EMBEDDED PC MONTHLY SECTION

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#91 FEBRUARY 1998

COMMUNICATIONS

Making NT Play Nice with Unix

Embedding Voice Recognition

Zilog's Z89*xx*3 = DSP + MCU

Converting PC GUIs for NonPC Devices

$3.95 U.S.
$4.95 Canada

And Yet More Change

In my tenth-anniversary editorial last month, I spent some time talking about change—specifically, the changes that have come about over the past decade. One area in which we've seen drastic change just in the past few years is how we communicate.

Printed publications have been around since the printing press was invented in the 15th century, and will continue to thrive for the foreseeable future. There is something about holding a book or magazine that people still enjoy and aren't willing to give up. They don't require special equipment to read. They're lightweight, low cost, highly portable, and don't require any power. On the downside are the cost of entry, use and cost of paper, required lead time, and limited distribution.

Along came the electronic bulletin board system, and suddenly time and distance were compressed, enabling anybody with a computer and modem to publish their own work for anyone in the world to access instantly. The next step was the Internet and the World Wide Web, further reducing cost and increasing availability and quality.

Publishers of print media are seeing the handwriting on the wall in a big way, and virtually everybody has some kind of on-line version of their print publication. We at Circuit Cellar brought the Circuit Cellar BBS on-line back in 1986 to first support Steve's *BYTE* articles and to later support *Circuit Cellar INK*. It was a tremendous success, fueling active discussions between talented engineers located throughout the world. Long-time readers were also treated to a sampling of those discussions each month through my ConnecTime column.

The Circuit Cellar BBS has just gone off-line for good as a BBS. However, various Internet elements have replaced it, making the same functions available to anyone in the world with Internet access. Magazine information and articles can be obtained from the Circuit Cellar Web site (www.circuitcellar.com). Files can be downloaded from our FTP site (ftp.circuitcellar.com). And, messages can be exchanged in a public forum through our new newsgroup server. Don't look for our newsgroups among the other Usenet groups you may be receiving from your ISP (Internet Service Provider), though. You must point your newsreader at our server at bbs.circuitcellar.com to see what groups are available and what discussions are going on. If you're still not sure what I'm talking about, there is more information on the Web site.

In the coming months, I'll be spending more time enhancing and developing our on-line presence. Please send any suggestions or comments along with those article proposals and ideas to editor@circuitcellar.com and I'll respond as quickly as possible. I'll see you on-line!

[illegible]

ken.davidson@circuitcellar.com

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER Steve Ciarcia	**ASSOCIATE PUBLISHER** Sue (Hodge) Skolnick
EDITOR-IN-CHIEF Ken Davidson	**CIRCULATION MANAGER** Rose Mansella
MANAGING EDITOR Janice Hughes	**BUSINESS MANAGER** Jeannette Walters
TECHNICAL EDITOR Elizabeth Laurençot	**ART DIRECTOR** KC Zienka
WEST COAST EDITOR Tom Cantrell	**ENGINEERING STAFF** Jeff Bachiochi
CONTRIBUTING EDITORS Rick Lehrbaum, Fred Eady	**PRODUCTION STAFF** John Gorsky, James Soussounis
NEW PRODUCTS EDITOR Harv Weiner	

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

ADVERTISING

ADVERTISING SALES REPRESENTATIVE
Bobbi Yush (860) 872-3064 — Fax: (860) 871-0411 — Email: bobbi.yush@circuitcellar.com

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

CONTACTING CIRCUIT CELLAR INK

SUBSCRIPTIONS:
INFORMATION: www.circuitcellar.com or subscribe@circuitcellar.com
TO SUBSCRIBE: (800) 269-6301 or via our editorial offices: (860) 875-2199
GENERAL INFORMATION:
TELEPHONE: (860) 875-2199 FAX: (860) 871-0411
INTERNET: info@circuitcellar.com, editor@circuitcellar.com, or www.circuitcellar.com
EDITORIAL OFFICES: Editor, Circuit Cellar INK, 4 Park St., Vernon, CT 06066
AUTHOR CONTACT:
E-MAIL: Author addresses (when available) included at the end of each article.
ARTICLE FILES: ftp.circuitcellar.com

For information on authorized reprints of articles, contact Jeannette Walters (860) 875-2199.

CIRCUIT CELLAR INK®, THE COMPUTER APPLICATIONS JOURNAL (ISSN 0896-8985) is published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (860) 875-2751. Periodical rates paid at Vernon, CT and additional offices. **One-year (12 issues) subscription rate USA and possessions $21.95, Canada/Mexico $31.95, all other countries $49.95. Two-year (24 issues) subscription rate USA and possessions $39, Canada/Mexico $55, all other countries $85.** All subscription orders payable in U.S. funds only via VISA, MasterCard, international postal money order, or check drawn on U.S. bank. **Direct subscription orders and subscription-related questions to Circuit Cellar INK Subscriptions, P.O. Box 698, Holmes, PA 19043-9613 or call (800) 269-6301.**
Postmaster: Send address changes to Circuit Cellar INK, Circulation Dept., P.O. Box 698, Holmes, PA 19043-9613.

INSIDE ISSUE 91

EMBEDDED PC

www.circuitcellar.com

READER I/O

THAT DARN *x-y* RATIO

As a broadcast professional with over 20 years in the (PAL) business, I was happy to read Do-While Jones' article "HDTV—The New Digital Direction," (*INK* 86). Anything that helps people understand how high bandwidth content travels along a limited-size pipe is good.

One very minor mistake that I've seen made a thousand times is the reference to a 4 × 3 video wall. If you stack monitors symmetrically 4 across × 3 vertically, you end up with a 16:9 video wall—oops! To retain the aspect ratio of the individual monitors, you need to expand both dimensions by the same ratio!

Michael Coop
mcoop@pop.jaring.my

RENAMING "FUZZY LOGIC"

Ken's editorial in *INK* 88 ("Stuffed Mentality") made a good point—"fuzzy logic" sounds like fuzzy thinking. I suggest we rename it "quantitative logic." It is logic in which truth is a quantity (i.e., something you can have a little or a lot of).

There are other nonclassical logics, too. I'm currently doing research on defeasible logic, a system where generalizations are automatically overruled by specific exceptions. It's a good fit to the way human beings describe things.

Michael A. Covington
mc@ai.uga.edu

SLACKING OFF?

A programmer friend passed me a copy of *INK* 88 for the articles on automotive applications for fuzzy controls. Constantin von Altrock's "Fuzzy Logic in Automotive Engineering" was very clear and gave me a much better understanding of how and why fuzzy logic is being used. Unfortunately, Constantin shows his lack of automotive background (and/or the results of reading a poorly translated paper) in a couple of places—the article would have been much more convincing without these mistakes.

First of all, when the car is moving and a wheel stops turning, it is "locked" (not "blocked").

Also, the variable *s* in the text (p. 13) and in Figure 1 is "slip ratio" (not "slack"). There are a number of definitions for *s* used in different parts of the world. There's a summary of eight different slip-ratio relationships starting on p. 39 of *Race Car Vehicle Dynamics* (W.F. and D.L. Milliken, SAE, www.sae.org, 1995).

In Figure 1, the curve for "Snowy Road" would perhaps be more accurate if labeled "Hard Packed Snow". The three curves shown are all representative of tire performance on "hard" surfaces, where there is a definite peak in *m* (tire-road friction coefficient).

To give but one example of the difficulty of designing an ABS algorithm, consider that unpacked snow, gravel, and sand are all "deformable" surfaces, and braking on deformable surfaces has a completely different characteristic than on hard surfaces. On a deformable surface, a locked wheel usually gives the best braking (but no steering control), perhaps by building up a wedge of material in front of the tire. I say "perhaps" because tire-road friction on different surfaces is still not well understood. The inability of current production ABSs to properly distinguish between different surfaces remains a major problem.

The stopping ability of ABS on various slippery surfaces was discussed in a recent article (D. Simanaitis, "ABS: Putting a Stop to it All," *Road & Track*, July 1997). In their tests, locking the brakes (with the Mercedes-Benz ABS disarmed) produced shorter stopping distances on three out of four different icy/snowy surfaces tested. The ABS also lost on gravel and tied on sand. ABS won on wet roads and won dramatically on dry roads.

Douglas Milliken
bd427@freenet.buffalo.edu

Thanks for such detailed feedback. I'd like to briefly comment on your issues.

I have no doubt that it's "locking" instead of "blocking." I used "blocking" because "ABS" stands for Anti-Blockier System in German.

Regarding s, *the German is "Schlupf," which in technical dictionaries is translated to "slack" (which Oxford defines more precisely than it does "slip"). And, you're right:* s *has many definitions worldwide. Rather than discussing these, I took one, showed the definition, and focussed on fuzzy-logic use in this system. I don't think any other approach would have helped the reader.*

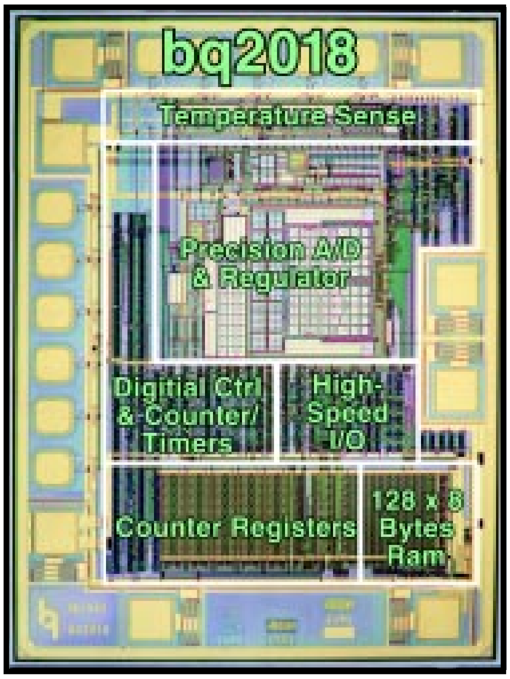
I limited the discussion on hard surfaces because discussing all the different things that occur within an ABS situation was not my goal. But, you're absolutely correct: there's a lot more to say about ABS.

Constantin von Altrock
cva@inform-ac.com

NEW PRODUCT NEWS

Edited by Harv Weiner

BATTERY POWER MINDER



The **bq2018 Power Minder IC** provides state-of-charge information for any type of rechargeable battery, including Li-Ion, NiMH, NiCd, lead acid, and rechargeable alkaline. Designed for battery-pack integration, the 8-pin bq2018 communicates critical battery information over a single-wire control/data serial interface to an intelligent host controller. Typical applications include cell phones, PDAs, and personal organizers.

The bq2018 measures the voltage drop across a low-value series sense resistor between the battery's negative terminal and the battery-pack ground contact. By using the accumulated counts in the charge, discharge, and self-discharge registers, an intelligent host controller can determine battery state-of-charge information. An internal offset count register improves accuracy.

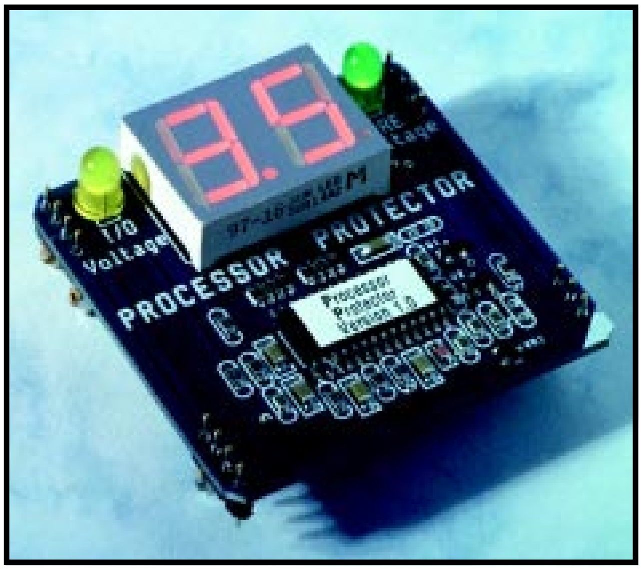
The bq2018 features 128 bytes of NVRAM, including 115 bytes of user RAM for storing battery characteristics, charge data, or ID information, thus eliminating the need for a separate battery ID chip. An internal ADC and reference operate from 2.8 to 5.5 V and at less than 80 µA. Standby-mode current is less than 10 µA and data-retention current is less than 50 nA, so critical information can be stored for over 10 years with 5 mAh of battery capacity.

The 8-pin, 150-mil SOIC package of the bq2018 is small enough to fit in the crevice between two adjacent cells. It is priced at **$1.85** in 10k quantities. The bq2118 Power Minder miniboard incorporates all of the necessary components for a fully functional implementation that sells for **$4.40** in 10k quantities.

Benchmarq Microelectronics, Inc.
17919 Waterview Pkwy.
Dallas, TX 75252
(972) 437-9195
Fax: (972) 437-9198
www.benchmarq.com

#501

PROCESSOR PROTECTOR



Incorrect jumper settings for the dual voltage levels on new microprocessors manufactured by AMD, IBM, Cyrix, and Intel can shorten CPU life and cause erratic operation. System lockup, memory errors, and other intermittent phenomena may be the result of an over- or under-voltage condition. Poorly written instruction manuals and the limitations of verifying the core voltage on the chip are two reasons why incorrect settings can occur. The **Processor Protector** provides a simple means to confirm the proper values.

The Processor Protector plugs into the CPU socket and indicates the value of the core and I/O voltages applied on a two-digit LED. Lights on the unit indicate which voltage is being measured. The Processor Protector is compatible with Socket 5 and 7 motherboards and sells for **$59**.

Autotime
6605 SW Macadam Blvd.
Portland, OR 97201
(503) 452-8577 • Fax: (503) 452-8495
www.autotime.com

#502

NEW PRODUCT NEWS

TMS320C6*x* SINGLE-PROCESSOR DEVELOPMENT SYSTEM

The **HEVAL6A** TMS320C6*x* single-processor development system integrates a 200-MHz (1600 MIPS) TI TMS320C6201 processor with several memory types, including SBSRAM, SDRAM, and asynchronous SRAM. A mezzanine slot supporting a pair of serial interfaces and two I/O interface slots supporting a variety of data-acquisition and communications modules enable developers to integrate the hardware into their chosen application environment. The DSP hardware can be programmed and debugged via the PC board's 16-bit ISA-bus host interface and JTAG controller. It can be booted without a host computer (for stand-alone or embedded applications) via its onboard flash memory.

Development tools include the TI 'C6*x* Code Development Tools (C compiler, assembly optimizer, assembler, and linker), software loader utility, GO DSP Code Composer for C source debugging, and PC-based API for DOS and Windows.

The HEVAL6A is priced at **$14,000**.

Traquair Data Systems, Inc.
114 Sheldon Rd.
Ithaca, NY, 14850
(607) 266-6000 • Fax: (607) 266-8221
www.traquair.com **#503**

POWER-TO-FREQUENCY CONVERTER

The **AD7750** is designed for residential and industrial power-metering applications. It can be configured for power measurement, voltage-to-frequency conversion, or converting the product of two voltages to a frequency.

It contains two ADCs, a multiplier, offset compensator, digital-to-frequency converter, reference, and other conditioning circuitry. Both channels are driven by differential gain amplifiers—channel 1 with selectable gains of 1 and 16, and channel 2 with a gain of 2. A high-pass filter can be switched into the signal path of one channel to remove offset effects.

AD7750
ANALOG DEVICES
AD7750
Instantaneous power output
Bipolar input on a single chip
Multiple output frequency

The AD7750's switched-capacitor architecture allows a bipolar analog input of 11 V with a single 5-V power supply. Nonlinearity for either input is less than 0.05% maximum.

The device features two sets of frequency outputs that consist of fixed-width pulse streams with pin-selectable frequencies. Low frequencies are suitable for stepping motors, while higher frequency pulse streams are appropriate for calibration and test. In the signed mode, outputs can be configured to represent the result of four-quadrant multiplication. In the unsigned mode, magnitude-only outputs are always positive regardless of the input polarities. A reverse-power indicator activates when negative power is detected in the unsigned mode.

The AD7750 comes in 20-pin SOIC and DIP packages and is priced at **$2.50** in 100,000-piece quantities.

Analog Devices, Inc.
804 Woburn St. • Wilmington, MA 01887
(781) 937-1428 • Fax: (781) 821-4273
www.analog.com **#504**

NEW PRODUCT NEWS

SCOPE UTILITY SYSTEM

EZ-View-SA functions as a data-acquisition system, oscilloscope, digital voltmeter, and chart recorder. The hardware module attaches to a parallel printer port and provides six single-ended channels of input with 12-bit resolution. An input range of ±10 VDC is available at an input impedance of 330 kΩ.

The software features auto-installation and configuration, mouse or keyboard control, remote start, and trigger controls. It works with all MS-DOS-, Win3.1-, and Win95-based computers with '386 or higher processors and VGA or better screens. Operating modes include real-time monitoring (oscilloscope), data acquisition (record), and rapid record (burst).

Features include gain adjustments, bias offsets, scale selection, variable sampling-rate and run-time selection, channel labeling, triggering, auto-scaling, and remote-start options. Acquired data can be transported to standard spreadsheets and expanded for detailed analysis. A notes feature permits a brief text description of the data to be attached to saved files.

EZ-View-SA costs **$199**, including the data-acquisition module, power supply, data cable, instruction manual, and screwdriver. Options include 16-bit data resolution, remote battery power supply, and probes.

Mid-Atlantic Systems Co.
2284 Golden Pond Ct. • Fenton, MI 48430-1097
(810) 750-4140 • Fax: (810) 629-4988
www.mid-atl-sys.com ` **#505**

NEW PRODUCT NEWS

WIRELESS KEYBOARD

SurfMate is a 79-key plug-n-play wireless keyboard that requires no software installation. The user plugs SurfMate's receiver unit into the computer's keyboard port to establish the interface. An optional integrated pointing device replaces the mouse. SurfMate is compatible with all Internet applications, including Web browsers and E-mail. However, it can be used with any software (e.g., presentation programs, games, word processing, accounting, etc.).

SurfMate can be positioned almost anywhere in a room and still maintain complete control of the computer. It transmits through infrared LED at distances up to 45′ (14 m) and, depending on the distance to the PC, at horizontal angles up to ±60° and vertical angles up to ±50°.

SurfMate features an ergonomic design with full-sized keys and includes three Windows 95 keys. It comes equipped with four Duracell AA alkaline batteries, weighs only 21 oz. (including batteries), and sells for **$129.99**, including shipping.

US Electronics
585 N. Bicycle Path, Ste. 52
Port Jefferson Station, NY 11776
(516) 331-2552 • Fax: (516) 331-1833
www.uselectronics.com/surfmate

#506

FEATURES

FEATURE ARTICLE

Brad Stewart

Low-Cost Voice Recognition

Brad's Tiny Voice—based on an 'HC705 and powered off a 9-V battery—can be trained to recognize up to 16 command templates and costs less than $5. Toys, voice-activated padlocks, and remote controls had better listen up.

Voice recognition has come a long way in the past five years, due mainly to the advent of cheap and powerful PCs equipped with Pentiums and MMX technology. Performance continues to improve to the point where parts of this article were comfortably voice-dictated via Kurzweil VoicePlus.

But, this performance comes at a cost. You need fast Pentiums with MMX, at least 16 MB of DRAM, and even more disk stroage.

What if your application has a budget of a couple dollars? Can you still embed some form of voice recognition or voice command and control into your product?

In this article, I'll show you how to implement a voice-command system for under $5. I conclude with some application examples and recommendations to improve the system even further.

TINY VOICE

My system—Tiny Voice—is based on a low-cost, 20-pin single-chip controller. It's a speaker-dependent, template-based, isolated-word recognizer. You train it to recognize your voice.

Up to 16 voice patterns are stored in a nonvolatile 512-byte serial EEPROM. Five push buttons enable programming and operation, and seven LEDs give status.

For embedded systems, Tiny Voice can be controlled over a parallel or serial protocol from a host microcontroller or it can run stand-alone. The source code may be modified to fit your requirements.

At under $5, Tiny Voice won't do dictation. But, it's good for applications like toys, repertory phone dialers, voice-activated padlocks, security systems, remote controls, and other low-cost consumer products.

A voice command can be one or several words, with a total maximum length of 1.6 s and a minimum of 0.2 s. Response time is typically <100 ms. By carefully selecting the vocabulary and context, over 95% recognition accuracy is possible.

The heart of the system is the 68HC-705J1A Motorola 8-bit processor. There were a number of reasons why I chose this part over a comparable one from Zilog or Microchip.

There's sufficient RAM (64 bytes) to buffer the input waveforms and hold template structures, and its 1240 bytes of ROM provide enough program storage. Also, interrupts are supported, including changes on the I/O lines.

This system is inexpensive (<$2) in high volume. The development kit is cheap, too, at $99.

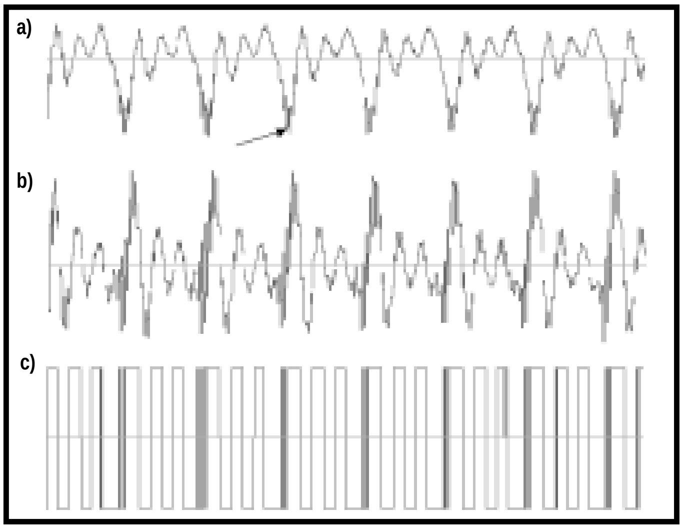


Figure 1a—*This is a waveform of the voiced sound "ee" as in "speech." The arrow points to high-frequency wiggles corresponding to the second formant (F2). Note that these wiggles do not cross the zero axis.* **b—***After preemphasis or high-pass filtering, the F2 components now cross the zero axis with the same waveform.* **c—***After being infinitely clipped, the waveform of Figure 1b is a square wave showing both F1 and F2 components. This signal is applied to the microprocessor via a digital input pin.*

Shown in Photo 1, the Tiny Voice system was built on a 3″ × 3″ breadboard and is powered off a 9-V battery. Standby current consumption is ~2 mA, which is primarily due to the op-amp and electret microphone bias.

With some added power management, standby current could be reduced to a few microamps. Operating power while sampling and analyzing speech is ~10 mA.

THEORY OF OPERATION

The 68HC05 processor is very simple. There are no ADCs, so you need a way to convert the time domain signal to a format the microcontroller can recognize.

The small amount of memory requires a lot of approximations and simplifications to convert the speech into a small set of features.

To meet these limitations, I use a simplified formant tracker. The microphone input is high-pass filtered and then infinitely clipped using two operational amplifiers. The resulting square wave is connected to an MCU input.

