it. Look at your own behavior patterns and how much less passive you are online. Would you rather spend your time watching a TV sitcom or surfing the web? And, why is it that a 30- or 60- second TV commercial is passively accepted, but if some web advertiser took over your computer screen for a 30-second commercial, you'd go berserk? If you're interested in a print magazine ad these days, you don't fill out a bingo card, you go to the web site. When you're thinking about a new car, you don't go to 10 car dealerships to see what they have; you do a little comparison browsing at the manufacturer and car magazine sites. The reasons are clear—choice, freedom, and control.

When I'm asked startup questions these days, my advice is about the Internet:

- Web users aren't passive.
- Everyone online is a comparison shopper. No amount of advertising dollars turns a poorly rated product into a first-rate seller.
- Internet shopping facilitates a build-to-order business.
- An Internet company has no geographical boundaries. Your competitors and your customers will come from around the world.
- No one sleeps in cyberspace. Your company is viewed as always open and service is always expected.
- The product learning cycle online is much faster than the traditional model. Online selling is faster and therefore customer feedback is faster.

The kind of person who asks me about starting a business isn't a venture capitalist giving me an IQ test. It's usually an individual with a great idea and limited capital. The traditional business model has always been an obstacle because product advertising and promotion were so costly. Low-cost web communication gives the entrepreneur a chance to play the game again.

But even with the reality of a web-conscious society, just using the Internet isn't enough. Ultimate success for any new business depends on its ability to offer products with real performance advantages. Efficiency and low cost aren't sufficient. Purchasers will have to be given the information required to make decisions in an informative and entertaining way.

The bottom line isn't hard to understand. The Internet is definitely the vehicle for someone starting a business, but it is a deathtrap for the mediocre.

It's almost enough to make a guy want to do it all over again.... Well, not really.



steve.ciarcia@circuitcellar.com