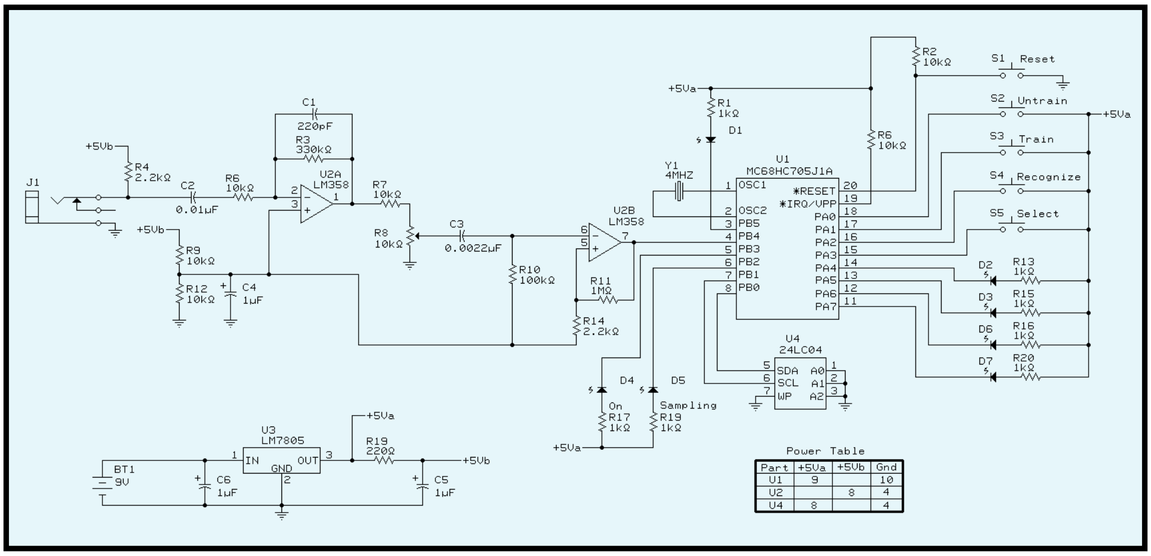
By sorting and tallying long and short pulse widths of the square wave, you get a crude but effective two-channel frequency analyzer. One channel gives frequencies below 1500 Hz, and the other ranges from 1500 Hz to 5 kHz.

These two frequency areas roughly define F1 and F2, the two formant regions of speech. It's a well-known principle that F1 and F2 for a given speaker and a given set of vowels remain the same.

Using F1 and F2 was first tried in 1952 by Bell Labs employing vacuum tubes and capacitors for memory. Crude as it sounds, that system achieved 97% recognition accuracy!

The input signal is high-pass filtered (i.e., pre-emphasized) to accentuate the F2 frequencies. Figure 1 illustrates why this is necessary.

Figure 1a is a sample of the voiced vowel sound "ee" as in "speech." Note the F2 component shown by the arrow. Also note that these high-frequency



Part	+5Va	+5Vb	Gnd
U1	9		10
U2		8	4
U4	8		4

Figure 2—*An electret condenser microphone (not shown) is biased to 5 V via R4. The signal is then amplified by U2a. C2 and R6 (along with C3 and R10) form a high-pass filter. The output is fed to the second op-amp, which is configured as a comparator whose output is connected to PB4 of the 68HC705J1. The EEPROM has a two-wire I^{2}C interface, which is connected to PB1 and PB0. The remaining pins of the processor are connected to LEDs and push buttons.*

wiggles do not cross the zero axis. Thus, if the waveform is infinitely amplified and clipped, the square wave would not reveal the F2 component.

However, Figure 1b shows what happens after pre-emphasis. The F2 wiggles cross the zero axis, and the resultant infinitely clipped square wave now contains both F1 and F2 (see Figure 1c).

TINY HARDWARE

Figure 2 shows a schematic of the system. An electret condenser microphone is biased to 5 V via R4. The signal is then amplified by U2a.

C2 and R6 (along with C3 and R10) form a high-pass filter, with a cut-off frequency of 1600 Hz with an added zero at 800 Hz. This setup provides a pre-emphasis function.

C1 serves as a mild antialiasing low-pass filter. The output is fed to the second op-amp, which is configured as a comparator with some hysteresis. R8 sets the threshold of the comparator.

The comparator's output is a square wave that's applied to an input pin of the processor. The threshold defines the beginning and end of a speech utterance. With no signal present, the second op-amp's output is at a DC level.

Voice pattern data is stored in a non-volatile EEPROM. For this project, I selected Ramtron's FM24C04, which uses ferroelectric cells.

It has several advantages over a more generic part. For one thing, the FRAM part can be written to over 10 billion times, compared to about 10k cycles with a generic EEPROM. This feature is important here because the first 128 bytes are used for scratch-pad memory and are constantly written to.

Also, it has a deep write buffer. So, once the starting address is specified, memory address is autoincremented and additional writes can be performed with no more intervention. As a result, writing to the device is very fast.

Generic parts, however, require you to set up the address every other byte before you write data. This task creates additional time overhead that may cause a bottleneck in the software flow—a major concern in a real-time system.

The FM24C04 has a low standby current of 25 µA as well as a low operation current of 100 µA. So, it's well suited for battery operation.

The EEPROM's first 128 bytes hold the transformed input utterance to be recognized or trained. Locations 128–512 store the feature vectors of a previously trained utterance. Each vector occupies 24 bytes, so the maximum number of templates that can be stored is 16.

The rest of the circuit comprises a 5-V regulator, switches, and LEDs.

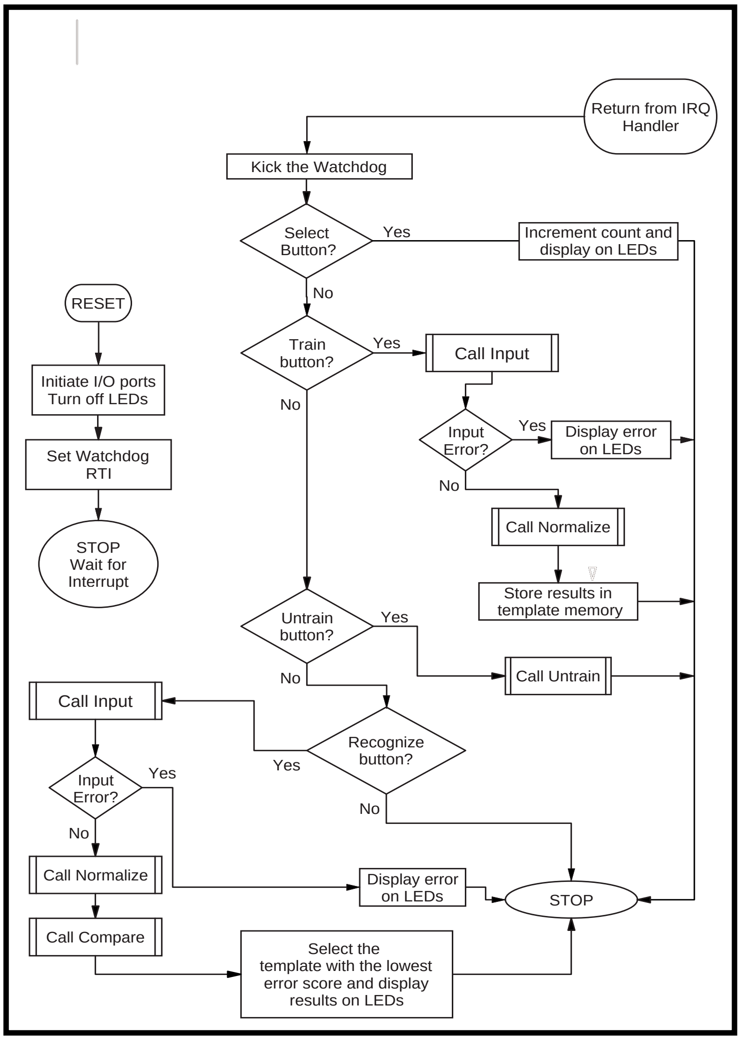


Figure 3—*The main routine performs the event handler. Events are generated by an interrupt caused by pressing a push button or by system reset. The events dispatched are Select, Train, Untrain, and Recognize.*

TINY USER INTERFACE

Before discussing the voice-recognition software, I want to describe the interface and how the system works from the user's point of view.

Seven LEDs and four switches compose the Tiny Voice user interface. LEDs D2, D3, D4, and D5 make up a four-bit binary number that gives Tiny Voice's status. It can either be the index of a voice command or an error message.

When power is connected or when the Reset switch is pressed, the Stop mode is entered. Pressing a push button activates the system and performs a certain function.

Pressing Select displays a binary number from 0 to 15 on four LEDs which selects the template number to be trained or untrained. Each time Select is pressed, the number increments to 15 and back to 0.

Pressing Train starts the Training mode. The On LED is activated, and the user is prompted to say the command to be trained.

While the user is speaking, the Sampling LED is lit during periods of speech and off during periods of silence. If the training is successful, the template is stored in EEPROM at the selected template location and the system enters the Stop mode.

Untrain modifies the data in the stored template so the pattern-matching algorithm skips over this template and does not consider it as a possible candidate.

This is useful for context switching of vocabularies. For example, out of the 16 templates, you may only need to scan for two words (e.g., "yes" or "no"), while ignoring the remaining 14.

To enable a template that was previously untrained, press the Train button and then press another button (e.g., Select) before speaking.

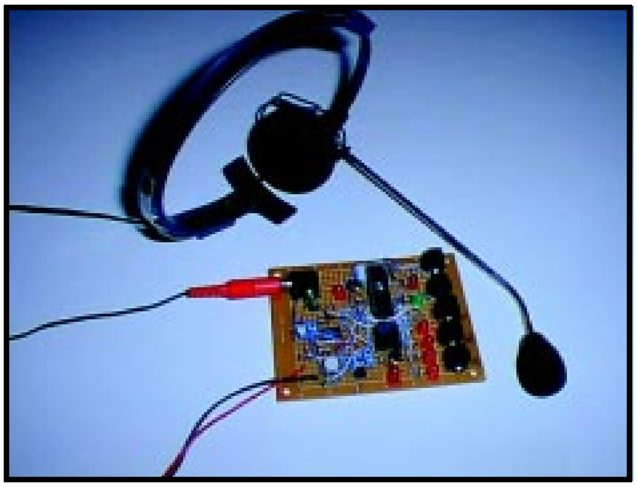


Photo 1—*My prototype was built on a 3″× 3″ breadboard and is powered off a 9-V battery. The only ICs are the 68HC705J1 processor, LM358 dual-operational amplifier, the 4096-bit FM24C04 FRAM serial memory, and a 78L05 5-V regulator.*

In Recognition mode, the speech is sampled and analyzed. The On LED is activated, and the user is prompted to say a previously trained command. As before, the Sampling LED is lit during speech and off during periods of silence.

The input is compared to the templates in memory and a decision made. If recognition is successful, the result is displayed on the four LEDs in binary.

When Reset is pressed, Stop mode is entered and the system is ready to accept a push-button command. Previously trained commands are not erased.

When an error occurs, the Error LED (D1) is lit and the error code is displayed in binary using the same four LEDs that display the template index number. After ~2 s, the LEDs go off and the system enters Stop mode.

The error codes—Time Out, Buffer Full, and Not Recognized—are defined in the header file.

After Train or Recognize is pressed, the system waits for valid speech input. If no input occurs after ~6.5 s, the system enters the Stop condition and the Time Out error code is displayed.

On the other hand, if the length of the utterance is longer than 1.6 s, the system enters the Stop mode and the Buffer Full error is displayed.

The Not Recognized error code is displayed if the input utterance doesn't match a stored template. The system then enters Stop mode and waits for new input.

TINY ALGORITHMS

The software for Tiny Voice was written entirely in assembly. There is a total of eight routines.

The main program, `MAIN.ASM`, responds to events and schedules the remaining subroutines.

`COMPARE.SUB` handles the pattern matching. It compares the input template to each active template in memory and calculates the best match.

`EEPROM.SUB` handles the reading and writing of data to the EEPROM. It bit-bangs two I/O pins to simulate an I^{2}C protocol used by the EEPROM.

`IRQ.SUB` is the interrupt handler. Interrupts are caused by a button press.

The most complicated routine is `INPUT.SUB`. It samples the input, determines where the word starts and ends, and builds up the voice template.

`TIME_NOR.SUB` normalizes the length of the speech input to a fixed length of twelve two-element data values.

`DIV16_8.SUB` is an integer divide routine that divides a 16-bit number by an 8-bit number. This routine is called repeatedly by the time-normalization routine.

And finally, `DELAYMS.SUB` is a simple program where a delay is set by the value passed in the accumulator.

Tiny Voice is entirely event-driven and spends most of its time in the Stop mode. Events are caused by the interrupt of pressing push buttons. The event handler is shown in Figure 3.

INPUT ROUTINE

When a Recognize or Train event occurs, the input routine is invoked (see Figure 4). A timer is set up and polled until 110 µs has elapsed.

An interrupt routine could have been used to time the samples every 110 µs, but I was concerned that the overhead to service the interrupt might make it difficult to complete all the paths in the input routine within 110 µs.

Once the time elapses, the input square wave is sampled. If the sign changes from the previous measurement, one of the two frequency bytes is updated.

The threshold limit is set to six. In other words, if the pulse (positive or negative) is greater than six samples (roughly corresponding to 1.5 kHz), the "high" frequency byte is incremented by one. If it's less than six, the "low" frequency byte is incremented.

The rest of the routine is basically a state machine that uses speech activity as an input to determine a utterance bounded by silence. At each rising or falling edge, another byte counts the zero crossings.

After 256 samples, a frame counter advances and several tests are made. If the frame counter is greater than 64, the input buffer is filled (i.e., you spoke too long) or there is too much background noise, and an error is generated.

Otherwise, a timeout value is decremented and tested. This setup enables the routine to exit if too much time elapses before any sound is input.

If the buffer isn't full or a timeout has not occurred, then it tests the zero-crossing counter. Too low a value signifies silence, and a silence counter is incremented.

Otherwise, a sound-activity counter is incremented. If the sound-activity value is above a certain threshold and the silence value is high enough, the routine exits with a valid data sample.

TIME NORMALIZATION

Words vary in length. But for this algorithm to work, the lengths must be normalized to a fixed value.

Each sample consists of two bytes sampled over one frame of 256 samples. The unnormalized data in the first 128 bytes of the EEPROM is normalized to a set of 12 vectors in main RAM.

The vector in RAM is built up, element by element, by down- or up-sampling the raw data in EEPROM. Since there are two elements per feature, a template has a fixed memory length of 24 bytes.

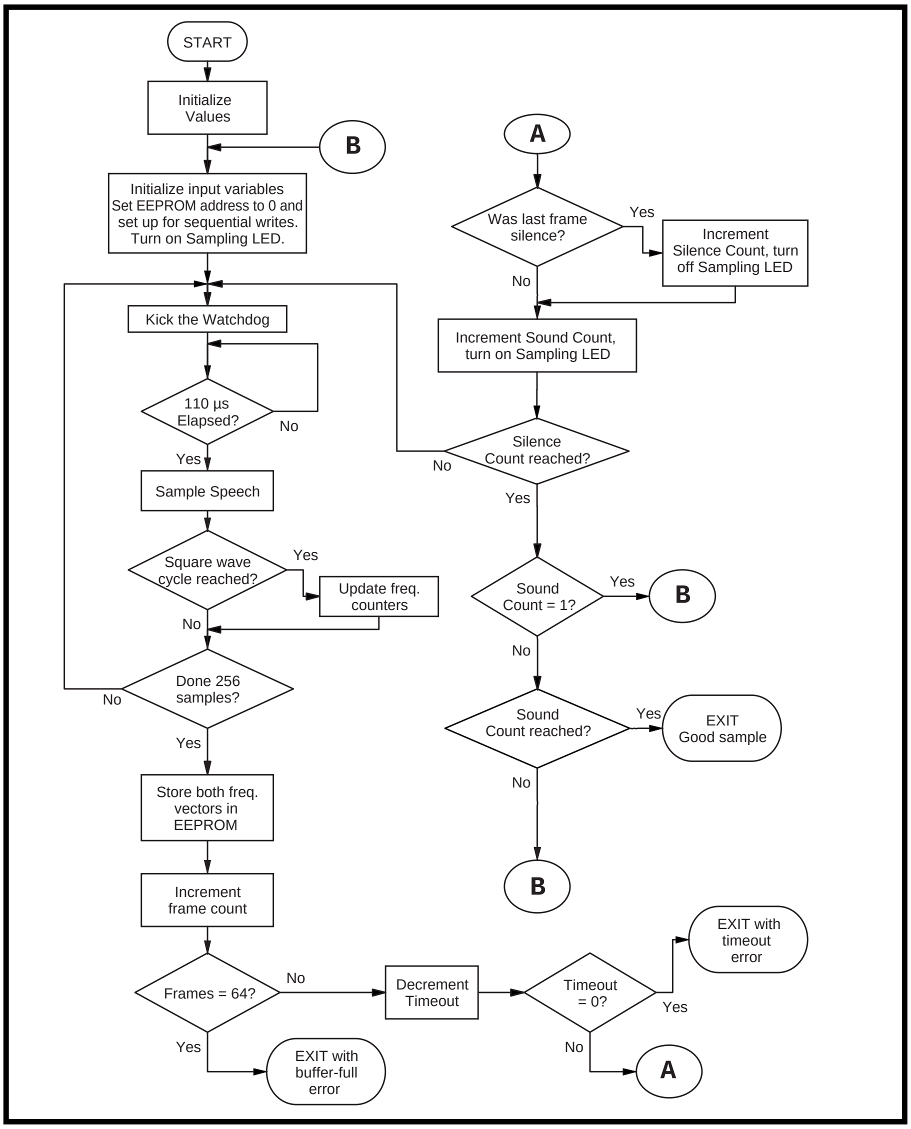


Figure 4—*Every 110 µs, the square-wave input is sampled and several options are considered, depending on the state of the frame, zero-crossing, silence, and sound counts. The state machine effectively captures the input utterance, while rejecting short bursts and input errors due to excessive background noise.*

THE MAIN ROUTINE

If the event is for training, the normalized vector in RAM is stored in memory according to the template number selected. Templates are stored in memory locations 128–512, which allows for sixteen 24-byte templates. No comparisons are performed.

If the system is recognizing, the normalized input utterance, which is stored in RAM, is compared element by element to each previously trained template stored in EEPROM.

The comparison is a simple Euclidean distance measure, and an error value accumulates. The minimum error value is selected and compared to a threshold.

If the result is above the threshold, the system rejects the recognition. If the value is low enough, the word is recognized.

Well, almost. Two more criteria must also be met: the score must be low enough, and the two smallest scores must differ by a large enough value.

TINY APPLICATIONS

For testing purposes, the system was trained with eight words: "VCR", "television", "telephone", "stereo", "CD", "PC", "yes", and "no". Each word was trained twice, thereby occupying 16 templates.

Recognition accuracy approaches 100% when background noise isn't too severe. It also works with ~90% accuracy using speakers who didn't train the system.

A speaker-independent vocabulary can be constructed by having multiple trainings of a few words. For example, training "yes" and "no" eight times over a set of different speakers yields excellent results.

A note of caution: when using Tiny Voice, don't use a lot of short words (e.g., the numbers "one", "two", etc.). They're a bit beyond its capabilities.

And watch for commands that sound alike. For example, "on" and "off" will get you in trouble. Instead, try "turn on" and "off please".

A fun application might be a voice-activated padlock. Change the code so you have to enter one, two, or three voice commands in sequence. Then, multiply the scores. If the result is small enough, then "open sez me."

FUTURE TINY ENHANCEMENTS

Naturally, there are ways to improve the system. I was surprised by the HC05's speed. I also wound up with at least 200 bytes of leftover ROM for more code. Tiny Voice's code is modular, and updates can be easily added.

I can increase the EEPROM capacity to 1 or even 2 KB. This size would provide more template storage or allow for more frame features to better resolve differences in speech patterns.

I'd also like to add some fuzzy logic to the pattern-matching algorithm to improve recognition accuracy and the rejection criteria.

Adding a serial port instead of push buttons and LEDs could reduce cost and add more functionality. Threshold values could be changed, templates uploaded and downloaded, and so on.

I want an MCU-controlled gain adjustment on the input for different microphone levels and background noise.

Another improvement would be to add a dynamic time warp (DTW) algorithm to the pattern-matching routine. The DTW takes into account slight variations on how each word is pronounced—in particular, variations in lengths of phonemes.

But with only 200 bytes of code space left over, adding a DTW would be challenging. A first-order approximation may be achievable, however.

I'd rather use C than assembly language. When I started this project, I knew squeezing this functionality into 1200 bytes would be tough. So, a high-level language was out of the question.

Since then, I've had the opportunity to try out a C compiler from Byte Craft. The good news is, it generates small enough code. The bad news: I wish I'd used it earlier.

And as a final wish, I would like to use a different processor. Of all these improvements, this one is probably the best. You can now get equivalent MCUs with built-in ADCs, which would provide more elaborate signal processing and better noise rejection.

One of the best candidates for a low-cost system is the Sharp SM8500 8-bit MCU. It has almost everything you need for an embedded voice-command system, including a 10-bit ADC (8 channels) and an 8-bit DAC, which is useful for voice feedback and verification.

The SM8500 features SIO and UART ports to communicate with other system devices, 2 KB of internal RAM, as well as internal ROM and the ability to access external ROM or RAM. It also offers 80+ I/O pins for keypad and display interfacing, hardware multiply and divide, and a 250-ns instruction cycle time. And, it costs under $3.

If you're willing to spend a bit more, then a new level of performance may be realized. New 32-bit RISC MCUs are becoming available in the sub $15 or even sub $10 range.

For example, the Sharp ARM710M RISC processor, running at a conservative 16 MHz, performs a complete FFT-Mel-Cepstrum analysis using only 50% of the processor's resources.

With the ability of RISC processors to address large amounts of memory, you have the ingredients to put together a dictation system like the one I'm using now. And, it can run off a couple pen-light batteries! ■

Brad Stewart is currently the product technical manager for RISC processors at Sharp Electronics. He also served as technical director for IPI, which specialized in voice-recognition and speech-compression software, and vice president of Covox, which specialized in multimedia products. You may reach Brad at bstewart@e-z.net or bstewart@sharpsec.com.

SOFTWARE

Source code (tinyvoice.zip) for this article may be downloaded from the Circuit Cellar Web site.

REFERENCES

B. Georgiou, "Give an Ear to Your Computer," *BYTE*, 56–91, June, 1978.

Motorola, *MC68HC05J1A Technical Data Manual*, 1997.

Sharp Electronics, *SM8500 User's Guide*, 1997.

B.C. Stewart and S. Sidman, "Design and Use of Voice Recognition in Embedded Applications," Paper presented at ESC East, Boston, MA, 1997.

SOURCES

68HC705J1A
Motorola
MCU Information Line
P.O. Box 13026
Austin, TX 78711-3026
(512) 328-2268
Fax: (512) 891-4465

FM24C02
Ramtron Intl. Corp.
1850 Ramtron Dr.
Colorado Springs, CO 80921
(719) 481-7000
Fax: (719) 481-9294
www.ramtron.com

C Compiler
Byte Craft Limited
421 King St. N.
Waterloo, ON
Canada N21 4E4
(519) 888-6911
Fax: (519) 746-6751
www.bytecraft.com

ARM710M, SM8500
Sharp Electronics Corp.
Microelectronics Gr.
5700 NW Pacific Rim Blvd., Ste. 20
Camas, WA 98607
(206) 834-2500
Fax: (206) 834-8903
www.sharpmeg.com

I R S

401 Very Useful
402 Moderately Useful
403 Not Useful

FEATURE ARTICLE

Richard Ames

Building an Embedded Web Server from Scratch

Tired of surfing? Ready to make some waves of your own? Richard demonstrates how to implement your own embedded Web server—from creating a base TCP/IP application to writing interactive HTML forms.

Web surfing may be an absorbing and potentially educational activity, but there's something about it that's just so...passive. Sometimes, you yearn to not only partake of the networked wonders of the world, but to add to them as well.

So, you take this opportunity to create your own Web page, complete with scanned images of your pets, a local map highlighting your favorite pizza parlors, and links to magnetic media duplicators.

But, the hit counter isn't incrementing quickly. And besides, the page is stored on some massive drive in some computer you've never seen before.

Being a hands-on sort of person, you're ready for the next step. It's time to put together your own embedded Web server from scratch.

Your own Web server can do a lot more than serve up text and GIFs. It can also provide a way to monitor and control an embedded system.

Since powerful Web browsers are given away free today, there's a great opportunity to add a graphical front end to control your embedded system and display status or supply control parameters in a user-friendly manner.

Fortunately, the protocol that describes the operations of a Web server is rather straightforward. The most recent version of the formal specification HTTP 1.1 is contained in RFC 2068. The preceding version—HTTP 1.0—is simpler to implement and widely supported.

The example I present here follows the earlier standard. But first, let me briefly summarize Web-server operation.

SERVER OPERATION

In a typical client/server system, the client establishes a connection with the server, submits a request to the server, interprets the server's response, and then sends further requests or closes the connection if it's no longer needed.

A Web browser is a client application that establishes a connection with a Web server, requests a resource from the server, reads the information that the server sends, displays it using the built-in formatting information, and then closes the connection.

If the page just loaded contains references to multimedia resources that haven't been loaded yet, then additional connections are established to read and display this information.

This action continues until all the resources on a Web page are retrieved. At this point, the system waits for you to click on a new resource, which leads to a server being contacted to request the new resource and the cycle starts anew.

To implement your own Web server, you need to create your own TCP/IP application that runs on top of a TCP/IP stack. The application establishes the connection, reads and writes data, and closes the connection using func-

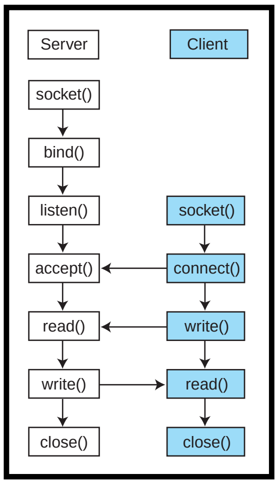


Figure 1—*When clients and servers are written using the BSD Sockets interface, these are the typical function calls made to transfer information.*

Listing 1— *This code implements a minimal Web server that can be set up to deliver ROMed Web pages. This example compiles and runs under Linux, and can be easily adapted for other environments.*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include "romfile.h"
#define HDR "HTTP/1.0 200 OK\r\n"
#define HDRHTML "Content-type: text/html\r\n"
#define HDRJPEG "Content-type: image/jpeg\r\n"
#define DEFAULTRESOURCE "index.html"
main(){
  char buf[500];  /* holds incoming request, outgoing response */
  char defaultresource[] = DEFAULTRESOURCE;
  char resourcename[80];    /* holds name of requested resource */
  char *filename;              /* pointer to requested resource */
  char *extension;  /* ptr to extension for requested resource */
  int remSize;              /* holds size of address structure */
  int retcod;                   /* general purpose return code */
  int s;                         /* handle to listening socket */
  int s2;                   /* handle to socket being serviced */
  long filelen;                  /* size of requested resource */
  struct sockaddr_in locAddr;  /* addr struct for local address */
  struct sockaddr_in remAddr; /* addr struct for remote address */
  RFHTYPE rfhandle;                      /* handle for rom file */
  printf("Sample Web Server v0.0\n");
  s = socket(AF_INET, SOCK_STREAM, 0);  /* create listening socket */
  if (s < 0) {
    printf("Error opening socket\n");
    return -1; }
  memset((void *) &locAddr, 0, sizeof(locAddr));
  locAddr.sin_family = AF_INET;
  locAddr.sin_addr.s_addr = htonl(INADDR_ANY);
  locAddr.sin_port = htons(80);
  retcod = bind(s, (struct sockaddr *) &locAddr, sizeof(locAddr));
  if (retcod < 0) {
    printf("Error binding socket\n");
    close(s);
    return -1; }
  retcod = listen(s, 5);
  if (retcod < 0) {
    printf("Error in listen\n");
    close(s);
    return -1; }
  while (1) {
    remSize = sizeof(remAddr);
    s2 = accept(s, (struct sockaddr *) &remAddr, &remSize);
    if (s2 < 0) {
      printf("Error in accept\n");
      close(s);
      return -1; }
    retcod = read(s2, buf, sizeof(buf));        /* read request */
    if (retcod >= 0)
      buf[retcod] = 0;          /* change to null terminatd str */
    printf("%s\n", buf);           /* display request for debug */
    write(s2, HDR, strlen(HDR));   /* write first response line */
    sscanf(buf, "GET %s", resourcename);       /* find resource */
    filename = defaultresource;  /* use this resource for deflt */
    if (strcmp("/", resourcename) != 0)/* is this default request? */
      filename = resourcename + 1; /* no, skip past initial '/' */
    extension = filename;                  /* isolate extension */
    while ((*extension) && (*extension != '.'))
     extension++;
    if (strcmp(".html", extension) == 0)           /* write type */
      write(s2, HDRHTML, strlen(HDRHTML));
    else if (strcmp(".jpg", extension) == 0)
      write(s2, HDRJPEG, strlen(HDRJPEG));
```

(continued)

tions provided by the stack. (Refer to "TCP/IP in Embedded Systems" [*INK* 79] for an overview of stacks in embedded systems.)

Although the RFCs suggest the general form and the capabilities to be supplied by the interface between a network application and TCP/IP stack, they don't fully specify the Application Program Interface (API).

A number of APIs have been established, but by far the most common is the BSD Sockets interface, which is provided by BSD releases of the Unix operating system. A close relative, the WinSock interface, is used for Windows networking applications.

I'll use the BSD Sockets interface to illustrate a sample Web-server application because it is well known and widely available. Of course, my code may need slight adaptation to work with other TCP/IP-stack implementations.

Figure 1 illustrates a typical sequence of BSD Sockets functions that are called by server and client applications in a network transfer. They act as a road map to the Web-server code in Listing 1.

GETTING A HANDLE ON THINGS

The Web server's first task is to indicate its interest in receiving TCP segments directed to port 80, which is the default port for an HTTP server. Under BSD Sockets, four function calls are made to set this up.

The first step allocates a socket for the server to use. A socket is a data structure that maintains information on a network connection.

The prototype for the function is:

```
int socket(int domain, int
   type, int protocol);
```

The first parameter is the communications domain, which in this case is `PF_INET`, indicating that I want to work with the Internet protocol family. Other domains can be specified for other communications families (e.g., ISO).

The next parameter is the socket type, which I specified as `SOCK_STREAM`, indicating that I want reliable bytestream service (i.e., TCP). `SOCK_DGRAM` would be specified for UDP service.

The final parameter can be used to further specify the protocol, but for Internet bytestream service, a 0 suffices. socket() returns a handle to the socket or –1 to indicate an error.

This seems like a roundabout way of indicating that you want a handle to a socket that talks TCP. However, the BSD Sockets interface is used for more than applications running over a TCP/IP stack.

There's a whole world of protocols—old, current, and yet to be defined—that can be coupled to an application through this interface. The protocol doesn't even need to be a network protocol. For example, the Unix domain protocol permits interprocess communication within the same system.

Once you have a handle to a socket, you need to specify the port at which you're listening for incoming information. This task is accomplished via a call to the bind() function:

```
int bind(int s, struct sockaddr
    *my_addr, int addrlen);
```

The first parameter is the handle to the socket that was returned earlier. The second parameter passes a pointer to a socket address structure that specifies the local IP address and port number for this connection.

For Internet addresses, the sockaddr_in structure is used, which contains fields for the address family, IP address, and port number. You clear the structure and then fill in these values before passing a pointer to the structure in the call to bind(). The port-number field is a two-byte value that should be expressed in network byte order, which is Big Endian.

To make the code portable, the utility function htons() translates between the host's native format and network byte order before saving the value in the structure. The constant INADDR_ANY indicates that this socket should accept connections from any of the system's network interfaces. This four-byte IP address also needs to be translated by the htonl() utility function to put it in network byte order before storing the value in the structure.

The last parameter in the call is simply the size of the sockaddr_in structure, another indication of the flexible design of the socket's interface. It returns –1 if there is an error.

Listing 1—*continued*

```
    rfhandle = romfileopen(filename);    /* open local resource */
    filelen = romfilelen(rfhandle);          /* determine size */
    sprintf(buf, "Content-length: %d\r\n\r\n", filelen);
    write(s2, buf, strlen(buf));                 /* write size */
    while (1) {                         /* write out resource */
      if ((retcod = romfileread(rfhandle, buf, sizeof(buf))) > 0)
        write(s2, buf, retcod);
      else
        break; }
    romfileclose(rfhandle);                  /* close resource */
    close(s2); }                           /* close connection */
  return 0; }
```

Now, I have a socket associated with a port. The next step is to put the socket into the listen state, so it's ready to service incoming requests for connections to port 80. This task is done with a call to listen:

```
int listen(int s, int backlog);
```

Here, I simply specify the socket and a backlog value, which indicates the number of connections that will be held in a queue awaiting service. An error is indicated by a –1 return value.

Finally, I make a call to accept a connection:

```
int accept(int s, struct sockaddr
    *addr, int *addrlen);
```

Here again, I pass a socket handle and then a pointer to an address structure, followed by a pointer to the size of the address structure.

In this case, the accept() function fills in the IP address and port number of the remote system that is establishing a connection on the socket in the address parameter. The application therefore knows a little about the remote system requesting services when the function returns.

The address-length parameter should contain a pointer to an integer with the length of the address structure. On return from accept(), this parameter contains the length of the address structure that was filled in.

For Internet protocols, this value doesn't change. The accept() function blocks until a connection is established and then returns a handle to a new socket associated with the client that connected.

The original socket continues to collect subsequent clients that want to connect to port 80. A return value of –1 indicates an error.

ITERATIVE VS. CONCURRENT

When a client establishes a connection with a server, the server creates a data structure that holds the state of the connection until the connection closes. The server then listens for and responds to client requests until one side or the other indicates that the connection should be closed.

What happens if another client contacts the server while the first client is being served? The outcome depends on the design of the server.

In an iterative server, the second client's requests are ignored until the connection with the first client closes.

In a concurrent server, the code that services a connection is set up as a task, and this task is launched every time a connection is established. So, a concurrent server can serve more than one connection at once, assuming that the system software supports multitasking.

For this demonstration, I take the approach of an iterative server. This approach isn't the most common for a Web server, but it will do, especially since I expect the server to only service one client at a time.

Other clients that request services are forced to wait until a previous request has been fulfilled. This scenario is often quite workable but doesn't make sense for a large-scale server.

Basic HTML

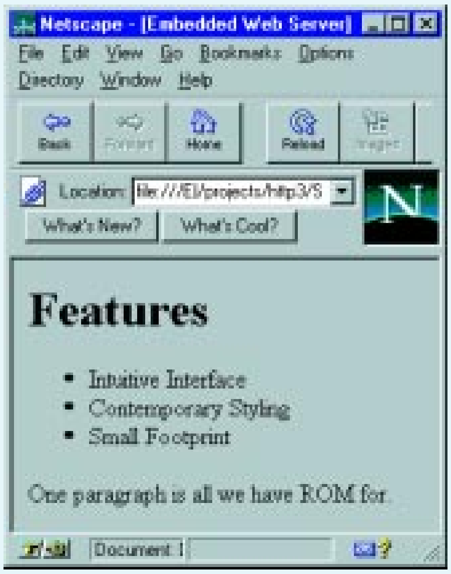
Hypertext Markup Language (HTML) is the formatting language that transforms plain text into the attractive Web pages that fill the Internet. Creating effective HTML pages for an embedded system requires an extra measure of skill because you need to make the most of a limited set of resources. We all know how graphic images can eat up memory, so it makes sense to keep GIF images to a minimum and make the most of the other formatting features.

The formatting information is included in the document via tags that appear between angled brackets (e.g., `<TT>`, which specifies a teletype-like font). The syntactic rules for HTML can be inferred by reviewing examples, and RFC 1866 can be consulted for the specifics of HTML V.2.0.

Much of the formatting is specified by a pairing of a start tag and an end tag, such as `<I>italic</I>`, where the slash indicates the end of italic font. Some tags stand alone though. For example, `<HR>` creates a horizontal line across the page.

Tags may also contain attributes which may further specify formatting information. Listing i shows the basic form of a document and when displayed appears as Photo i. In most cases, the browser treats all white space in the same way, allowing you to format the information so it is easier to follow.

The document is made up of `HEAD` and `BODY` sections. A `TITLE` must be present in the `HEAD` section. This title is typically displayed in a title bar on the Web browser and is also stored in a browser's list of saved links. The `BODY` section contains the contents of the page. Table i lists some formatting features that can liven up this section.


Netscape - [Embedded Web Server]
File Edit View Go Bookmarks Options Directory Window Help
Back Forward Home Reload Images
Location: file:///E|/projects/http3/S
What's New? What's Cool?
Document: Done

Features

- Intuitive Interface
- Contemporary Styling
- Small Footprint

One paragraph is all we have ROM for.

Photo i—*The code of Listing i produces this minimal Web page. It looks good as 109,208 pixels.*

Listing i—*A minimal Web page can be rather short. Here are 274 bytes worth.*

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
<HEAD>
<TITLE>Embedded Web Server</TITLE>
</HEAD>

<BODY>
<H1>Features</H1>
<UL>
  <LI>Intuitive Interface
  <LI>Contemporary Styling
  <LI>Small Footprint
</UL>
<P>One paragraph is all we have ROM for.
</BODY>

</HTML>
```

A number of utilities—commercial and otherwise—can be used to simplify the creation of HTML text. Most browsers have a View Source option that lets you to see how a particularly neat feature of a Web page has been implemented. Beware of incorporating nonstandard HTML into your ROM, though.

<H1>–<H6>	Headers of increasingly less emphasis
<P>	Starts new paragraph, leaving a blank space to separate from the previous paragraph. A </P> end tag is optional.
<PRE>	Preformatted text. Preserves line breaks in the original text. Usually, line breaks are ignored and the text is flowed to fit in the browser's window.
<UL>	Starts unordered list, typically presented as a series of bulleted items. Within this section, the <LI> tag starts a list item.
<OL>	Starts ordered list, similar to above, but with numerals
<EM>	Starts emphasis, often expressed as italics
<STRONG>	Starts strong emphasis, usually in bold
<BR>	Forces line break. An end tag is not needed.
<HR>	Inserts horizontal line
<IMG>	Inserts graphical image

Table i—*These common formatting tags help spice up the text on your pages.*

Now is the time for the server to `read()` a request from the server. The `read()` function is similar to a file read:

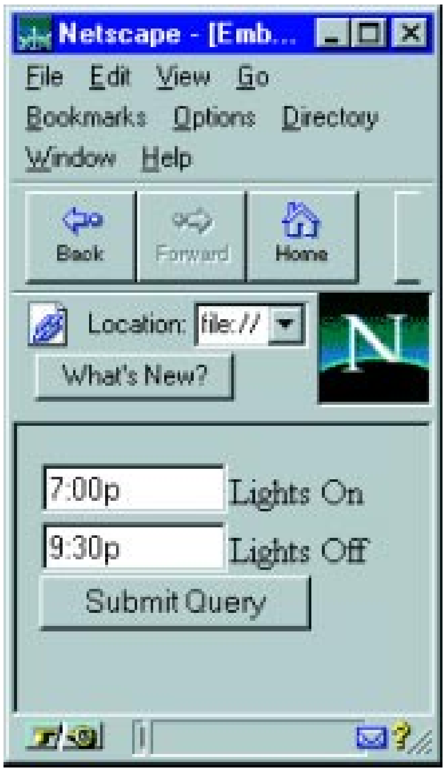
```
int read(int s, char *buf,
    int count);
```

The application specifies the socket handle, a pointer to a buffer that stores the incoming data, and the buffer size. When the function returns, the number of bytes that were read is returned.

This value may be less than the requested amount of information, and the application should continue to issue calls to read from the connection if the application-level transaction syntax indicates that more information is expected.

TCP acts like a pipeline delivering a bytestream, and the `read()` function delivers information as soon as it is available. However, the application developer should be aware that there may be more in the pipeline, especially if the buffer being read is large.

So, let's assume now that the server program is running on your Web server, which you set up with the IP address of 192.168.173.15. You fire up your favorite Web browser and request the URL <http://192.168.173.15/>.



7:00p Lights On
9:30p Lights Off
Submit Query

Photo 1—*Using HTML form tags, we achieve something very much like a traditional graphical user interface. Listing 4 contains the code to instruct the browser to generate this image.*

Listing 2—*When you pull up a simple page, your Web browser and server trade data. The request that the Web browser sends to the Web server* ***(a)*** *results in the Web server giving this response* ***(b)***.

a)
```
GET / HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.01(Win95;I)
Host: 192.168.173.15
Accept: image/gif, image/x-bitmap, image/jpeg, image/pjpeg, */*
```

b)
```
HTTP/1.0 200 OK
Date: Sun, 06 Nov 1994 08:49:37 GMT
Content-Type: text/html
Content-Length: 1523
<HTML>
<HEAD>
<TITLE> Sample Embedded Web Page </TITLE>
[the rest of the contents of the default page]
</HTML>
```

On the server end, the first buffer of data might look something like Listing 2a. This buffer is filled with the request message from the browser, the first line of which is called the request line. The next lines contain headers that provide additional details about the request.

The request line is of the form:

```
Method <SPACE> Resource <SPACE>
   HTTP-Version <CR><LF>
```

This request indicates that the Web browser would like for the server to GET the resource known as "/", and that this request is coming from an HTTP V.1.0 client.

At this point, the Web server should examine the requested resource and provide an appropriate response. Since the resource identified as "/" corresponds to the default Web page, that's the page that will be delivered.

The response might look like Listing 2b. Here, the first line is a status line, indicating that the server was able to locate the resource and is about to send it.

The next three lines are headers providing more information about the resource to be transferred. The first line is a time stamp, which indicates the time that the resource is being delivered by the server, followed by the type and length of the resource.

A blank line separates the end of the header fields from the contents of the resource itself, known as the Entity Body. See the sidebar "Basic HTML" for more details on the formatting and features of hypertext documents.

After this response is sent, the server should close the connection. Under HTTP 1.1, the default behavior is to leave the connection open to reduce the overhead of HTTP transfers.

FOLLOWING THE LINKS

The beautiful thing about Web pages is that they can be "deeply intertwingled," as Ted Nelson described the concept when first introducing the idea of hypertext in 1965. Within the body of the first default Web page that is delivered, a number of hyperlinks can be indicated by anchor tags:

```
<A HREF="newstats.htm"> The
   Latest Statistics </A>
```

The text "The Latest Statistics" is displayed as a hypertext link on a browser, and should the user point to this text and click the mouse, a fresh GET request will be sent by the browser, specifying the resource indicated in the HREF attribute (i.e., newstats.htm). The Web-browser function that interprets this request tries to look up this resource and deliver it as a fresh Web page to the browser.

FROM DISK TO ROM

The easy way to satisfy a resource request is to defer to a file system to look up and deliver the resource requested by the Web browser. Unfortunately, a file system isn't standard equipment with all embedded systems.

A reasonable facsimile, however, isn't hard to come by. Instead of storing information in a file on a disk, the same information can be compiled into large arrays of data that can be linked into the image in the embedded system's ROM.

To accomplish this, the example Web server includes two utility programs. The first utility, rfmake, converts a binary file into an array of data that is acceptable to a C compiler.

After all of the file images have been converted into C data arrays, these arrays are collected into a directory-like structure using the romdir utility. The output from romdir is another C structure that acts as a directory to all the ROMed file images.

When the output from the utilities is linked in with the Web server, this data can be accessed through a set of routines that resemble file I/O functions (see Listing 3). Another advantage of this system is that if a disk-based file system is available, then the server program can easily be adapted to use real file I/O functions.

INTERACTIVITY

Although it may be great fun to have your embedded Web server dishing up those pages with the best of them, things get really interesting when you add interactive functions.

To accomplish this, the server needs to be implemented with a combination of appropriately written HTML documents and functions that can interpret interactive requests, such as image maps and forms.

An image map is usually included in the body of a page as part of an anchor, such as:

```
<A HREF="panelmap"><IMG ISMAP
   SRC="panel.gif"></A>
```

The ISMAP attribute in the IMG tag indicates that this image should be treated in a special way. The contents of panel.gif will be displayed as a clickable image on the browser, and when the user clicks within this area, a request is sent to the server specifying the offset into the bitmap where the mouse click occurred.

The server may see a request for `panelmap?12,4`, which indicates that the user clicked inside this area at a point 12 pixels from the left edge and 4 pixels down from the top of the image.

By supplying a routine on the server that interprets location information, the system recognizes the object a user points to and responds appropriately. So, clicking on a darkened window in an image of a house might command a home controller to turn on the lighting in this area.

To obtain full style points, the server could update the image with one that shows the window being lit.

Form submission is another technique that can be used to send information to a server, providing many of the familiar dialog box tools, such as text boxes, check boxes, radio buttons, and lists. These features start with a `FORM` tag in an HTML document.

Listing 4 presents an example of a form with two text boxes and a submit button. The second line refers to the first text box. The fourth line in the form definition defines the submit button, which is needed whenever there is more than one input in a form.

The user can type any text string into the boxes for the time at which the home controller should turn the lights on and off. When the user clicks on the submit button, the home-control Web server is sent the request:

```
control?LiteOn="7:00p";
      LiteOff="9:30p"
```

Photo 1 displays this interface.

Again, an appropriate routine needs to be provided on the server to interpret this information and take appropriate action. The server should also generate a page that tells the user that the command was successfully processed.

ERROR HANDLING

In a number of situations, the Web server may not be able to respond to a request. The user may have typed in a request for a resource the server has never heard of, or the user may have used an interactive control to submit information the server won't accept.

Listing 3—*These function prototypes form the link between the Web server and a file system. They have been defined to make it easy to use either a disk- or memory-based file system.*

```
RFHTYPE romfileopen(char *romfilename);
long romfilelen(RFHTYPE handle);
int romfileread(RFHTYPE handle, char *buf, int bufsize);
int romfileclose(RFHTYPE handle);
```

Listing 4—*In the second line of this section of HTML, `TYPE=TEXT` specifies that the form's input will come as a text box, `NAME="LiteOn"` is the variable name associated with the input, `SIZE=10` entails that 10 spaces will be available in text box, and `Lights On` is the label to be displayed beside the box on the form. See Photo 1 for the end result.*

```
<FORM ACTION="control" METHOD="POST">
  <INPUT TYPE=TEXT NAME="LiteOn" SIZE=10>Lights On
  <INPUT TYPE=TEXT NAME="LiteOff" SIZE=10>Lights Off
  <INPUT TYPE="submit" NAME="Save">
</FORM>
```

When everything is in order, the server sends the string "200 OK" response as part of the status line for the response.

Additional three-digit codes that are suitable for other conditions that might arise are divided into a series of related responses.

Codes in the 100 series are informational, the 200 series indicates success, the 300 series indicates a need for redirection, and the 400 and 500 series are for client and server errors, respectively.

For example, a request for an unknown resource could generate a status line containing "404 Not Found". In the response, the Entity Body could contain HTML text to further explain that the resource couldn't be found on this server.

ON YOUR OWN

Of course, this discussion just begins to describe the sorts of capabilities that might be implemented on an embedded Web server.

Additional information on HTML and HTTP is available online and from a library of ever thicker books, and there's a variety of software tools that do everything from verifying the syntax of an HTML page to providing a turn-key server.

Now you can sleep well, knowing your home page could be served up from a networked controller living in a shoebox under your bed.

Richard Ames is a staff engineer at U.S. Software. He finds that working with networking software allows him to gather more computers around him than the average engineer. You may reach Richard at richard@ussw.com.

SOFTWARE

The `rfmake` and `romdir` utilities discussed in this article are available on the Circuit Cellar Web site.

REFERENCES

S. Berners-Lee, R. Fielding, and H. Nielsen, *Hypertext Transfer Protocol—HTTP/1.0*, RFC 1945, 1996.

T. Berners-Lee and D. Connolly, *Hypertext Markup Language—2.0*, RFC 1866, 1995.

R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee, *Hypertext Transfer Protocol—HTTP/1.1*, RFC 2068, 1997.

T.H. Nelson, *Computer Lib/Dream Machines*, Tempus Books, Redmond, WA, 1977 (Reprinted by Microsoft Press).

D. Raggett, *HTML Tables*, RFC 1942, 1996.

W.R. Stevens, *Unix Network Programming*, **1**, Prentice-Hall, Englewood Cliffs, NJ, 1998.

I R S

404 Very Useful
405 Moderately Useful
406 Not Useful

FEATURE ARTICLE

Bill Payne

Integrating Windows NT 4.0 into a TCP/IP Environment

Bill needs to connect Windows NT stations into an existing TCP/IP network, but Microsoft gave the Domain Name Server its own personal twist. After reviewing the protocols, he shows how to get Windows and Unix to talk.

I was recently involved in a large-scale project in which I integrated Windows NT 4.0 servers and workstations into an existing TCP/IP network. The network comprised multiple Unix, IBM mainframe, PC, and DEC Alpha-based hosts.

In addition, approximately 3000 clients were to be upgraded from Windows 3.11 to Windows 95 and Windows NT 4.0 Workstation.

The network used the Dynamic Host Configuration Protocol (DHCP), an open industry standard designed to reduce the complexity of managing a TCP/IP network.

Every computer and resource on a TCP/IP network must be given a unique IP address and computer name. DHCP assigns a client an IP address from a pool of addresses when it starts up. This enables the IP addresses to be managed from one central point on the network.

The network also uses the Domain Name System (DNS) for name-to-IP address resolution. Cisco routers were implemented to segment the network into smaller manageable sections.

After the initial installation of the Windows NT 4.0 servers, we began having problems with the address resolution process on various clients throughout the network. And, the problem wasn't DHCP. It worked properly, assigning IP addresses to client workstations as requested.

But, if we used a command that tried to resolve a name to an IP address, we had problems. Sometimes the name would resolve, but not necessarily.

We also began seeing a lot of traffic on the segments that had been created with the routers. One thing stood out—if the name to be resolved was on the same segment as the client issuing the query, it worked.

After many hours of going through traces from a network sniffer, the problem was found and corrected. It turns out that even though DNS is a standard, certain companies give it their own personal twist.

Such is the case when interconnecting devices that use the enhancements developed by Microsoft. The problems do not manifest themselves until routers are added to a network.

Routers in general do not forward broadcast messages. The reason for using routers in the first place is to segment the network and to isolate traffic to separate segments.

Windows NT relies on a process called Windows Internet Name Service (WINS) for name resolution. This service augments the traditional DNS service with dynamic name registration capabilities and uses the NetBIOS protocol encapsulated in TCP/IP for communication between nodes.

To help you gain an understanding of the problem, I first present the basic concepts of TCP/IP, DNS, and WINS as implemented by Microsoft. Once you have the nuances down, interconnecting systems using TCP/IP in a routed network becomes fairly straightforward.

TCP/IP

The Transmission Control Protocol/Internet Protocol (TCP/IP) suite is a standard set of networking protocols. It was originally developed by the Department of Defense and is sometimes referred to as the DOD model.

A protocol is an agreed-to set of rules governing the communication of data between two parties. You could compare it to two people trying to communicate with each other. They must both speak the same language or no transfer of information occurs.

TCP/IP is a scalable, robust, client-server networking protocol. It connects dissimilar systems and is the basis of the global Internet.

The TCP/IP family comprises four layers—network interface, Internet, transport (host-to-host), and application. These layers map to one or more layers of the International Standards Organization (ISO) seven-layer Open Systems Interconnection (OSI) model as shown in Figure 1.

Each layer of the TCP/IP model contains defined protocols that dictate how computers communicate and connect. The most common are the TCP, IP, User Datagram Protocol (UDP), Address Resolution Protocol (ARP), and Internet Control Message Protocol (ICMP). The breakdown of the various protocols is shown in Figure 2.

TCP establishes a virtual circuit between hosts, providing a reliable connection for exchanging data. All transmitted packets are sequenced and acknowledged by the receiving host.

If a packet is corrupted or lost during transmission, this protocol retransmits the faulty packet. The protocol is used by applications such as telnet, file transfer protocol (FTP), client-server applications, and E-mail.

UDP provides an unreliable, connectionless delivery service. It doesn't guarantee delivery or correct sequencing of the delivered packets.

Therefore, applications can exchange data without the overhead of acknowledging packets and maintaining a virtual circuit. UDP is used by applications that rely on broadcasts to multiple receivers.

Trivial FTP (TFTP) and the Unix Network File System (NFS) use this transport system. Because it is a broadcast message, most routers won't pass these messages between different segments on a network.

IP provides packet delivery for all other protocols within the suite. It is used primarily to route packets between different hosts. IP finds the shortest path between hosts, fragmenting the data into packets and then reassembling the packets into data.

ARP functions in a support role to the TCP/IP suite. It translates a remote host's IP address into a physical address.

ICMP lets systems share status and error information in much the same way as ARP. As an example, the `ping` program uses this protocol to determine whether a route exists to a particular IP address.

DOMAIN NAME SYSTEM

The Domain Name System (DNS) resolves host names to IP addresses. It is a distributed database that is structured as an inverted tree.

DNS was developed in the early 1980s to solve problems that resulted from the dramatic rise in the number of hosts on the Internet. The specifications for the DNS are defined in RFC 1034 and RFC 1035.

DNS relies on a static mapping of hosts to IP addresses. However, some confusion occurs from the term "host."

In this context, all computers with IP address are referred to as hosts. They may be known as servers and client workstations, but to DNS they are only hosts. The domain name refers to the computer's position in the hierarchical tree relative to the parent domain. The full name for a domain is constructed by listing all of the labels on the path from the domain to the root.

DNS computer names consist of two parts—the host name and the domain name. No second-level domain name can exceed 12 characters.

When combined, these names form the fully qualified domain name (FQDN). Each FQDN has a maximum length limitation of 255 characters. The FQDN is not case sensitive.

DNS is further divided into partitions, referred to as zones or subdomains. A zone begins at a specified domain and extends downward until either an end node is reached or another subzone begins. These zones represent the logical divisions of the Internet.

For example, my server's FQDN is kramerkent.com. kramerkent is the name of the zone which is registered with the InterNIC. It is located under the com domain of the hierarchical tree.

The names and IP addresses for all hosts in a zone are maintained on a single server referred to as the master server for the domain. The information on this server is the authoritative database for that zone.

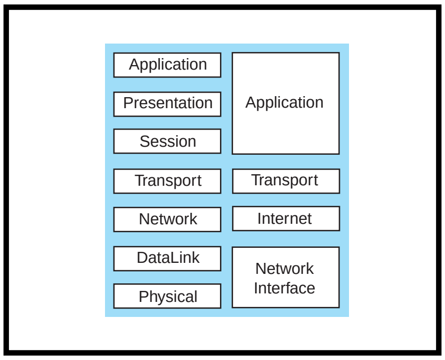


Figure 1—*This figure shows the mapping of the TCP/IP model to the seven-layer OSI model. The Application and Network Interface layers map to multiple layers of the OSI model.*

The database includes the names and addresses of all IP hosts within the domain, the names of all subzones and the addresses of the name servers for those zones, and the addresses of the name servers for the root domain. These addresses provide the necessary links between your domain and the existing DNS hierarchy.

Keep in mind that all of these links are static. If you change the IP address of a host, it must be manually changed on the DNS master server for your zone.

The primary task for DNS is to resolve user-friendly names to IP addresses. The name-resolution process is performed from left to right. If the local name server does not have the address record for the requested name, it queries other name servers on behalf of the resolver. The name resolution process consists of three key concepts: recursive resolution, iterative resolution, and caching.

A recursive resolution request is typically passed from the resolver to the local name server. The local name server processes the query and returns a complete answer to the resolver. The local name server contacts other name servers if necessary to resolve the name. It doesn't return a pointer to another name server, which enables the resolver to be small and simple. The workload is placed on the local name server.

Iterative resolution requests are passed to other name servers if the local name server cannot fully resolve the query. The contacted name server is instructed to only attempt to resolve the name locally. If it cannot, it returns a pointer to the next name server in the DNS tree.

This process continues to walk up the hierarchical tree until the primary master name server for the zone is reached. An error to the requester is returned if the name cannot be resolved at this level.

With caching, the local name server keeps a copy of the resolver request answer in local memory. This speeds DNS performance and eases the burden on other name servers.

The local name server first checks its static mapping information when a resolver request arrives. If the answer is not found, the cache is then checked for either an answer or a pointer to the name server containing the answer. This process reduces the number of iterative resolution requests for the name servers on the network.

WINS

Windows Internet Name Service (WINS) is a Microsoft-developed naming service. Microsoft recommends that their naming service be used in conjunction with DNS. Unlike the static naming which is inherent in DNS, WINS offers dynamic naming capability. To understand how WINS works, an understanding of both NetBIOS and NetBEUI is necessary.

All Microsoft Windows networking components rely on NetBIOS, which is a software interface and naming convention, not a protocol. It is a set of Application Programming Interfaces (APIs) which lets applications request services from lower-level network processes.

All computers and resources within a Microsoft networking environment must be assigned a unique NetBIOS name, which cannot exceed 16 characters. Microsoft allows the first 15 characters to be specified by the user or administrator. The sixteenth character of the name is reserved to indicate the resource type. Since it is an 8-bit field, there are 256 resource types available.

These device names are stored within the NetBIOS namespace database. This database is designed as a flat, single-level structure.

A resource is dynamically registered with the database when it starts. This occurs when a computer boots, a service on a server starts, or a user logs on to a resource. Names can be registered as either a single owner or as a group composed of multiple owners.

In 1985, IBM developed a protocol named NetBIOS Extended User Interface

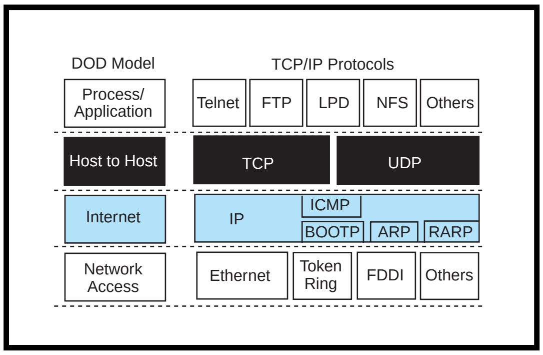


Figure 2—*Each layer within the DOD model is broken down into individual protocols. The protocols in the Transport (Host to Host) layer and the Internet layer are the ones of interest in this article.*

(NetBEUI) for programs designed around the NetBIOS APIs. This small protocol does not have a networking layer, so it is not routable and cannot be passed between segments connected by routers.

NetBEUI was originally designed for department-sized LANs ranging from 20 to 200 computers. It supports both connection-oriented and connectionless communications.

Its greatest strengths are that it is self-configuring and self-tunable. It requires a very small memory overhead in the client and provides for reasonable error detection and correction.

The NetBEUI protocol is the glue that provides interoperability with older networking systems, such as Microsoft LAN Manager and Windows for Workgroups 3.11. When you set up shared drives and folders under Windows 3.11, 95, and NT, you are using the NetBEUI protocol.

Microsoft extended the reach of the NetBIOS programming interface with the NT products to include the world of client/server applications in a WAN environment. This was accomplished by encapsulating the NetBIOS requests in TCP/IP packets. This is known as NetBT or NetBIOS over TCP/IP.

Earlier Microsoft implementations of the NetBT protocol relied on broadcasts, local cache, or an LMHOSTS file for locating resources on the network.

In Windows NT 3.5*x* and 4.0, a NetBIOS name server was implemented called the WINS server. It lets programs query the DNS namespace by appending user-configurable domain suffixes to a NetBIOS name.

NetBT uses a session layer network service to perform name-to-IP mapping for name resolution. There are four basic modes that determine how network resources are identified and accessed. The name resolution order depends on the mode type and computer configuration.

The B-node mode uses broadcast messages to resolve NetBIOS names to IP addresses. This is the default mode type for a Windows resource on a network.

The P-node mode uses point-to-point communications with a WINS name server for name registration and resolution.

The M-node mode first registers with the name server using broadcasts before trying to resolve names by using the B-node mode. If this fails, it switches to the P-node mode to resolve names.

The fourth mode—the H-node—uses the WINS name server for both name registration and resolution. If the name server cannot be located, it switches back to the B-node mode.

It then continues to poll for a name server and switches back to the P-node mode when one is located. WINS clients (i.e., Windows NT 3.5*x* and 4.0 workstations and servers) are configured as H-node devices by default.

PUTTING IT ALL TOGETHER

Now that you have a basic understanding of the mechanisms involved, let me explain how DNS name resolution works in conjunction with WINS name resolution. This will give the foundation needed to understand the original problem and see what solutions are necessary to run Windows NT in a mixed network environment.

For this discussion, let's assume that Circuit Cellar is running a Windows NT 4.0 Server as their Primary

Domain Controller (PDC). This server also has the DNS and WINS services running on it.

Connected to this server are 20 workstations running a combination of Windows 95 and Windows NT 4.0 Workstation. I want to look specifically at communication between hosts on this simple network.

The resolution process is started when a host issues a query to resolve a name to an IP address. It can be as simple as issuing the `ping` command directed at an address such as janice.circuitcellar.com (`ping janice.circuitcellar.com`).

If the name to be resolved is greater than 16 characters or contains a ".", it is passed on to the DNS name server. If the address record for the host janice is found in the DNS database, it's returned to the host which sent the query.

If the name is not found in the DNS database or the name is a valid NetBIOS computer name, the WINS client performs the NetBIOS computer name-to-IP address resolution using NetBT and WINS.

The first thing the WINS client does is to determine the mode type it is operating as: H-node, P-node, M-node, or B-node. The mode type is defined on a Windows NT computer through the TCP/IP configuration settings. Each mode type processes the query in a different manner.

The first step is common to all four node types. The local NetBIOS name cache is searched for the computer name. If found, it returns the IP address to the host which issued the query. The following steps are executed if the local cache doesn't contain the NetBIOS computer name.

As the first step in the H-node mode, the client issues a query to the local WINS server. If the name is found, the IP address is returned to the requesting host.

The client then uses local broadcasts to locate the NetBIOS computer name. If the name is found, the IP address is returned to the requesting host.

As the third step, the client checks the local LMHOSTS file for the NetBIOS computer name. This occurs only if the proper option has been selected in the TCP/IP properties page. If found, the IP address is returned.

The client then checks the local HOSTS file for the NetBIOS computer name. This also only occurs if the proper option has been selected in the TCP/IP properties page. If found, the IP address is returned.

The client ends by querying the DNS server. If the NetBIOS computer name is in the database, the IP address is returned to the requesting host.

Resolving network addresses goes back to the basics of Windows networking protocol—something originally designed by IBM in 1985.

In P-node mode, the client first issues a query to the local WINS server. If the name is found, the IP address is returned to the requesting host.

Next, the third, fourth, and fifth steps of the H-node mode protocol are executed.

In M-node mode, the client uses local broadcasts to locate the NetBIOS computer name. If the name is found, the IP address is returned to the requesting host.

The client then issues a query to the local WINS server. If the name is found, the IP address is returned to the requesting host. Steps 3, 4, and 5 are then executed.

And finally, in B-node mode, the client uses local broadcasts to locate the NetBIOS computer name. If the name is found, the IP address is returned to the requesting host, and the third, fourth, and fifth steps are executed.

PROBLEM RESOLVED

In a traditional TCP/IP environment (e.g., Unix), the DNS name-resolution process is fairly straightforward. The host addresses to be resolved are in the HOSTS file on the DNS server.

Networks containing routers do not present a problem to a system based on this older technology. The only name registration that occurs if the network uses DHCP is assigning IP addresses to clients. The drawback to this is that DNS by its nature is static.

The implementation of the Microsoft WINS server for NetBIOS computer name and IP address resolution provides the ability to dynamically modify the information in the translation database.

Windows NT uses UDP broadcasts as the protocol for NetBIOS (NetBT) name resolution. Unfortunately, UDP broadcast packets are not usually forwarded by routers between network segments.

In addition, the WINS name-resolution process can generate a considerable amount of burst traffic on a network segment. WINS name resolution in a routed environment requires either a name server of some type on each segment of the network or the use of static database files.

In the case of the company I did the Windows NT integration for, WINS name servers were established on each segment of the routed network.

Once the problem is understood, the answer is fairly simple. The problem is in understanding the processes and how they interact with each other. ■

Bill Payne has many years' experience as a digital design engineer. He is a Novell Master CNE, Novell Certified Instructor, and a Microsoft Certified Trainer for the NT 4.0 Products. He may be reached at bpayne@kramer-kent.com.

REFERENCES

Microsoft, *Microsoft Windows NT Workstation Resource Kit*, Redmond, WA, 1996.

Microsoft, *Microsoft Windows NT Server Networking Guide*, Redmond, WA, 1996.

Inside TCP/IP, Second Ed., New Riders, Indianapolis, IN, 1995.

Novell, *DNS and FTP Server Installation and Configuration*, Orem, UT, 1996–97.

I R S

407 Very Useful
408 Moderately Useful
409 Not Useful

Photo courtesy of Spyglass, Inc.

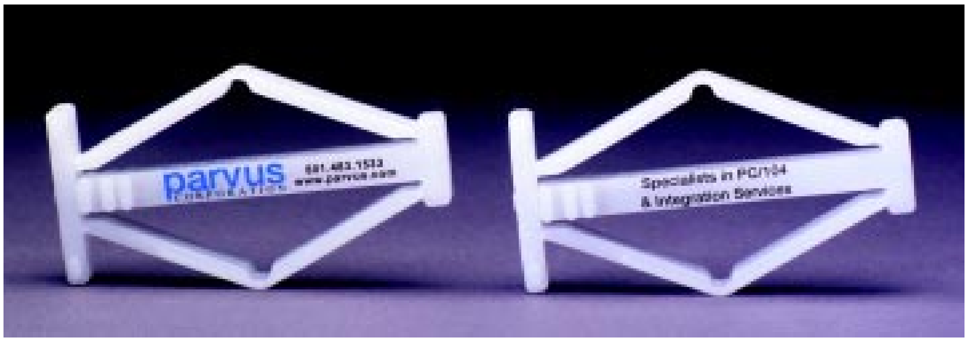
PC/104 EXTRACTION TOOL

The **X-Tool**, a PC/104 extraction tool, is designed to separate PC/104 cards from each other or from single-board computers without bending or breaking header pins. The X-Tool can be used to quickly disconnect components without damage to the pins.

Simply place the tool between the boards and squeeze, and the boards easily detach from the stack. Single-board computer components can also be separated with this tool. Snap the single-board computer adapter to the extraction tool. Vertically position the adapter next to the board and gently squeeze to remove the board. The X-Tool can be used with static-sensitive devices and will not demagnetize sensitive materials or documents.

The X-Tool retails for **$19.95**.

parvus Corp.
396 W. Ironwood Dr.
Salt Lake City, UT 84115
(801) 483-1533
Fax: (801) 483-1523
www.parvus.com **#510**



EMBEDDED PC SYSTEM

The CoreModule/P5*i* is a PC/104-*Plus*-compliant module that contains the functions of an entire Pentium-based embedded-PC system. The module includes a 133-MHz voltage reduction technology (VRT) Pentium processor, up to 64-MB system DRAM, and a battery-backed real-time clock. It also features peripheral interfaces for serial (two 16550 buffered UARTs), parallel (IEEE-1284 EPP/ECP), keyboard, speaker, floppy, IDE, and USB. A built-in bootable read/write flash disk (up to 16 MB) enables stand-alone operation as a self-contained embedded computer in many applications. Two system expansion buses (ISA and PCI) facilitate interfacing with application-specific custom electronics or off-the-shelf PC/104 and PC/104-*Plus* function modules.

The CoreModule/P5*i* is designed for mobile and portable embedded applications. Its environmental specs include an extended operating temperature range of –40 to +85°C, MIL-STD-202F shock and vibration specs of 50 and 12 Gs, respectively, and compliance with European CE Mark requirements for EMI, EMC, and ESD. Hardware and software enhancements include a ruggedized embedded-PC BIOS, watchdog timer, battery-free boot (to enable use without battery-backed "CMOS SETUP" or to boot despite dead battery), serial program loader, serial console, and fail-safe boot.

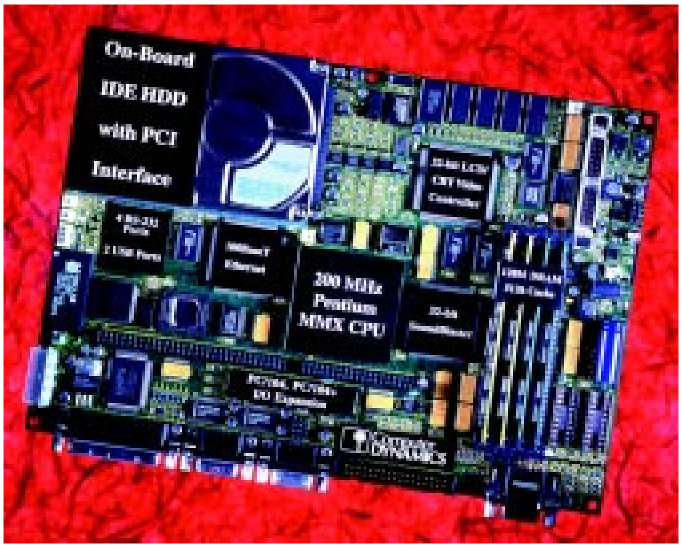
Extensive power management in both hardware and software results in extremely low power operation. To support the low voltage (3.3 and 2.9 VDC) requirements of the VRT Pentium processor and core logic, a highly efficient DC-to-DC converter is built into the CoreModule/P5*i*, resulting in single-supply (+5 VDC) system operation and minimal power consumption. A temperature sensor monitors operating temperature, and can trigger a slowdown of system clocks if needed.

The CoreModule/P5*i* sells for **$875**.

Ampro Computers, Inc.
4757 Hellyer Ave.
San Jose, CA 95138.
(408) 360-0200
Fax: (408) 360-0220
www.ampro.com **#511**

Nouveau PC

edited by Harv Weiner



SINGLE-BOARD COMPUTER

The **SBC-MaX** is a small form factor, industrial-strength, SBC with the functionality of a desktop MMX machine. It features an MMX Pentium CPU to 200 MHz, a 32-bit Chips and Technologies 65555 advanced video controller 100BaseT Ethernet, 16-bit Sound Blaster audio, and two USB ports. Up to 128 MB of synchronous DRAM and 512K level-2 cache are accommodated. A hard disk drive can be mounted directly onto the board, and the two-channel IDE hard-disk interface supports up to four IDE drives. The floppy-disk controller supports two 1.44- or 2.88-MB floppy drives. PC/104 and ISA-bus expansion are also provided.

Other I/O features include four RS-232 serial ports, one printer port, and keyboard, PS/2 mouse, speaker, and IrDA interfaces. The board measures 9.625″ × 6.875″ and is rated at 0–60°C. Power requirements are less than 20 W.

The 65555 video controller provides flat-panel support, especially for the newer passive color LCDs. This chip includes up to 4-MB video memory for maximum color depth in all resolutions and operating systems. Its Temporal Modulated Energy Distribution (TMED) technology enables the display of 16.7 million colors on STN (passive) LCDs without dithering. The 65555 also supports NTSC overlay capabilities on flat panels, a feature useful for displaying a video window over graphics or full-screen video.

The SBC-MaX SBC with a 200-MHz MMX Pentium CPU and 4 MB of DRAM sells for **$1290** in quantities of 100.

Computer Dynamics
7640 Pelham Rd.
Greenville, SC 29615
(864) 627-8800
Fax: (864) 675-0106
www.cdynamics.com

#512

WEB SERVER FOR EMBEDDED SYSTEMS

The **OSE Web Server** enables you to interface an embedded device to a Web browser via the Internet. It is fully compatible with all standard browsers, such as Netscape and Internet Explorer, and it runs on top of the OSE INET TCP/IP stack or any standard third-party TCP/IP stack.

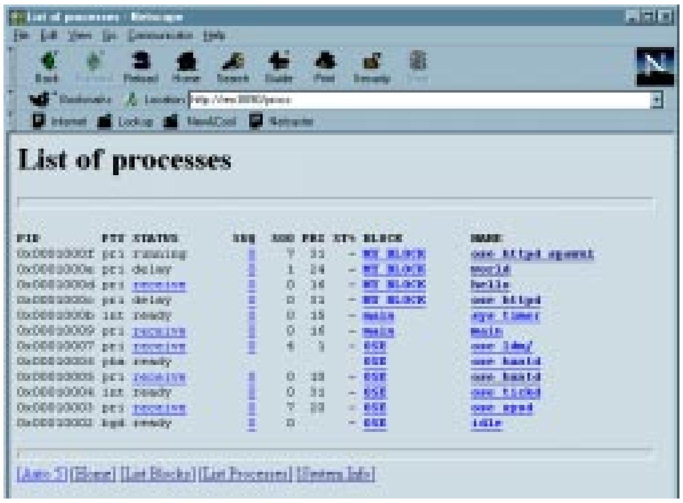
The OSE Web Server generates Web pages that are graphically displayed in a Web browser. Three complementing methods are supported—generating HTML pages dynamically, using a Web-page compiler, or managing with a file system. So, software engineers can use readily available and inexpensive Web browsers for flexible and powerful analysis, debug, and management of embedded devices.

Generating HTML pages dynamically allows continuous run-time information from the embedded system to be displayed in the browser (e.g., during debugging or run-time supervision of a product in the field). The Web-page compiler enables files in any format (e.g., JPG, PDF, GIF, TIFF, ASCII, HTML, etc.) to be displayed, including Java applets. This method means the compiled files can be viewed in a tree structure as in a virtual file system but without the need for an actual file system. If a standard file system is needed, OSE's embedded file system permits file upload and download via FTP. All files can be displayed and managed in a Web browser.

The OSE Web Server occupies 5 KB of ROM on a 68k target or 7 KB on a PPC target. Prices start at **$3000**.

Enea OSE Systems Inc.
5949 Sherry Ln.,
Ste. 1710
Dallas, TX 75225
(214) 346-9339
Fax: (214) 346-9344
www.enea.com

#513



List of processes

```
PID         PTY STATUS    SIG USE PRI STS BLOCK     NAME
0x0001000f  pri running   [illegible]  7  31  -  MY BLOCK  ose_httpd_agent
0x0001000e  pri delay     [illegible]  1  24  -  MY BLOCK  world
0x0001000d  pri receive   [illegible]  0  16  -  MY BLOCK  hello
0x0001000c  pri delay     [illegible]  0  31  -  MY BLOCK  ose_httpd
0x0001000b  int ready     [illegible]  0  15  -  main      sys_timer
0x00010009  pri receive   [illegible]  0  16  -  main      main
0x00010007  pri receive   [illegible]  4   1  -  OSE       ose_ldm/
0x00010006  phm ready                            OSE       ose_hasld
0x00010005  pri receive   [illegible]  0  15  -  OSE       ose_hasld
0x00010004  int ready     [illegible]  0  31  -  OSE       ose_tickd
0x00010003  pri receive   [illegible]  7  23  -  OSE       ose_spsd
0x00010002  bgd ready     [illegible]  0      -  OSE       idle
```

[Auto 5] [Home] [List Blocks] [List Processes] [System Info]

Nouveau PC

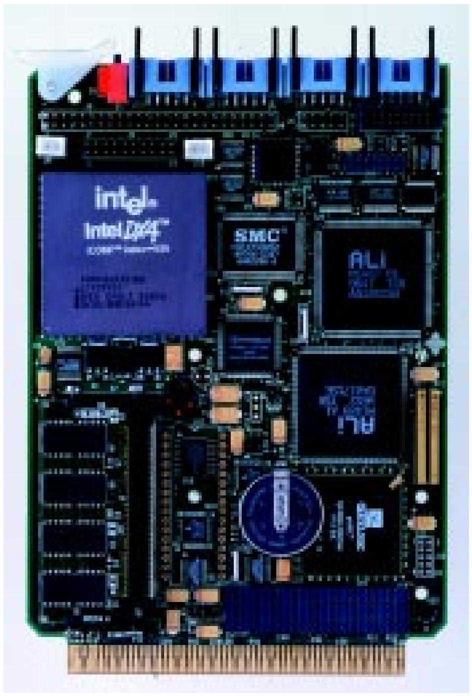
SINGLE-BOARD COMPUTER

Ziatech's **ZT 8907 Single Board 486/PCI Computer** features a range of '486 processor options (25–100 MHz) and a feature set designed to accommodate embedded applications. Based on the small (4.5″ × 6.5″), rugged STD 32 computer standard, the ZT 8907 includes up to 4 MB of flash memory, up to 32 MB of DRAM, two RS-232 serial ports, a printer port, 24 points of DIO, six counter/timers, and a real-time clock. The computer is PC software compatible and runs MS-DOS, the major Windows environments, and real-time operating systems such as QNX and VxWorks.

The ZT 8907 can operate as a single CPU or as one of several CPUs in Ziatech's multiprocessing system, the STD 32 Star System. The board is equipped with an industrial BIOS, which supports many embedded control features and multiprocessing. In addition, the ZT 8907 provides a Local-bus expansion connector that enables the system designer to add PCI peripherals such as SVGA, flat panel, or 100-Mb Ethernet interfaces. A local IDE hard drive is another option.

The ZT 8907 Single Board 486/PCI Computer sells for **$1270** in single quantities.

Ziatech Corp.
1050 Southwood Dr. • San Luis Obispo, CA 93401
(805) 541-0488 • Fax: (805) 541-5088
www.ziatech.com **#514**



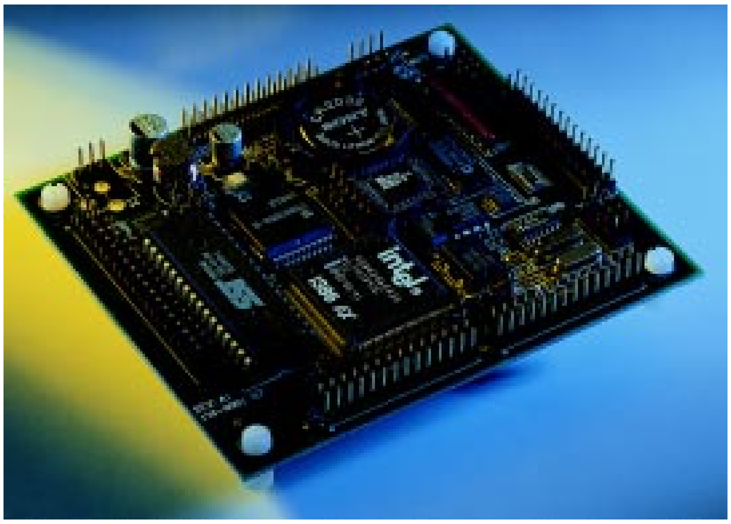
EMBEDDED CONTROLLER

The **FlashLite 386EX** is a single-board computer that's ideal for control applications. The DOS-based board features a 32-bit processor running at 25 MHz in protected mode as well as complete PC serial-port compatibility. The two serial ports can be configured as two RS-232 ports or as one RS-232 and one RS-485 port. The controller has 24 parallel I/O lines, two DMA channels, three counter/timers, and three available interrupt lines, providing a powerful platform for developing embedded DOS applications.

An onboard switching power supply accepts 7–34 VDC, converting the input to 5 VDC to provide power for the FlashLite 386EX and optional subsystems. A battery-backed RTC (clock/calendar), IBM-PC speaker, and watchdog timer are also included.

Applications can be developed using Borland C/C++, Microsoft QuickC, QuickBASIC, or other DOS development tools. The board has 512-KB SRAM, 512-KB flash memory, and a socket for another 512-KB flash memory, RAM, or EPROM. Developers can easily upload compiled code through one of the serial ports.

The FlashLite 386EX comes with a user manual and schematic and is priced at **$279**. A Developer's Kit that includes the FlashLite EX, an AC adapter, programming and port cables, utilities disk, manual, and schematic is also available for **$349**.



JK Microsystems, Inc.
1275 Yuba Ave.
San Pablo, CA 94806
(510) 236-1151
Fax: (510) 236-2999
www.jkmicro.com

#515

Nouveau PC

Dan Johnson

Converting PC GUIs for NonPC Devices

Sure, it's easy to create Web GUIs for your PC, but what about devices that connect to the Internet—TVs, PDAs, set-top boxes, or smart phones? Dan shows how Prism distills complex Web pages for one-bit 200 × 200-pixel displays.

As the Web evolves, it is becoming a medium accessed not only by PCs, but also a wide range of nonPC devices. Televisions, set-top boxes, smart phones, PDAs—they're all connecting to the Internet.

Perhaps the most significant challenge presented by these new devices is how to display content, originally created to be viewed on a PC, on the large variety of often limited displays found on these devices.

A computer may feature a 24-bit, 21″ SVGA screen, but a typical cell phone features a one-bit 200 × 200-pixel display. The heavily formatted content initially designed for the PC may take too long to download or be illegible on many nonPC devices. However, the success of these devices primarily depends on their ability to access and display meaningful content for their users.

Although it's possible for Web-site developers to create special versions of their sites for different devices (this is now done for different PC Web browsers), such a trend is unlikely at this time. Creating multiple versions of one Web site for different nonPC devices is a complex, time-consuming process. Many content providers aren't ready to make that investment yet.

So, we're left with a classic chicken-and-egg dilemma. Device manufacturers need quality content before they can sell large quantities of their devices, and content providers don't want to develop specialized content until large numbers of devices are sold.

To meet current needs, Spyglass Prism converts existing Web content for a variety of devices on-the-fly. Content providers can maintain a single version of their content on a server for use with all devices without any proprietary HTML tags or resorting to entirely new markup languages. And, users of nonPC devices gain faster access to Web content that looks good on their devices.

SERVER-BASED TECHNOLOGY

In essence, Spyglass Prism acts as a proxy server, operating on either Windows NT or Solaris. It is an intermediary server software that receives requests from a client (the device), makes requests for documents on behalf of that client, and returns the appropriate content to the client.

Through a series of conversion routines, the software automatically massages Web content into a format that matches the capabilities of the device. Let's look at how Prism functions from a high level.

On requesting a URL, a nonPC device connects to an Internet access provider's server where Prism resides. Once connected, the device identifies itself and the user to Spyglass Prism.

This information is cross-referenced against two different databases. The user database tracks user preferences, and the device database contains the characteristics of various devices, such as display resolution, color or monochrome support, and textual or graphical display.

Prism uses its own Web-browser component, which accesses the URL requested by the user. The browser uses this stored data about the Web site, user, and device to convert the data into the best format for the device.

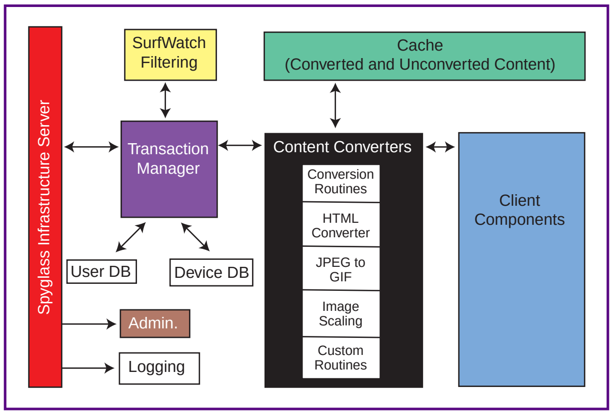


Figure 1—Spyglass Prism features a modular architecture, so many of the key components (e.g., the databases and caching module) can be replaced if necessary.

Images as well as HTML content are converted depending on the needs of each device. These conversions include changing color images to grayscale, reducing color depth, and converting JPEGs to GIFs.

HTML conversions can take many forms. Element start and end tags may be removed, leaving the content between the tags intact.

For example, if a device cannot display tables, the table-formatting tags are removed. But, the text in the table is still sent to the device in a simplified format.

Other removable elements include anchors, character formats (e.g., bold, italic), frames, tables, and tags such as `<isindex>`, `<link>`, `<object>`, `<bgsound>`, `<center>`, and `<META>`.

Attributes of an element may also be removed. The tag remains intact, but any attributes of the tag which are inappropriate for the target device are deleted. Hence, the background attribute of the `<BODY>` tag may be removed so no background images are downloaded which the device cannot display.

Similarly, if the `SRC`, `WIDTH`, `HEIGHT` attributes of the `<IMG>` tag are removed, the `ALT` text is displayed in place of the image, and the image cannot download.

Elements may also be replaced by elements more appropriate for the target device (e.g., marquee text is changed to bold). On a device that cannot display marquee text, you could draw attention to the text by resizing all `<H1>` to `<H3>`.

Attribute values (e.g., background colors) can be changed as well. Elements like comments, applets, scripts, styles, and images may be removed entirely.

If you're familiar with how Web browsers work, you'll have noticed that these conversions can be accomplished by a browser.

For instance, take a look at Microsoft's Pocket Internet Explorer (PIE). PIE is part of its Windows CE operating system and is found on all WinCE-based hand-held personal computers (HPCs).

Because these devices have a grayscale display, PIE must convert color images to grayscale. It also does some HTML conversions, eliminating applets and frames, but image conversion is its most important task.

The weakness in this solution, however, is that before PIE can do the color-to-grayscale conversion, it must first download a relatively large color image. To the user, this is wasted download time.

In contrast, Prism does this conversion on the server, only downloading a much smaller grayscale image. Given the fact that the dial-up modem connection is the slowest link in the access chain, this feature represents significant time savings.

Performance gain is also increased since the server on which Prism resides has considerably more processing power for image conversions than the HPC. The net result is that an HPC using Prism as a proxy server can access typical Web sites up to four times faster than it can via a direct connection to the Web.

PRISM ARCHITECTURE

Figure 1 diagrams the components that make up Prism. Let's look at a sample application to get a feel for how its functions work.

Spyglass's Infrastructure Server acts as the connection point for devices. In addition to its basic tasks of managing connections and user requests, the Infrastructure Server is also the point where security (e.g., SSL) and user identification are implemented.

Identifying the user, through a login screen or other methods, permits personalized content services to be delivered to the user. One of these services is SurfWatch content filtering, which lets users block access to content they deem inappropriate (e.g., sexually oriented, hate, gambling, drugs, or alcohol).

Identifying the user, and the type of device and browser, is the beginning of the content-conversion process. The Infrastructure Server creates a metadata structure that offers a construct for conveying information about the request between the components of Prism.

The format of the metadata structure is a series of comma-delimited name,value pairs that begins with the content of the original client request and any additional information it passes.

Header	Description	Example
spyga-element-neutralize	Remove element markup but leave content intact	spyga-element-neutralize blink
spyga-element-remove	Remove element markup and content	spyga-element-remove applet
spyga-element-replace	Replace start/end tags of element but retain attributes and element content	spyga-element-replace blink=strong
spyga-comment-remove	Remove comment markup and content	spyga-comment-remove
spyga-attribute-neutralize	Remove attribute name and value but retain element content	spyga-attribute-neutralize bodybackground
spyga-attribute-replace	Replace attribute value but leave attribute name and element content intact	spyga-attribute-replace hrwidth=100
spyga-attribute-scale	Scale attribute value by scaling factor	spyga-attribute-scale imgheight=075
spyga-attribute-min	Force attribute minimum value if original value exists and is not equal to 0	spyga-attribute-min hrsize=2
spyga-attribute-max	Force attribute maximum value	spyga-attribute-max hrwidth=100
spyga-image-scale	Scale image by specified amount	spyga-image-scale 50
spyga-image-encode	Translate file to specified encoding	spyga-element-encode gif

Table 1—This table lists the metadata headers. The metadata structure enables device and user-related information to be passed between the various components of Prism.

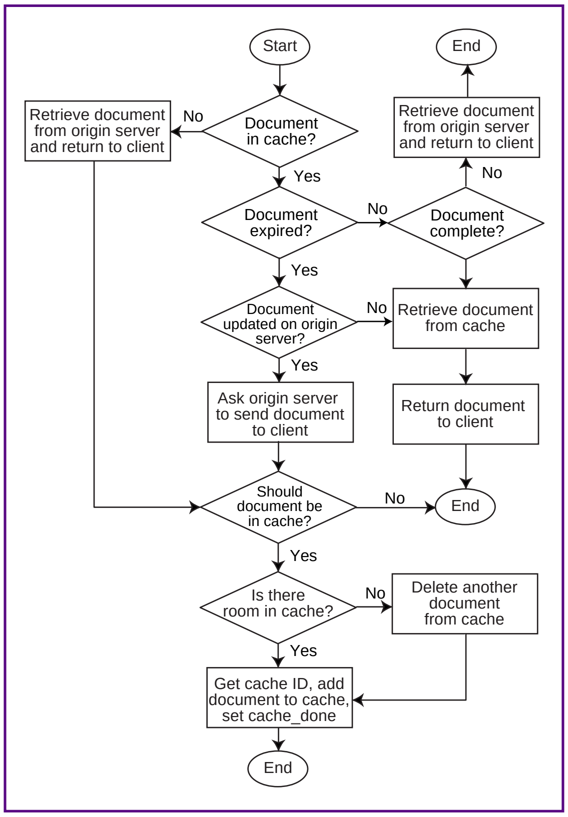

Start
Document in cache?
No
Retrieve document from origin server and return to client
Yes
Document expired?
No
Document complete?
No
Retrieve document from origin server and return to client
End
Yes
Document updated on origin server?
No
Retrieve document from cache
Return document to client
Yes
Ask origin server to send document to client
Should document be in cache?
No
End
Yes
Is there room in cache?
No
Delete another document from cache
Yes
Get cache ID, add document to cache, set cache_done
End

Figure 2—Prism's caching mechanism improves overall performance. Both converted and unconverted versions of content are stored in the cache, reducing the need to retrieve documents from the Web.

At this point, Prism has identified the user and, by means of the browser's user agent string, the type of device making the request. The request is now passed to the Transaction Manager.

The Transaction Manager enhances incoming requests with information about the user and device. Using the user-agent information passed on by the Infrastructure Server as a key, the Transaction Manager queries the device and user databases for additional information.

The databases are object oriented, using Versant technology. The user database maintains conversion and filtering preferences for each user. The device database offers information about each type of device such as display resolution and supported image formats (GIF, JPEG) and HTML tags.

These databases determine how to convert the Web content before passing it back to the client. For example, they can contain the following information for a particular device and user:

- convert color images to four-bit grayscale
- eliminate images larger than 50 KB
- convert tables to text
- no SurfWatch filtering

The Transaction Manager appends these preferences to the metadata before passing the request on to the Content Converter.

Tables 1 list Prism's metadata headers. Only the first conversion in the collection is performed per element or `element.attribute` pair.

Hence, `spyga-attribute-scale img.height=0.75` and `spyga-attribute-min img.height=10` cannot work together. Only the one that appears first is performed.

In addition, UA-color headers provided by the device indicate what type of color-depth conversion is performed. These headers are included in metadata.

The Content Converter module tailors Web content to meet the characteristics of a specific user and device. It is invoked by the Transaction Manager, which passes it the metadata of user and device preferences.

Using a multivalue key derived from the metadata, the Content Converter queries the cache to determine if a previously converted version of the content is available. There are three possible results of this query.

If there is a converted version in the cache, the Content Converter returns it to the Transaction Manager. If the cache only contains the original content with no converted version, the Content Converter converts it to suit the needs of the user and device.

If the original version of the content is not in cache, the Content Converter invokes a Client module to retrieve the requested content and then converts it according to the needs of the requesting user and device. The original and converted versions are then added to the cache.

When content requiring conversion is returned from cache or the Web, the Content Converter first looks at the MIME type(s) of the content and loads the appropriate conversion routine(s).

For example, for an HTML document, the HTML Parse routine is loaded. For a JPEG image, JPEG-to-GIF and color-reduction routines might be loaded.

In addition to the MIME type, other information derived from the metadata determines which conversion routines execute as well as which parameters are used by a routine. Such information includes the requested URL from the HTTP request headers, user preferences from the user database, and device attributes and preferences from the device database.

The HTML Parse routine creates a parse tree and delivers parsed data back to the Content Converter. After comparing the parsed data with the metadata, additional conversion routines may be executed. Routines may be invoked to remove, replace, or modify HTML tags and attributes.

The output from the conversion routines is assembled into the new converted document based on the parse tree. The Content Converter then adds the converted content to the cache and provides it to Transaction Manager for subsequent delivery to the client device.

The architecture of the Content Converter requires all content conversion routines to be provided as shared libraries (DLLs). Therefore, new and updated conversion routines can be added to Prism at any time.

By default, routines are loaded on demand. Frequently used routines may be marked "resident" so that they are loaded at startup and never unloaded.

Routines can be written that remove part of the content or that transform elements in the content. They can also be written to customize content based on specialized knowledge of particular content (e.g., converting a table to a specific format based on the contents of that table).

CACHE

Caching plays a significant role in Prism's performance. To understand how beneficial caching can be, it's important to understand the overall solution.

Prism is an intermediary between the Web and some nonPC device. The nature of Web access on these devices is quite different than it is on PCs.

Because of the wide variety of tasks performed on a PC and its relatively high performance, a typical user accesses many Web sites. This variety reduces the likelihood that the document they need is cached.

PDA users behave differently. They perform more specialized tasks and access a much smaller number of Web sites. Therefore, they have an increased chance of benefiting from caching.

However, it's not always appropriate to cache content. There are some cases where Prism must always retrieve content from the origin server.

Obviously, content must be received from the origin server if this is the first time the content has been requested or if the content was removed from the cache.

The Caching module assigns an expiration time to all content when it is cached. If the origin server doesn't provide an expiration period, the Caching module calculates one based on a configurable value.

The content is considered fresh until the end of the expiration period, at which time the Caching module marks it as stale.

When a client requests content which is in the cache but marked stale, Prism asks the origin server if the content has been updated. If it has, the server transfers the updated content.

If the content has not been updated, the origin server returns a status code of 304, and the Caching module marks the content as fresh again. Prism never checks if fresh content has been updated.

Prism will also retrieve content from the origin server if there is a cookie-related request or response header. This content might be customized for the client based on the cookie value.

Cookies are commonly used to record a user's activity on a Web site or to recall their preferences for the type of data they see on a site. For example, many news-related sites let you personalize the topics of news you see by having you fill out a form with your preferences.

The same is true when there is an authorization-related request header. The content is only accessible by those with authorization.

If the response is intended as a server-push, then the content might be excessively long and involve intentional pauses in transmission. So, caching is not appropriate here.

As well, responses that are missing the Content-Length and Last-Modified response header fields are likely to be generated on-the-fly and should not be cached.

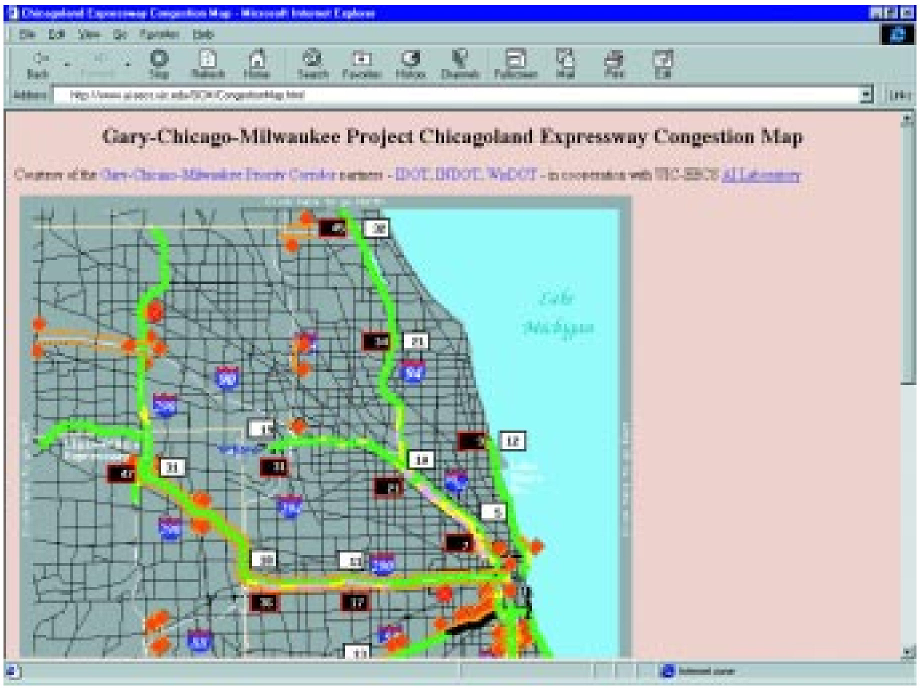

Chicagoland Expressway Congestion Map - Microsoft Internet Explorer
Gary-Chicago-Milwaukee Project Chicagoland Expressway Congestion Map
Lake Michigan

Photo 1—Here is a typical Web page as viewed on a personal computer with Microsoft's Internet Explorer. Spyglass Prism hasn't converted this page.

If the response is fairly new, then it is assumed that this content is more likely to be updated in a short period.

A cookie, which contains your preferences, is then created by the server, sent to your browser, and stored on your computer. Whenever you visit the site, the cookie is retrieved by the server so your personalized information can be displayed.

When content is not in the cache, the Content Converter invokes the Client Components module to retrieve it from the Web. The caching function is summarized in Figure 2.

CLIENT COMPONENTS

Prism includes client components used to retrieve content from the Web. The components include support for images (GIF, JPEG), HTML, HTTP, FTP, and other Web client technologies.

Content retrieved by the client components is returned to the Content Converter, starting the data flow back to the user.

ROADMAPS EN ROUTE

This example illustrates the steps that occur when Prism converts a Web page designed for a PC to information suitable for viewing on an HPC with a simple display. Photo 1 shows how the page appears when viewed on a PC with Internet Explorer.

In this page, the GIF image provides a full-color representation of real-time roadway conditions, and text links provide HTML text descriptions of current roadway conditions.

A mailto link enables users to send E-mail to the Webmaster. A GIF image acts as a banner ad, providing name exposure for page sponsors.

The target HPC has 256 KB of memory and a 480 × 240 monochrome display running PIE. Photo 2 depicts how the page appears on an HPC after being converted by Prism.

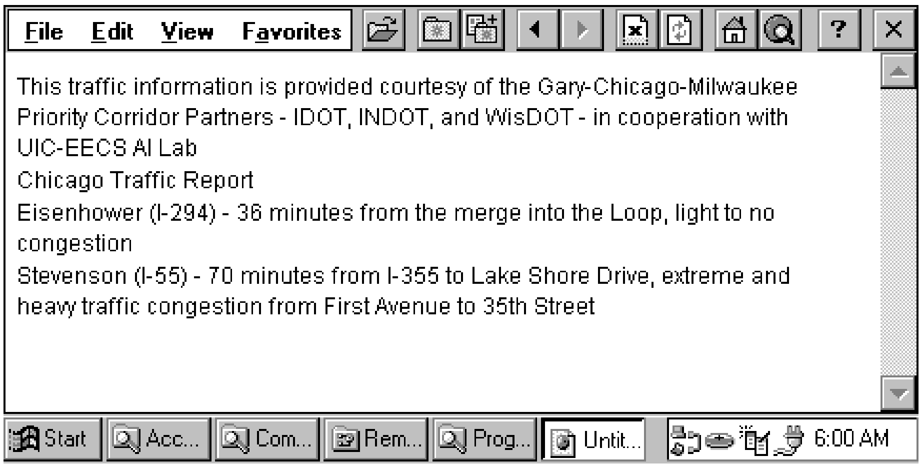
Without Prism, most of the content on this Web page cannot be displayed on an HPC. To display this information, Prism sends the traffic information as scrollable text, provides alternate text for the banner ad, and removes the images and background.

The text alternatives of the graphical traffic-congestion data and advertiser information are not changed.

UNIVERSAL CONVERSION

Content conversion is a relatively new concept. As Web content becomes more complex, conversion will become more difficult, even though the need for it will be much greater.

Our desire to access key information, regardless of where we are, shows no signs of declining. And, the Web is in-



This traffic information is provided courtesy of the Gary-Chicago-Milwaukee Priority Corridor Partners - IDOT, INDOT, and WisDOT - in cooperation with UIC-EECS AI Lab
Chicago Traffic Report
Eisenhower (I-294) - 36 minutes from the merge into the Loop, light to no congestion
Stevenson (I-55) - 70 minutes from I-355 to Lake Shore Drive, extreme and heavy traffic congestion from First Avenue to 35th Street

Photo 2—Here's the same Web page as viewed on a hand-held personal computer (HPC) after Spyglass Prism content conversion.

creasingly the repository for this data in both the public Internet and corporate Intranets.

Effective ways to bridge the variety of form factors found in information access devices must be created. The ability to take existing standard HTML content and deliver it to multiple devices represents a win for everyone involved—from the end user to the device manufacturer and content provider.

This type of solution accelerates the expansion of the Web beyond just the PC, bringing universal access to what has become the universal medium. EPC

As product manager, Dan Johnson leads the team responsible for developing the marketing programs for Spyglass's line of Internet software products and services for wireless and telecommunications devices including smart phones, PDAs, and laptops. His main challenge is to define, direct, and measure product marketing strategy for the wireless and telecommunications market. You may reach Dan at djohnson@spyglass.com.

SOURCES

Spyglass Prism
Spyglass, Inc.
1240 E. Diehl Rd., 4th Fl.
Naperville, IL 60563
(630) 505-1010
Fax: (630) 505-4944
needs@spyglass.com
www.spyglass.com

PIE, Windows CE
Microsoft Corp.
10500 NE 8th St., Ste. 1300
Bellevue, WA 98004
(206) 635-1900
Fax: (206) 635-1049

Object-oriented databases
Versant Object Technology
6539 Dumbarton Circle
Fremont, CA 94555
(510) 789-1500
Fax: (510) 789-1515
info@versant.com
www.versant.com

IRS

410 Very Useful
411 Moderately Useful
412 Not Useful

Marc Guillemont

Real-Time Operating Systems

Part 2: RTOS Interfacing

After defining what a real-time system is in Part 1, Marc puts the RTOS in its place—right between the device driver and application software, where it can implement hardware abstractions and allocate resources more efficiently.

In Part 1, I introduced real-time operating systems and defined some of their components as well as the differences between hard and soft real-time applications. Now, I want to cover how the RTOS interfaces to hardware with device drivers and applications through the application programming interface (API). It's where the rubber meets the road.

This duo paves the way for *INK*'s new "Real-Time PC" column. Articles on choosing an RTOS, using it in various applications, networking, and so on are in the works.

START YOUR ENGINES

I think of the RTOS as sitting between the device-driver and application software. Being in the middle enables it to act like a traffic cop, directing each component to the resources it needs.

An RTOS tries to hide the heterogeneity of different hardware architectures to simplify development and enhance portability of applications across various processors and boards. In a generic OS, which is concerned with allocating resources fairly, the abstraction level is high and far removed from the hardware.

Unix, for example, only deals with two different kinds of devices—block oriented, like disks, and character based, which are unlike disks. Block-oriented devices are only used by the file-system driver. Character-based devices permit a consistent interface to hardware and files via generic file I/O primitives like open, read, write, and close.

RTOSs, however, don't care about dealing with resources fairly. In fact, you want high-priority tasks to get the resources they need right away. This way, they can meet their deadlines at the cost of preempting tasks that don't have as high a priority.

So, in real-time embedded systems, the hardware abstraction is not cast in stone. In some cases, there may not be any kind of isolation and an application may even touch a device register itself.

The abstraction required usually depends on the device and applications. An RTOS gives you the tools to implement different abstractions through its API. Let's look at how this might work.

I first discuss some typical hardware devices that might be found in an embedded system in terms of their function as seen by the programmer. I then show how a device driver might be constructed and how the device driver and application use the RTOS's API to abstract the hardware.

To understand how the RTOS interfaces to the hardware, you need to be familiar with the hardware devices you're likely to encounter. So, let's discuss how some common devices work and what kind of interface they present to the software.

SERIAL INTERFACES

Serial ports are probably the most common hardware device. The serial interface is also known as the Universal Asynchronous Receiver and Transmitter (UART), and UARTs that implement synchronous serial interfaces are referred to as Universal Synchronous and Asynchronous Receiver and Transmitters (USARTs).

The UART's primary function is to convert parallel data into serial bitstreams and then convert received bitstreams back to parallel. It accomplishes these tasks via shift registers.

Parallel data is loaded in a shift register and shifted out over a single interface. The receiver takes this serial bitstream and deserializes it using a serial-to-parallel shift register. Once the word is received, it can be read by the receiver.

This technique would work fine—if the transmitter and receiver were perfectly synchronized and knew implicitly where the word boundaries in the serial bitstream were. In real life, especially if the devices are separated by some distance and are not synchronized, this is close to impossible. Here's where the asynchronous serial protocol comes to the rescue.

With the asynchronous serial protocol, each word is introduced by a start bit. The start bit is a mark condition, which is opposite of the space condition (i.e., the serial line is between words of data). The word is followed by one or more stop bits.

A stop bit is just a bit period at which the line is held at a space condition to ensure you can detect the start mark of the next word. This scheme enables devices to transmit data at any rate, even with idle spaces between words.

The bit clock rate transmitting serial data is usually much slower than the device producing data. To prevent the transmitting device from overrunning the serial transmitter with data, a handshaking method is used.

The transmit buffer empty (TBE) flag is set whenever the transmit shift register has transmitted the start bit, data word, and stop bit(s). Loading a value into the transmit register clears this flag.

The receiver, on the other hand, listens to the serial bitstream, waiting for a transition from space to a start-bit mark. When it sees this transition, it shifts the next required number of bits into a register. Once the data is shifted in, it verifies this was a valid transmission by making sure the line returns to an idle state, indicating the stop condition.

If the word is received successfully, the receiver logic asserts the receiver buffer full (RBF) flag, indicating that data is ready to be read. The device reading the data clears this flag by reading the received data register.

Listing 1—The serial-port interrupt service routine is responsible for pulling characters from the serial-port interface and queuing them for the upper level driver.

```
ser_int(){
  if (ser_status() & SER_RBF){
    MutexWait(ser.mutex);
    ser.inbuf[ser.inhead] = ser_data();
    ser.inhead %= SER_BUFSIZE;
    MutexRelease(ser.mutex);
    Signal(ser.event);} }
```

You may also add flags in the receiver indicating error conditions, such as not detecting a stop condition (framing error) or when the receiving device doesn't clear the data from the received data register before new data comes in (overrun error). You can also add a parity bit to the serial data transmitted, either even or odd, which would cause a parity error condition in the receiver if the parity didn't match.

Commercial UART interfaces, like the 8250/16450 and 16550, add features like a programmable bit-rate register, various buffers, and FIFO. Also, a modem interface lets the UART module control modem functions and sense modem status information (e.g., carrier detect).

Serial port interfaces of interest to RTOS developers can also interrupt the CPU when characters are received, the transmitter buffer becomes empty, or the modem status and error conditions change.

PARALLEL PORTS

Next to serial interfaces, parallel interfaces are very common. They're flexible, and they enable the system to communicate with a variety of external devices, send individual digital control signals, and sense external signals.

The simplest implementation of a parallel I/O interface is the use of a register on the system's bus to output signals and a tristate buffer to read signals onto the processor bus when addressed. These ports are useful when you interface simple devices like lights, relays, or switches.

Parallel-port controller chips are flexible devices, containing a cluster of parallel ports that can be configured as output and input ports. Parallel-port controllers also implement handshaking hardware logic that can operate some I/O signals as handshake signals to simplify communication with other parallel peripherals.

Implementing these handshake signals directly in the parallel-port controller chip offloads the processor from individually monitoring control signals. The controller can usually be configured to interrupt the CPU when a parallel word has arrived or when it's ready to transmit the next word.

COUNTERS AND TIMERS

Timers and counters measure time intervals and count events. A counter/timer chip (CTC) contains several counters that are clocked from an external clock signal and controlled with external inputs (gates). Each CTC also has an output signal, which can be wired to an interrupt or another device.

Each CTC counter unit is programmable to operate in various modes and may include a programmable prescaler. The prescaler reduces the external clock rate to slow down the counting rates.

Modes such as pulse width measurement will, once the counter is armed, measure the width of a pulse presented on the gate input. Other modes may provide a fixed delayed output, depending on the input-gate start signal.

CTC counters can also generate system interrupts. The programmable-delay and free-running modes are of interest here.

In the programmable-delay mode, the counter interrupts when it reaches zero. The program desiring such service loads the timer with the desired delay count and goes about its business. This facility is useful for scheduling events in an RTOS.

You can use the delay mode for watchdog or failsafe timers by wiring the output to a nonmaskable interrupt or the system's reset signal. As a watchdog timer, the software must reload the timer before it times out and resets the system. This helps detect when a system hangs.

If the counter is set in a free-running mode, it counts to zero and then pulses or toggles the output line. It also reloads itself from a counter-holding register and restarts the cycle. The steady rate of interrupts produced by this mode is useful for heartbeat interrupts in the system and software-based real-time clocks.

CTCs also let you start and stop counters, as well as read and load counter values while a counter is running.

INTERRUPT CONTROLLERS

Real-time systems rely on interrupts to offload the CPU from polling hardware devices. Interrupt controllers manage multiple interrupt sources from various devices and prioritize them.

For example, a timer interrupt may need a very low interrupt latency to properly time the generation of an event. By giving this timer a higher priority and allowing it to interrupt other interrupt service routines, you can ensure that the timer interrupts at the lowest latency.

An interrupt controller prioritizes interrupts via a priority encoder. A typical number of interrupt sources for an interrupt controller is eight.

Interrupt sources may be nested. On the PC/AT architecture, for example, the interrupt level IRQ2 maps interrupts IRQ9–15. In this case, IRQ2's interrupt source is another interrupt controller with eight more interrupt sources.

The interrupt controller also has to generate an interrupt vector for the processor when the CPU acknowledges the interrupt. This vector says which interrupt occurred and which service routine to call.

To manage the interrupts, the interrupt controller uses a set of registers. The interrupt mask register masks off (i.e., disables) a particular interrupt source. The interrupt pending register indicates which interrupt is currently pending. And, the interrupt service register indicates the interrupt currently being serviced by the CPU.

When an interrupt source becomes active and is not masked in the interrupt mask register, it sets a corresponding bit in the interrupt pending register. If this interrupt is currently the highest priority interrupt, an interrupt is generated on the CPU.

The CPU acknowledges the interrupt to which the interrupt controller supplies the interrupt vector for the interrupt source and sets the corresponding bit in the interrupt service register. The CPU then executes the appropriate interrupt service routine.

Another interrupt may only initiate an interrupt cycle on the CPU if its priority is higher than those already in service, as indicated by the interrupt service register. When the CPU executes an end-of-interrupt instruction, the bit for the interrupt is cleared in the interrupt service register and the system is ready to respond to this particular interrupt again.

Listing 2—The upper level driver module implements the line-oriented abstraction of this serial port to the application. This module processes raw characters received from the serial interface and packages them into a line buffer for the application.

```
ser_edit(){
  char data;
  while(1){
    Wait(ser.event);
    MutexWait(ser.mutex);
    data = ser.buf[ser.tail++];
    ser.tail %= SER_BUFSIZE;
    MutexRelease(ser.mutex);
    switch(data){
      case CNTL_H:
        MutexWait(cmd.mutex);
        if(cmd.len != 0)
          cmd.len–;
        MutexRelease(cmd.mutex);
        break;
      case CNTL_X:
        MutexWait(cmd.mutex);
        while(cmd.len != 0)
          cmd.len–;
        MutexRelease(cmd.mutex);
        break;
      case CNTL_M:
        Signal(cmd.event);
        break;
      default:
        MutexWait(cmd.mutex);
        cmd.buf[cmd.len++] = data;
        MutexRelease(cmd.mutex);
        break;} } }
```

DMA CONTROLLERS

Direct memory access controllers (DMACs) move data between I/O devices and memory or from one memory region to another without CPU intervention. Like interrupt controllers, these data-movement engines prioritize requests from several sources. Priorities are normally fixed, but some controllers can implement priority schemes like round robin.

A DMAC maintains several bits of information about each DMA channel it intends to service. It needs to know where to start the transfer as well as the length and types of DMA transfer to do.

After a DMA channel is initialized, the DMAC listens for a DMA request from a device. The device indicates its readiness to transfer data by asserting the DMA request signal. The DMAC then tries to take control of the system bus by requesting that the CPU stop and get off the bus.

Once the CPU says it has relinquished the bus, the DMAC asserts the target address and signals the requesting device to transfer the data. Once the data is transferred, the DMAC decrements the length count and asserts a signal (EOP) when it reaches zero. The device signals that it's done transferring data by deasserting the request signal.

When using a separate DMA, as in a PC/AT-architecture–based system, the device-driver routine needs to initialize the start and count register for each transfer.

However, master-capable devices on multimaster buses (e.g., PCI or VME bus) that may include the DMAC in the peripheral make memory transfers between the device and system memory transparent. Some devices, such as network and disk controllers, may even be intelligent enough to implement their own buffer management in system memory by using DMA.

DATA-ACQUISITION DEVICES

One common data-acquisition device is the ADC. The conversion process from analog signal to digital word may take some time. To begin, an ADC requires a start signal. It then interrupts the processor when the conversion is complete.

The start signal can come from a timer or may be generated by the CPU when it writes to a register. For constant rate acquisitions, such as used in DSP applications, the start signal is generated by a hardware timer. Very high-speed ADCs may also use DMA to transfer data into buffer memory, which is then read and processed by the CPU.

DEVICE DRIVERS

Device drivers are the glue that interfaces hardware to the RTOS and, in some cases, the application. They abstract the device for the application.

In real-time systems, the device should be abstracted in such a way that the amount of time spent in the interrupt service routine is minimal. The more time spent in a interrupt service routine, the longer that lower priority interrupts and task execution are blocked.

When designing device drivers, we usually try to divide the driver into two layers—the lower and upper levels. The lower level is the software that directly touches the hardware registers and handles interrupts. The upper level does the rest.

In a thread-based RTOS, the lower level is implemented as a interrupt service routine and the upper level is usually a thread.

Consider a hypothetical serial-port device driver. This serial-port device is ideal and deals only with the receive buffer full and transmit buffer empty conditions. (Real serial device drivers may also have to deal with error conditions, modem control signals, and transmit buffer empty interrupts.)

Let's look at the low-level part first. The device driver has one interrupt service routine (`ser_int`) which, once invoked by the CPU's interrupt dispatch system, needs to figure out why we were interrupted and what to do about it. Listing 1 shows what this might look like.

Once it is determined to be a receive buffer full event, the routine makes sure that it has exclusive access to the circular buffer using a mutex. It then extracts the data received and inserts it into the buffer. Once the data is stored, the driver sends a signal to whomever is waiting, indicating that data has been added to the buffer.

In this example, I implement the upper level device-driver routine as a single task that implements a simple line editor. A user can enter data into line buffer, and once a carriage return is received, the line buffer is made available to other tasks.

Ctrl-H erases the last character in the command line buffer, and Ctrl-X erases the whole line. Listing 2 shows the editor task.

This device driver abstracts the serial input port as a line-oriented device. This situation is useful if the serial port is connected to a keyboard and the application implements a command line interpreter. On the other hand, if the serial port is connected to a communications device, the device driver may need to implement a different kind of device abstraction.

All that is left now is the initialization routine, which is shown in Listing 3. During the initialization, I set up the queue used by the interrupt service routine to communicate with the upper level and the mutex

Listing 3—The initialization routine is responsible for setting up data structures and OS resources that are needed to implement the driver.

```
struct ser{
  Mutex mutex;
  Event event;
  int   tail;
  int   head;
  char  buf[SER_BUFSIZE];}
ser;
struct cmd{
  Mutex cmdmutex;
  Event cmdevent
  char  cmdlen;
  char  cmdbuf[SER_BUFSIZE];}
ser_init(){
  ser.mutex = MutexCreate();
  ser.event = EventCreate();
  ser.tail  = ser.head = 0;
  cmd.mutex = MutexCreate();
  cmd.event = EventCreate();
  cmd.len   = 0;
  RegisterISR(ser_int,IRQ_SERIAL);
  ser_hardware_init(baudrate,stopsbits,wordsize);}
```

needed to protect the data structure.

API

In my example, I access several RTOS functions through the RTOS's API, which is usually presented as a set of function calls. The functions calls are accessed by either linking in a set of libraries or including `include` files.

The RTOS API is normally divided into several groups of functionality. Examples of these groups are task management, memory management, interprocess communication, and networking. In large RTOSs, these API groups are sometimes bundled in separate library modules.

In each functional group, an API usually falls into three categories of routines. One category initializes the functionality or resource. In the serial-driver example, these were calls to the routine `MutexCreate()`.

Other routines use or operate with this resource or functionality (e.g., `MutexWait()` and `MutexRelease()`). And others free up the resources we may have allocated. For example, `MutexFree()` might free up a mutex resource.

By using the API, you can make sure that the interface between the application and the RTOS is defined and everyone understands what to expect. Also, an RTOS implementer may change or enhance some functionality, without breaking existing code.

You could even imagine a new mutex type, which may have a new API, like `NewMutex{Create,Wait,Release}`. The application-code developer can then decide whether to use the new mutex type by accessing the routines in the new API.

WHAT NOW?

In this series, I've introduced RTOSs, described why they are useful, and showed you how to use some of their key functions.

Of course, it's impossible to define all RTOS terms and structure in just two articles. However, future Real-Time PC articles will develop RTOS-related topics and idiosyncracies and how to implement them. Be sure to check in for details. RPC.EPC

Marc Guillemont, ChorusOS product manager for Sun Microsystems' Embedded Systems Group, joined INRIA in 1977 to work on the Cyclades project. He was a member of the initial Chorus research project team in 1980 before becoming head of the team. Marc managed the final research phases of ChorusOS before developing the commercial version. You may reach him at marc.guillemont@france.sun.com.

REFERENCES

M.J. Bach, *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, NJ, 1986.
Chorus Systems, *CHORUS/OS User Manual*, 1997.
D. Comer, *Operating System Design: The XINU Approach*, Prentice-Hall, Englewood Cliffs, NJ, 1984.
L.C. Eggbrecht, *Interfacing to the IBM Personal Computer*, SAMS, Carmel, IN, 1990.
H.-P. Messmer, *The Indispensable PC Hardware Book*, Addison-Wesley, Reading, MA, 1997.
Phar Lap Software, *ETS Technical Reference*, TNT Embedded ToolSuite, 1996.
QNX, *QNX Operating Systems: System Architecture*, 1996.
T. Shanley, *PCI System Architecture*, Addison-Wesley, Reading, MA, 1996.
E. Solari, *ISA & EISA: Theory and Operation*, Annabooks, San Diego, CA, 1992.
A.S. Tanenbaum, *Operating Systems: Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ, 1987.

IRS

413 Very Useful
414 Moderately Useful
415 Not Useful

Fred Eady

RF Telemetry

Part 2: You're on the Air

Fred wants to implement duplex RF communications between two sites using a BiM RF module. Connected to a VIPer 806 and a PCM-4862, these modules transmit and receive RF data he can view on a software-based datascope.

I probably scared the stuffing out of you in my first RF-oriented installment. To us that work in the world of magnetic-wave generation, the mere mention of the FCC usually means trouble one way or another. But, in actuality, the FCC is our friend.

Without their rules and regulations, there would be untold magnetic-wave bashing going on out there. Even in our embedded environment, the FCC is more help than hindrance, as RF-enhanced embedded applications are becoming more and more popular.

Last time, I promised a closer look at putting some data in the ether between an embedded system and some kind of target. Thanks to a company called Linx, in Part 1, I was able to introduce you to a basic transmitter/receiver pair that with a little tweaking could be used to further data along its course in the magnetic ocean we live in.

In my travels since then, I've become acquainted with another company across the Canadian border that offers an abundance of RF goodies well-suited to embedded RF applications. So, let's take a look at some of the magnetic wave-bending modules offered by Abacom Technologies.

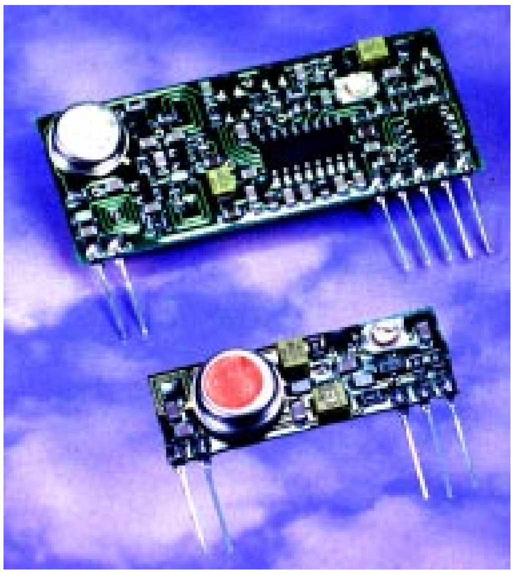


Photo 1—These little jewels are marvels of today's miniature technology. The transmitter is the smaller of the two units.

RF MODULE HEAVEN

Just like the Linx products described in Part 1, Abacom offers the TXM-*xxx* transmitter module. The RXM receivers offered by Linx are paralleled by Abacom's SILRX series.

The modules are pretty much identical in form and operation. All you need to get on the air is clean power, a suitable antenna, and some data to send. Photo 1 shows us this entry-level pair of RF modules.

These versatile little boards work at 418, 433.92, and 403 MHz. Interestingly enough, the 403-MHz variant is exclusive to the South African RF bit-bangers.

Abacom also has a more secure RXM receiver (RXM-*xxx*-A), which incorporates features like received signal strength indication and antenna tamper sensing. The "*xxx*" is the module operating frequency.

Up to now, every RF module I've described has been single-functioned. Receiver modules receive, and transmitter modules transmit. Imagine that.

With what we know now, to implement duplex RF communications between two

Photo 2—This is getting better and better. Everything for the RF-challenged designer is included in a single neat little package.

sites, we'd need a minimum of four RF modules. It is indeed possible to put this type of communications system to work, but the physics of double antennae at each site and the design of elaborate transmit/receive switching circuitry would have to be dealt with.

A simple pair of kid's walkie-talkies would be a more elegantly engineered RF lashup than the suite of modules I just described. Obviously, the solution is to find a suitable RF module out there that would perhaps incorporate some of these desirable functions.

BiM-4*xx*-F/HP TRANSCEIVER

The BiM RF module family is a series of miniature UHF radio modules capable of half-duplex data transmission at speed of up to 40 kbps (see Photo 2). These low-power RF modules can communicate at distances up to 120 m over open terrain. For indoor applications, the effective range is 30 m.

Like the simpler TXM/RXM, the BiM uses SAW (Surface Acoustic Wave) controlled FM transmission at frequencies of 418 and 433.92 MHz. The "*xx*" in "4*xx*" designates the frequency of operation.

Radiated RF energy from the F series is set at –6 dBm. A higher powered version (HP) radiates at 0 dBm. With a receive sensitivity of –107 dBm, the BiM's receiver section can pull in signals way down in the dust.

A single antenna services both transmitter and receiver circuitry, as on-board antenna and power supply switches are integral to all BiM modules. A single 4.5–5.5-V at 25-mA power source is all that's required to fire up a BiM. If you use all of the BiM's features including the loop test, power-supply consumption varies from 1 µA to around 20 mA. Under normal operation, 15 mA is the average current consumption.

CMOS-level logic interfacing and fast receiver power-up capability make the BiM transceiver modules ideal for battery-powered applications and easy to interface to most microcontrollers and processors. Typically, the BiM's receiver section can come to life in about 1 ms. A block diagram of a typical BiM is shown in Figure 1.

JUST WHEN YOU THOUGHT...

You were settling in for some C code and application physics, Fred throws a changeup. Sorry. No C here.

BiM is designed to be versatile. Its standard CMOS data and control pins scream for a CMOS microcontroller's I/O interface.

Sorry. No stand-alone micros in this article, either. C and an appropriate embedded PC coupled with some front-end CMOS logic or processor would be typical here, but you're reading *INK*. That means you are most likely not the typical cookie-cutter engineer. So, let's do something entirely different.

Ever hear of Visual Basic? Did you know that VB V.5 compiles? Did you know Visual Basic 5 is object oriented and event driven? Do you know anybody who uses a computer and hasn't written a program in BASIC?

Whether you answered yes or no to any of these questions, it doesn't matter. We're going to use the BiM modules, Visual Basic 5, Windows 95, a couple of very capable embedded PCs, and some trick bench equipment to modulate some ether. But before we get any deeper, there are some more details we need to cover concerning the BiM.

BiM BASICS

Although the BiM is nifty as far as RF stuff goes, it is simply a transmitter and receiver that can be controlled by another device of higher intelligence that needs to transport data from point A to point B without the aid of hard wires.

The BiM has no control of how the data it sends and receives is formatted. It is up to the intelligent subsystem to code and decode the data placed into the ether by the BiM. Since this article has been declared a "no micro, no C" zone, intelligence in our world is defined as an embedded-PC hardware/software suite.

Wait a minute. Just because I said no micros doesn't mean we can't use a micro example to push a point forward. I love using micros. I'm just not going to in this article.

Figure 2 shows us how the typical micro would interface with a BiM. Basically, there's serial I/O (SDO and SDI) working in harmony with bit-based I/O (I/O 0–2).

You know where I'm going. What if we replace the micro in Figure 2 with the serial and parallel interfaces of an embedded PC? Theoretically, the physical layout in Figure 2 would not change and as long as we didn't

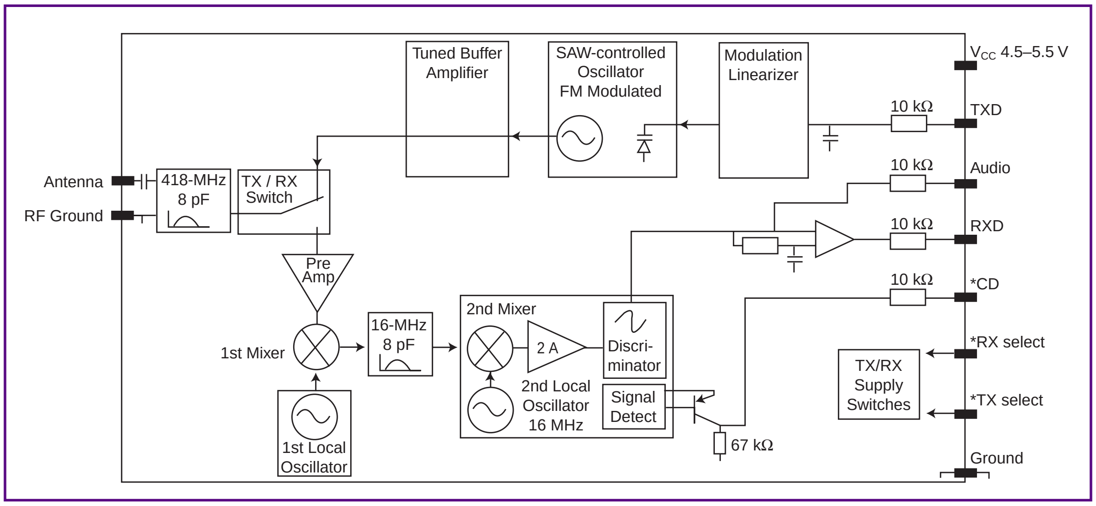


Figure 1—The CMOS interface and simplistic layout of the BiM can take the pain out of many embedded RF applications.

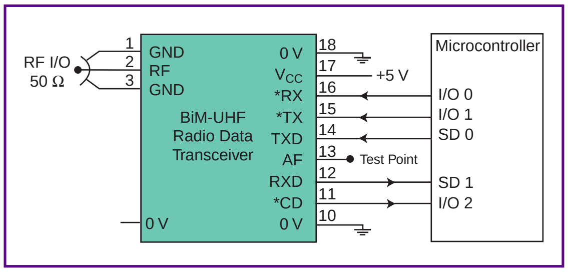


Figure 2—As you can see, this is a perfect union of RF and micro. It's also the basis for a perfect union of RF and embedded PC.

violate the BiM's 5-V interface rule, the whole thing would probably fly. Right!

How many times have I taken that shortcut? How many times have you taken that shortcut? How many times did you end up buffering things you knew you should have buffered in the first place?

If you want to connect directly out of the standard embedded ports, be my guest. If not, Figure 3 ensures that CMOS levels will be present at the BiM's interface.

Now that the data and control interfaces between our embedded PC and the BiM have been defined, let's look at what needs to be considered once the data has entered the BiM's electronics. First of all, the data path within the BiM is AC coupled.

As you well know, where there are capacitors or capacitance, timing is everything. The minimum time allowed between consecutive transitions is 25 µs, while 2 ms is the maximum. By keeping our data rates between 4800 and 38,400 bps, even the worst case of all zeros or all ones in an eight-bit frame won't violate the timing window.

To provide increased immunity to RF interference, the BiM's AFC and data-slicer electronics require a preamble of at least 3 ms of hex 55 or hex AA characters. These are transmitted before data at the RXD output may be considered reliable. The datasheet recommends 5 ms of preamble for increased data integrity. This preamble is also specified as the receive settling time.

To reduce pulse width distortion and increase noise tolerance, it is recommended that the mark/space ratio be kept as close to 50:50 as possible. The data slicer is optimized for 50:50 but can handle ratios of 30:70 or vice versa with some degradation of the data quality. The bit error rate is based on the mark/space ratio over a 4-ms period.

If you've assumed that we will be providing the BiM serial data from a UART-based embedded serial port, you're on the beam. With that assumption and our working knowledge of typical async data packets, we have quickly come to the conclusion that we can't guarantee a 50:50 mix of transitions within any particular data framed from the UART.

The good news is that the BiM can handle raw RS-232 serial data as long as some rules are followed.

First of all, the data rate must fall between 4800 and 38,400 bps. Our application can live with this. If you need a higher speed for your app, just hold on. I'll get to that.

Of course, the transmission must be in half duplex mode. No problem there, either.

Secondly, the data must be packetized and contain no gaps between the bytes. Here's how that's done. The preamble is sent for a minimum of 5 ms, which allows the data slicer to settle.

This is immediately followed by a couple of bytes of hex FF. The hex FFs lock the UART by placing the datastream in a marking state. The UART expects to see a start bit following the hex FF bytes. A unique start of message byte is then sent followed by the actual data.

If you want extreme accuracy, a checksum or CRC byte is sent at the end of the transmitted packet. Since this is a duplex system, the receiver can verify the CRC or checksum and request a resend of the last packet if things don't match.

Now, for you guys and gals who want to operate at hyperspeed, you need to adhere to the 50:50 mark/space ratio rule. If you plan to operate above 20 kbps, this is mandatory. Three ways are recommended to get the 50:50 mark/space ratio and thus transmission speeds of up to 40 kbps between BiMs.

The first method is called Biphase or Manchester coding. Each bit is sent as two bits. The first bit is the actual data bit to be sent and the second bit is the first bit's complement. This guarantees a transition in the "center" of the bit which is a perfect 50:50 mark/space ratio. The 100% redundancy of the data offers 40 kbps throughput, but it actually equates to only 20 kbps of usable throughput.

Another scheme called byte-coding assumes that only a subset of the ASCII code is required as data. A look-up table is needed as only 70 of the possible 256 eight-bit codes are used. All of the 70 codes contain four ones and four zeros. In actuality, only 68 of the 70 are used because hex 0F and hex F0 are left out to minimize consecutive zeros and ones.

Using a standard UART setup for one stop bit, one start bit, and no parity, these codes meet the 50:50 mark/space requirement. An additional plus to this method is that error checking is simplified because of the 4:4 ratio of ones to zeros in each byte.

FEC coding is the third way to balance the mark/space ratio. In this coding scheme, each byte is sent twice. The first send is true data, with the second send being a complement of the first.

This scheme can be enhanced by adding a parity bit. With parity, the decoder algorithm can check the true byte for correct parity and on parity failure select the next inverted byte to use for data if it's not corrupt.

The bottom line is that packets sent between BiMs must contain a preamble, a control bit or bytes, data, and some form of error checking in gapless succession.

The preamble gives the receiver in the BiM time to stabilize. The preamble time may be increased to allow more time for carrier detection or receiver wakeup and initialization.

The packet's control area is used for many purposes. Primarily, it signals the beginning of a message. Other information such as packet number, byte counts, or flow-control characters may be included here.

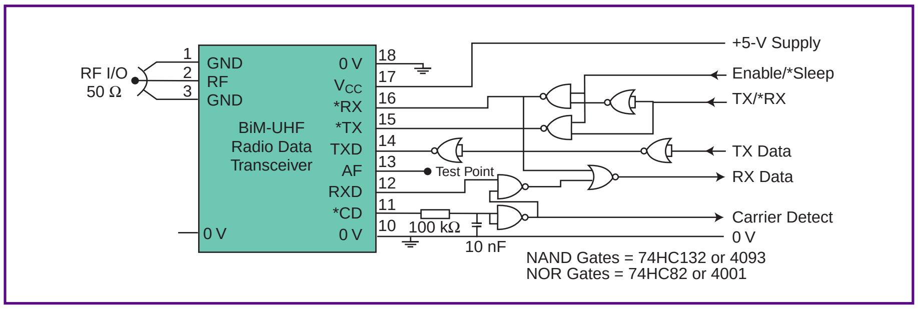


Figure 3—Adding this minimal number of components provides a fully buffered CMOS interface to the BiM. Note the pulse stretcher attached to the CD pin.

Listing 1—Visual Basic relieves the pain of getting the transmitted data. Here, we simply wait for the carrier detect to active.

```
Private Sub MSComm1_OnComm()
  Select Case MSComm1.CommEvent
    Case comEvCD
      If MSComm1.CDHolding
        Then txtwindow.Text = txtwindow.Text & MSComm1.Input
      End If
  End Select
End Sub
```

As well, an address area can be inserted between the control and data areas. The address data can be source or destination address information or a unique site identifier.

Data is data is data is data. The only restriction—keep the actual byte count as low as possible. In case of CRC or checksum errors, it's more bandwidth efficient to resend smaller packets, especially if there are multiple BiM stations that must communicate with each other or a central host site.

BASIC COMMUNICATIONS

Using Visual Basic 5 and the MSComm Control, let's assemble a software-based datascope that shows the raw data flowing between BiM's connected to a couple of embedded-PC serial and parallel ports. The hardware I use is depicted in Figure 3.

Each embedded PC is equipped with enough memory and disk space to accommodate Windows 95 and the Visual Basic 5 development suite. I chose the Teknor VIPer 806 and the Advantech PCM-4862 because they can support this kind of environment.

As well, the VIPer and the 4862 both have Ethernet adapters. I can develop the VB code on a third machine and transfer it electronically via the Florida Room LAN.

Third machine? Let's park the truck here and walk awhile. I use bunches of support tools that I don't talk about to make what you see here. But, there's one tool I use every day that you need to know about.

Listing 2—The `Form_Load` ***routine sets up the serial port. After that, it's just a click of the Transmit button to send the complete data packet.***

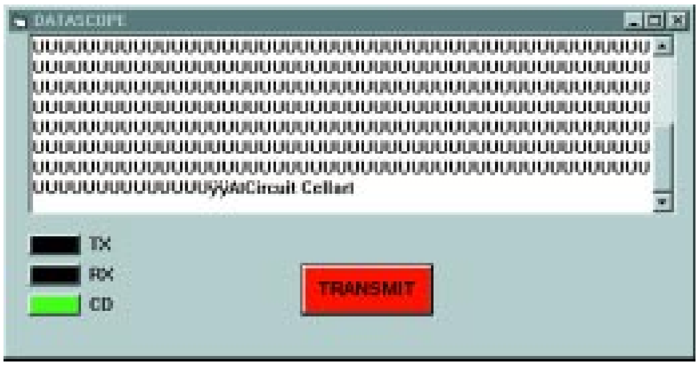
```
Private Sub Form_Load()          'init comm parms
  MSComm1.CommPort = 1
  MSComm1.Settings = "9600,N,8,1"
  MSComm1.InputLen = 0
  MSComm1.PortOpen = True
End Sub

Private Sub btnxmit_Click()      'set up storage and variables
  Dim x As Integer
  Dim xmitpreamble As Variant
  Dim xmitmsg As Variant
  Dim xmitcrc As Variant
  Dim msg As String
  Dim recbuf As String
  xmitcrc = 575                  'preload crc with 0xFF+0xFF+0x41
  xmitpreamble = Chr$(85)        'define message to send
  msg = "Circuit Cellar"
  msglen = Len(msg)
  xmitcrc = xmitcrc + msglen
  For x = 1 To msglen            'calculate a checksum
    xmitcrc = xmitcrc + Asc(Mid$(msg, x, 1))
  Next x
  xmitcrc = xmitcrc And &HFF
  xmitmsg = Chr$(255) & Chr$(255) & Chr$(65) & Chr$(msglen) _
            & msg &  Chr$(xmitcrc)  'send message packet
  For x = 1 To 700               'send preamble
    MSComm1.Output = xmitpreamble
  Next x
  MSComm1.Output = xmitmsg       'send message packet
End Sub
```

I'm constantly hooking up different embedded platforms for this and that. It's a drag to connect a monitor here, a keyboard there, and then swap them for another embedded PC project.

You either buy a bunch of monitors and keyboards and lay stuff all around, or do what I did. Get yourself a Vetra MegaSwitch.

I use the eight-port model (VIP 1328) in the Circuit Cellar Florida Room. Each port is equipped with a cable for video, keyboard, and mouse. By just pressing a button on MegaSwitch's front panel, a single monitor, keyboard, and mouse switches between eight embedded and/or desktop PCs. OK, back in the truck.



```
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUUU
UUUUUUUUUUUUUUUÿÿAICircuit Cellar!
```

Photo 3—It's all here, the preamble and everything else. The "A" begins our message, which is surrounded by a message length and checksum character.

MSComm

The MSComm control is VB5's way of controlling serial communications via available serial interfaces. With MSComm, you can open or close a serial port, set up the serial port parameters, and transfer data via the serial port's I/O structure.

I mentioned that VB5 was object oriented and event driven. Well, that's almost completely true. VB5 doesn't let certain object-oriented things happen that would let you corrupt original VB5 objects, but for our purposes that's OK.

By employing a property called CommEvent, we can interact with our BiM data by reacting to a change in events. In the case of our datascope, we will key on a change in state of the CD pin. Listing 1 describes how we will put MSComm to work.

PULLING DATA OUT OF THIN AIR

It's obvious that BiM can radiate bit patterns into electromagnetic space. So, it doesn't mean much to just send some data and say, "There…see."

That's why I built a pseudo-datascope so we can see what goes out and what comes in. From its beginning, Visual Basic has been strong in the user interface department. Photo 3 is a view of our datascope.

Here's the plan. Since BiMs are half-duplex devices, one PC will transmit, and the other will receive. The receiver runs code using MSComm's CommEvent property to trigger reception on a change in Carrier Detect from the BiM. If the change is a true detection of carrier, then the receiver will receive the RF-based datastream and display it on our datascope screen.

This is what we should see. First of all, the transmitting embedded PC is programmed to send a 700-ms preamble consisting of hex 55. This is approximately 700 8-bit characters at 9600 bps.

Once the receiving BiM locks in, two bytes of hex FF will be sent. This should look like a marking line to the UART.

The next character out will be a hex 41 or ASCII A. This is our control character and signals the VB5 application that real data is to follow. Listing 2 is the code snippet that created the view in Photo 3.

ROGER THAT...OVER

What we've just done is transmit and receive real data over the distance of a few feet. With low-power RF applications, that's sometimes all that is needed.

In Part 1, I discussed the good and bad of transmitting data over certain distances. In reality, where you are and over what distances you must communicate determine the power levels and frequencies you may use.

After speaking to the folks at Abacom and Linx, I found that other countries have far different attitudes about RF communications than we do here in the States.

To that, I say this: Don't use the FCC regulations for applications geared for other countries, and don't think FCC regulations are restrictive. The low-power RF modules I described are perfect for most applications.

For instance, inventory and POS applications don't need to pass data over a great distance. Your keyless entry system for your car is another good example. You don't unlock or lock it from a mile away. You're right there up close.

If you do find an application where a higher powered RF field is required, there is equipment available and FCC regulations that permit its use.

The BiM module is the basic building block for a multitude of RF-based applications. With careful selection of frequency, power output, and antenna characteristics, it can provide a solution to many of your RF data-transfer problems.

By establishing itself as an embedded building block, the BiM proves 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
RXM-418 UHF receiver, TXM-418 transmitter, LC series
Linx Technologies, Inc.
575 S.E. Ashley Pl.
Grants Pass, OR 97526
(541) 471-6256
Fax: (541) 471-6251
www.linxtechnologies.com

Abacom Technologies
67 Hamptonbrook Dr.
Etobicoke, ON
Canada M9P 1A2
(416) 242-3120
Fax: (416) 242-2697
abacomtech@compuserve.com

VIPer 806
Teknor Industrial Computers, Inc.
616 Cure Boivin
Boisbriand, QC
Canada J7G 2A7
(514) 437-5682
Fax: (514) 437-8053
www.teknor.com

PCM-4862
American Advantech Corp.
750 E. Arques Ave.
Sunnyvale, CA 94086
(408) 245-6678
Fax: (408) 245-8268
www.advantech-usa.com

IRS

416 Very Useful
417 Moderately Useful
418 Not Useful

Codesign

FEATURE ARTICLE

Richard Moseley

The Evolving Relationship Between Hardware and Software

Far too often, design schedules go awry due to the difficulty of coordinating hardware and software design teams. Richard brings us up to date on codesign tools and how they promise a smoother, faster development cycle in the future.

As pressures increase to bring products to market more quickly, design engineers are looking for ways to speed up design cycles and decrease the time required for debugging prototypes and fixing problems.

Of course, to accomplish their goals, they must find ways to do this without suffering the penalties associated with learning curves that result from using new design tools.

One promising solution is codesign—a synchronized collaboration between software- and hardware-development teams.

The most common approach for embedded-system design positions hardware and software teams on unconnected paths that are only reunited after the creation of the hardware prototype.

These individual design groups make presumptions about each other's contribution to the overall system, which are magnified and later surface as significant errors that can take a big toll on budgets and schedules.

In addition, time-to-market urgency typically mandates that fixes be carried out via software workarounds rather than hardware redesign. This approach can result in significant compromises to the performance of the finished system.

In a perfect world, hardware and software teams would work in harmony from initial design concept all the way through to benchmarking the finished system, staying in constant communication as the design reaches critical integration and test phases. The two groups would work together, and the interface between their designs would be constantly validated to identify problems early in the design cycle.

Employing the precepts of codesign, simulation and verification would be performed on hardware and software synergistically, resulting in robust designs with added functionality and shaving time off development cycles.

This next-generation system-level design formula embodies the creation of a virtual prototype—a speed-optimized combination of software simulation and hardware emulation that precisely mimics the target system, albeit at a slower speed in most cases.

This environment would feature a new generation of interoperable codesign tools for simulation and verification that identify and resolve errors during the specification and partitioning phases. The result: clearly defined parameters for hardware and software implementation.

Even though most concede that the isolation of hardware and software implementation paths is less than optimal, it's still the most common methodology used in designing embedded systems. Part of the problem lies in the fact that until recently, design-tool technologies haven't been available to orchestrate cooperative design efforts between hardware and software development.

However, the design-tool community has made great strides with this problem, and a number of significant new tool technologies have emerged over the past year.

But, sometimes even the best-laid plans suffer in implementation, and most codesign solutions are far from perfect. In truth, none of the codesign solutions available today are an optimal fit for every application since they are often largely based on the particular vendor's core competency, whether it be software or hardware emulation.

Each approach is more suited for a specific set of requirements. Even those showing the most promise are likely to endure slow adoption since they're regarded with skepticism until proven.

Significant barriers also lie in the organizations targeted by codesign-tool vendors. Even if robust codesign tools and environments were available, the infrastructure in place at most system OEMs could seriously hamper the tools' acceptance.

Problems such as a lack of communication between design teams, different management systems for hardware and software, and deep-seated biases in engineering managers contribute to an unfavorable climate for the implementation of codesign methodologies.

Although the potential benefits of implementing a codesign methodology outweigh the obstacles to its adoption, this is only true if the codesign is planned well in advance of the design project. For instance, the value of coverification is extremely high early in the design process.

With today's time-to-market demands, attaining design closure as soon in the process as possible is critical. If the coverification process can be carried out with a high degree of certainty, any later problems associated with integration and testing will be minimized.

A NEW PLAYING FIELD

Soon, however, the overall situation may change dramatically. Players in the codesign-tool arena seem to be migrating toward a common approach—a speed-optimized mix of software modeling and hardware emulation. A year from now, these tools are likely to be robust and remarkably similar in the features they offer.

Widespread adoption of codesign methodologies and tools isn't a question of "if" but "when." In fact, it would be beneficial for companies to work with codesign-tool vendors even this early. By doing this, they'd help shape tool features, while gaining the experience required to increase the probability of a successful design project.

With today's high-performance silicon technologies inexpensively combining tremendous computer horsepower with integrated peripherals and memory, it's no surprise the software content of today's embedded systems has exploded. Most estimates place the software-development cost for a typical system at well over half the total development budget—a reality that the system development-tool industry has practically ignored for years.

In addition to the hardware components, such as one or more CPUs, possibly a coprocessor (e.g., a DSP or graphics coprocessor), an ASIC or two, some memory, and assorted off-the-shelf parts, a typical embedded system includes software components like an RTOS, device drivers, and an embedded application.

CODESIGN IN DEVELOPMENT

Most aspects of the hardware and software development process can be automated, except in the cases of system definition, architectural design, and software/hardware partitioning. As well as not being well automated, these exceptions are fairly disconnected from the implementation process, which is itself divided into distinct hardware- and software-design efforts.

As a result, hardware and software engineers don't get to test their respective subsystems together until a physical prototype exists. Unfortunately,

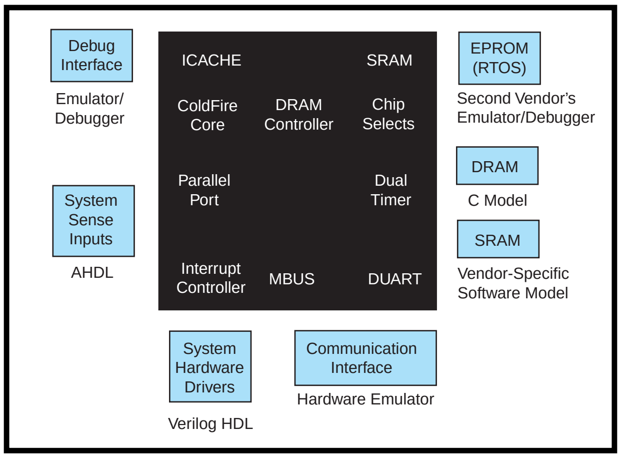


Figure 1—*Chip manufacturers are doing an excellent job of emulating the logic that goes onto a chip, as evidenced by shrinking debugger ports on new, complex chips. Next tasks: emulating the chip in both systems and standalone environments as well as eliminating bugs.*

most problems in the hardware/software interface aren't discovered until this point, possibly forcing a redesign. And so, schedules slip. Often, the integration and testing phase may represent the critical path or as much as 50% of the development cycle.

To counteract these problems, codesign solutions are extremely attractive. Unfortunately, the pieces aren't all in place yet. For instance, general-purpose automated tools for system definition, architectural design, and software/hardware partitioning that offer enormous leveraging over the subsequent phases of development are rare today.

In fact, most mainstream development-tool vendors focus on the implementation portion of the process (i.e., linking the software and hardware design phases of the project after partitioning). However, a lot can be gained by shortening the implementation/test phase.

Current codesign methodologies focus on two basic approaches to creating a virtual prototype. They attempt to shorten the "software waiting for hardware" gap encountered in most development efforts.

One camp, with roots deeply embedded in hardware and in-circuit emulation technologies, provides for the creation of the virtual prototype using a "black box" filled with FPGAs.

The other camp, whose technology originated from the high-level simulation and abstraction approaches of EDA, has developed unique techniques for speeding up logic simulation in order to build a virtual prototype.

Each approach has advantages and disadvantages, but one thing is clear. Both camps are on intersecting courses. Product developments will start to look quite similar as these new products continue to mature in the marketplace.

HARDWARE EMULATION

The integration levels and performance provided by today's FPGA vendors, plus the maturity of tools available for mapping the register-transistorlevel description of hardware, enable hardware emulation of a target system to be realized fairly quickly and easily.

The advantage of the hardware-emulation approach, led by systems such as Quickturn Design Systems' System Realizer, is that they provide gate-for-gate, wire-for-wire prototyping of the target system. In some cases, they may even be able to operate at the target system's full operating speed.

Motorola used System Realizer during the verification phase of the MC68060 and ColdFire microprocessor cores (see Figure 1). During the '060 verification, prior to first silicon, over one trillion instructions were run to verify that the chip was running instructions correctly, as illustrated in Figure 2.

The main downside to the hardware-emulation-only approach is the investment in front-end work required to map the system into programmable logic, although tools designed solely for this task greatly facilitate the process. Once the system is mapped, hardware changes can be quickly and easily implemented.

This step, however, must be completed at some stage of the design process anyway, to analyze the validity of the customer-designed logic. The only disadvantage is that the gate-level implementation must be completed at an earlier stage in the project.

SEAMLESS SIMULATION

On the simulation front, Mentor Graphics recently released its Seamless Co-Verification Environment (CVE). Using a unique approach to speeding up the verification process, this system enables the designer to minimize the time spent in logic simulation by isolating the CPU and memory from the rest of the system logic.

The CPU is modeled using an instruction-set simulator (ISS), which can run compiled C or assembly code at millions of instructions per second in full-emulation mode. The ISS tracks the number of CPU cycles required for each operation for performance analysis and synchronizing with the rest of the system. By abstracting the memory in addition to the CPU, a majority of the bus cycles, which are simply data or instruction fetches, can be avoided.

To handle concerns about accuracy or analysis at a deeper level, a bus functional model is also created in C to enable the logic simulator to stimulate bus activity and interact with the rest of the system logic. The memory system is also modeled in the logic simulator so it's visible to the rest of the logic. The memory and bus functional models communicate with the ISS using an API.

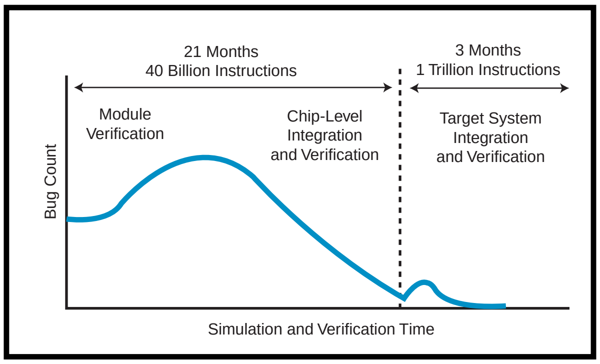


Figure 2—*The 68060 verification experience demonstrated that the verification process weeds out many bugs that would have slipped by through the sheer number of instructions run against the part. However, a certain number of bugs only manifest themselves in a full-blown end system that includes memory components, peripherals, and perhaps other processors.*

However, this approach alone is not enough, since waiting for the logic simulator to complete each operation renders it unusable. In addition, the contents of the memory model in the simulator must synchronize with the image expected by the CPU without tying up resource-intensive bus cycles.

To solve this problem, Seamless CVE employs an intelligent kernel between the ISS and logic simulator. It keeps the two simulations synchronized and maintains a consistent view of memory for the CPU and logic.

Most importantly, it optimizes the performance of the overall system simulation by letting the user selectively filter certain classes of bus activity to avoid unnecessary stimulation of the logic simulator. For example, once the instruction fetch cycle has been characterized for a portion of the memory map, fetches can be turned off to reduce load on the logic simulator.

The main disadvantage to the simulation approach is that it requires the development of a number of critical models—an ISS model for the target processor and any off-the-shelf VLSI components, a bus-functional model, and a memory model.

If models are not readily available from the chip vendor in a form that works with Seamless CVE, the time required to create them can add months to the overall development cycle. Plus, most vendors won't provide them even if they are available unless a binding nondisclosure agreement is signed and you're signed up to buy chips. Some vendors refuse to supply these models or even develop them to protect their intellectual property.

You can also include actual components in your virtual prototype if they're available, or you can do hardware emulation of blocks using Mentor's SimExpress box. Mentor's strategy is to enable designers to create the level of abstraction required.

Both approaches require hardware to be well-modeled and -defined before the creation of the virtual prototype, much more so than the software. The software then turns up bugs that require a redesign on portions of the hardware.

TOOLSETS

Before taking on a codesign methodology, examine the tools closely to ensure that specific project requirements can be met in the allotted time. Codesign tools should be adopted for smaller projects to give you time to learn the tools before committing the methodology to a critical design project.

A number of things should be considered before committing to codesign. Are the proper models available for the functions required? If not, how will the models be obtained?

What is the cost (in dollars and time) if models must be developed? If models must be developed, will the information be available to develop them?

What developments, if any, are planned by the chip or tool vendor that could aid the design process? What other tools will work with the target codesign toolset?

Codesign toolsets should embody a start-to-finish solution that ideally includes:

- a "real" fully-integrated hardware/software design environment
- software and hardware emulation with programmable breakpoints
- mixed hardware emulation and simulation with visibility at any level
- tool-vendor interface interoperability (a common interface standard)
- high abstraction levels to speed the simulation of large systems
- a flexible, distributed system that allows for multiple design seats
- minimized cost per design seat
- an investment outlay that's easy to amortize over large production runs
- a portable stand-alone detachable emulation board for system demos
- the ability to keep intellectual property secure for model or chip vendors

WHAT'S AHEAD?

As a methodology, codesign requires a fair bit of maturing before it can enjoy the kind of widespread adoption hoped for by the companies developing codesign products.

In the coming year, many new and upgraded products are anxiously being expected from development-tool vendors that could push the methodology into acceptance at mainstream system OEMs.

But, it's certainly true that codesign has some other hurdles to overcome. Many of these rise primarily from the mind-set of engineers involved in hardware and software design.

For now, sit back and observe. Only time will tell what will develop. ▣

Richard Moseley, principal staff engineer with Motorola, currently works with integrated systems, directing a systems and design support team involved in ColdFire and 680x0 systems using the FlexCore standard cell-design methodology. Richard has been involved in semicustom library development, CAD, and design since 1982. You may reach him at moseley@oakhill.sps.mot.com.

SOURCES

Seamless CVE, SimExpress
Mentor Graphics, Inc.
8005 SW Boeckman Rd.
Wilsonville, OR 97070
(503) 685-7000
Fax: (503) 685-1205
www.mentorg.com

MC68060, ColdFire
Motorola
MCU Information Line
P.O. Box 13026
Austin, TX 78711-3026
(512) 328-2268
Fax: (512) 891-4465

System Realizer
Quickturn Design Systems, Inc.
440 Clyde Ave.
Mountain View, CA 94043
(415) 967-3300
Fax: (415) 967-3199
www.quickturn.com

I R S

419 Very Useful
420 Moderately Useful
421 Not Useful

FEATURE ARTICLE

Norman Bujanos

Choosing the Right Crystal for Your Oscillator

Although critical to the timing of your design, crystals are glossed over in engineering schools. Norman puts a stop to that. His review of crystal parameters gets you up to snuff when it comes to picking the right crystal for your design.

Selecting quartz crystals for oscillators can be confusing and mysterious. Engineering programs tend to give them only a cursory overview.

Hence, many engineers are unfamiliar with crystal parameters and jargon. A crystal datasheet can appear to be as cryptic as Egyptian hieroglyphics.

In this article, I cover crystal parameters and explain how system design can affect clock accuracy. It should make crystal selection significantly easier.

WHY QUARTZ CRYSTALS

The quartz crystal integrates mechanical and electrical characteristics. If quartz is stressed, an electric field is generated in the direction perpendicular to the applied stress.

Conversely, if an electric field is applied to a quartz crystal, a mechanical stress appears in the direction perpendicular to the applied stress. This effect, known as the piezoelectric effect, is the basis for quartz being used so extensively in crystal manufacturing.

By placing a quartz crystal between two electrodes and applying a changing voltage, the crystal can be made to vibrate. Maximum vibration amplitude occurs when the frequency of the changing voltage matches the crystal resonant frequency. Oscillator circuits using a quartz crystal vibrate at the crystal resonant frequency.

High Q is one of the most desirable features of quartz crystals. It is a measure of how much energy is lost due to vibration. In mechanical terms, Q is:

$$Q = 2\pi\left(\frac{\text{Energy stored per cycle}}{\text{Energy lost per cycle}}\right)$$

In electrical terms, Q is the inductive reactance at resonant frequency divided by the equivalent series resistance (ESR).

A crystal with a high Q loses little energy while vibrating. Commercial-grade crystals have Qs ranging between 20,000 and 200,000. High-precision crystals have Qs up to 3 million.

In addition to high Qs, quartz crystals tend to be incredibly stable. The only drift associated with crystals is from temperature fluctuations and aging. Temperature effects are about 100 ppm over the operating range, while aging effects are around ±5 ppm per year.

TIMING BUDGET AND ACCURACY

When selecting a crystal, carefully consider how accurate your system has to be. Crystal selection and oscillator design must be weighed equally.

These factors work together, influencing the system operating frequency and cost. Typically, the more accurate a system is, the more expensive it is to build.

If you're building a system with an RTC, your target accuracy should be ±2 min. per month. PLL reference clocks, however, can tolerate less accuracy.

Four crystal parameters play a key role in system accuracy. The contribution from each must be added to obtain the total system accuracy.

Each parameter can be adjusted without influencing the others. This parameter independence makes customizing crystals attractive.

However, customization results in increased system cost. Look carefully at your system requirements. Try to use readily available, off-he-shelf crystals.

FREQUENCY TOLERANCE

The frequency tolerance (i.e., calibration effect) is the first of the four accuracy-budget parameters. It is a room-temperature (i.e., 25°C) spec stating how close the actual crystal frequency is to its specified frequency. Like most crystal specs, it is in parts per million.

For example, a 32,768-Hz crystal may have a frequency tolerance of ±20 ppm. At 25°C, the resonant frequency can be anywhere between 32,768.65536 and 32,767.34464 Hz.

The 32768 crystal is known as a watch crystal. This system might sound highly accurate, but when you consider its accuracy impact over a month, this one parameter alone can cause the watch to be off by almost 1 min.

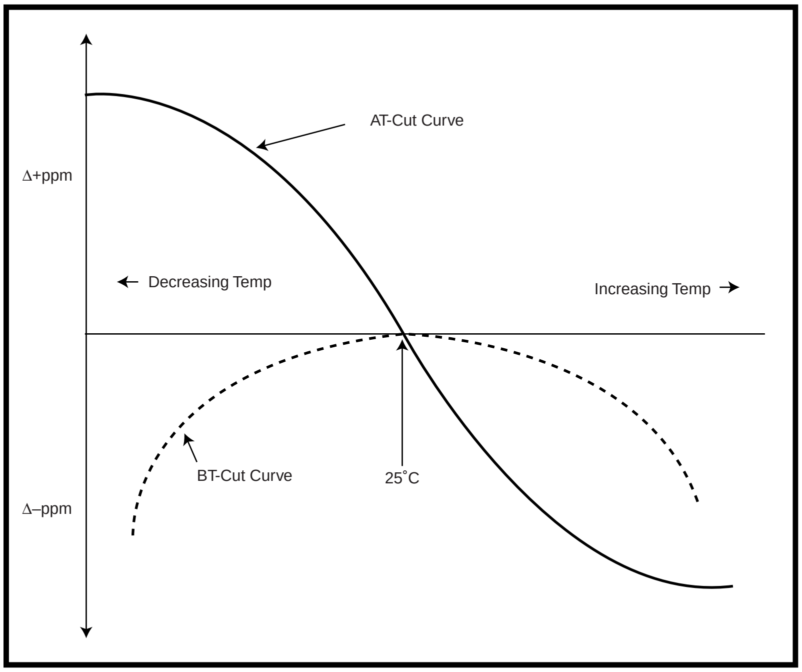


Figure 1—*AT-cut crystals (solid curve) exhibit a cubic temperature stability curve. On the other hand, BT-cut crystals (dashed curve) exhibit a parabolic temperature stability curve.*

Equation 1 shows that a ±20-ppm frequency tolerance can account for about 52 s per month:

$$60\frac{\text{s}}{\text{min}} \times 60\frac{\text{min}}{\text{h}} \times 24\frac{\text{h}}{\text{day}} \times 30\frac{\text{days}}{\text{mon}} \times \left(\pm\frac{20}{1{,}000{,}000}\right) = \pm 51.84\frac{\text{s}}{\text{mon}} \quad (1)$$

FREQUENCY STABILITY

Frequency stability is the second item to add to the timing budget. It is a function of temperature and is related to the crystal cut type.

The most common crystal cut types are AT and BT. Their temperature stability curves are different—a fact that should be considered when you're designing a system.

The AT curve is cubic [1], as depicted in Figure 1. Note that the curve moves between the +ppm and –ppm areas with temperature.

If a system using an AT-cut crystal is exposed to temperature fluctuations, the temperature effects tend to average to zero over time. However, an error is introduced from not operating at 25°C (i.e., crystal calibration temperature).

The BT cut, common in low-frequency crystals, is a parabolic form. Increasing or decreasing temperatures both cause a decreasing resonant frequency. Unlike the AT-cut crystal, temperature fluctuation effects do not average to zero.

AGING

The third item in the timing budget is aging as it relates to crystal contamination and drive level. Resonant frequency changes as a function of time.

It tends to be related to crystal contamination. That is, particles either drop off or fall onto the quartz surface. Because this happens inside the crystal case, there isn't anything you can do about it. It's up to the manufacturer.

However, keeping the drive level low can reduce aging effects as the crystal is not knocked around as much. Choose crystals that are hermetically sealed for best aging characteristics.

LOAD CAPACITANCE

The fourth timing budget parameter to consider has to do with load capacitance. For parallel resonant circuits, there is a load-capacitance spec.

If the load capacitance of your circuit doesn't match the crystal load capacitance, there is a resonant frequency shift. I'll discuss this effect shortly.

One nice characteristic of quartz crystals is that all of these parameters are independent. Each one can be adjusted without affecting other parameters.

You can request a crystal with 5-ppm frequency tolerance, and keep the other parameters the same or change them. However, you pay a premium for custom-made crystals.

SERIES AND PARALLEL RESONANCE

The question of parallel and series resonant crystals often comes up and is occasionally a source of confusion. Let me clarify the situation.

There is no such thing as a series or parallel resonant crystal. Instead, crystals have different parallel and series resonant frequencies.

When a crystal is calibrated at the factory, it is trimmed to hit a particular frequency while operating in the series or parallel resonant mode. The parallel resonant frequency is greater than the series resonant frequency.

Most oscillators operate in the parallel resonant mode (i.e., they see a parallel load capacitance). Some examples of parallel resonant oscillators are the Pierce-, Colpitts-, and Clapp-style oscillators. Series-resonant oscillators, on the other hand, are uncommon.

Transforming mechanical parameters into electrical parameters is known as creating the electrical dual. The equivalent electrical circuit for a crystal is shown in Figure 2. Components C_1, L_1, and R_1 make up the crystal's motional arm.

Co is the shunt capacitance. It is composed of packaging and lead effects, and is on the order of a few picofarads. *Co* is also known as the crystal's static capacitance.

L_1 is the crystal's motional inductance. This value is determined by the crystal's motional mass during oscillation, and is on the order of thousands of henries.

C_1 is the crystal's motional capacitance. It is determined by the crystal's

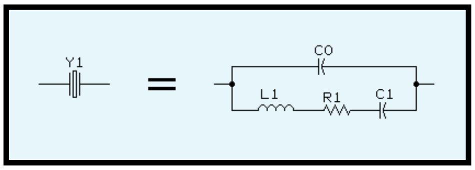


Figure 2—*The mechanical properties of mass, friction, and stiffness are mapped to inductance, resistance, and capacitance, respectively.*

stiffness, and is on the order of a few femtofarads.

R_1 is the crystal's ESR when oscillating, and it is related to mechanical loss during oscillation. ESRs range from a few ohms to tens of thousands of ohms.

If the ESR is small, the crystal loses little energy while vibrating. A small ESR helps with startup and continued oscillation.

The series equivalent circuit for a crystal omits the shunt capacitor, *Co*. The crystal series resonant frequency is:

$$Fs = \frac{1}{2\pi\sqrt{LC}} \qquad (2)$$

When crystals are connected to PC boards, they see a circuit that looks like Figure 3. Here, *CL* is equal to the series combination of CL_1 and CL_2, and is attributed to board parasitics and/or load caps added to the oscillator. The resonant frequency changes from equation 2 to:

$$Fp = Fs\sqrt{1 + \frac{C1}{Co + CL}} \qquad (3)$$

In most cases, *Fp*, the parallel load resonant frequency, is specified in the crystal datasheets. C_1 and *Co* are part of the crystal, but the load capacitance, *CL*, is not.

At the factory, the crystal is calibrated (frequency-tolerance spec) with a particular load capacitance. This number appears in the datasheet as the load capacitance.

If your load capacitance doesn't exactly match the load capacitance in the datasheet, your oscillator won't run at the spec *Fp* frequency. (I look at the effects of mismatched load capacitance in the next section.) Note that the parallel resonant frequency is greater than the series resonant frequency.

FREQUENCY TOLERANCE AND LOAD CAPACITANCE

When the oscillator circuit load capacitance doesn't equal the crystal spec load capacitance, the oscillator's operating frequency is different from the crystal's frequency tolerance spec.

Equation 3, for *Fp*, shows that as the board attributed load capacitance increases, *Fp* decreases. The change in frequency as a result of mismatched load capacitance is:

Crystal Specs

Here is a list of the crystal specs, along with an abbreviated description and typical values.

Nominal Frequency—This is the ideal crystal frequency, or target frequency at 25°C. Typical values are ±20 ppm.

Frequency Tolerance—This error is associated with the crystal calibration at 25°C. Tolerances range from ±5 to ±200 ppm.

Frequency Stability—This error is associated with temperatures away from the calibration temperature. Most crystals are calibrated at 25°C. Moving away from this temperature causes a shift in resonant frequency. The temperature-dependence curve depends on the crystal cut type.

The AT cut provides the best temperature curve. Typical AT numbers may be ±100 ppm across the operating temperature range, although you can get down to ±10 ppm.

BTs follow parabolic temperature curves. Resonant frequency decreases with temperature changes from 25°C. Typical spreads are –100 ppm across the operating temperature range.

Long Term Stability—Aging characteristics are largely a function of contamination. Keep the drive level low to reduce aging effects. Typical aging is around ±5 ppm per year.

Load Capacitance—If the load capacitance of your system (oscillator plus board) does not exactly match the crystal load capacitance, there is a resonant frequency shift, which you can calculate via equation 4. Typical load capacitance values are around 20 pF.

Operating Mode—Try to stay with the fundamental-mode crystals. If you have a high-frequency oscillator (i.e., greater than 50 MHz), use overtone crystals. Be aware of the pitfalls mentioned in the Mode of Operation section.

Drive Level—The crystal drive level is how much power the crystal can safely dissipate. Typical numbers are in the microwatt to milliwatt range. The exact drive-level equation is rather involved. However, a close approximation can be made from the following assumptions.

Referring to Figure 4, at resonance, the impedance of the motional arm composed of L_1, C_1, and R_1 is equal to the impedance of *CL* and *Co*. Power dissipation is given by:

$$P = I^2R_1$$

Since the current through the crystal is generally unknown, it is more useful to write the power as:

$$P = \left(\frac{V}{|Z|}\right)^2 \times R_1$$

where $|Z|$ is the impedance magnitude of *CL* and *Co*, and *V* is the peak voltage across the crystal. The drive level is:

$$P = \left(2\pi \times freq \times V\left(Co + CL\right)\right)^2 \times R_1$$

where *freq* is the resonant frequency [2]. For a parallel resonant circuit, the impedance goes to infinity. This implies that the net current goes to zero, but in reality, energy is lost through friction and joule heating. Also, the currents through the motional and capacitive arms are sinusoidal and 180° out of phase.

$$\Delta Fp = Fp_1 - Fp_2 = Fs\sqrt{1+\frac{C_1}{Co + CL_{spec}}} - \sqrt{1+\frac{C_1}{Co + CL_{system}}} \quad (4)$$

where Fp_1 is the spec parallel resonant frequency, Fp_2 is the actual parallel resonant frequency, CL_{spec} is the crystal spec load capacitance, and CL_{system} is the system load capacitance.

Equation 4 is known as the pullability equation and gives the frequency error of mismatched load capacitances. Often, this error is insignificant. However, it does come into play when there is a cumulative effect. If the crystal is used in a timekeeping application, cumulative effects are important.

If you're trying to tightly control accuracy, you must consider PCB stray capacitances. Routing to the crystal and socket effects, if used, also add to the load capacitance.

It may be necessary to use a trim cap to hit the target accuracy. If the circuit load capacitance is less than the target load capacitance, add a parallel trim cap to the circuit. Connect the cap between either the crystal pin or ground.

If the circuit load capacitance is greater than the target load capacitance, a series trim cap should be added to the circuit. The trim cap is connected to either crystal pin and the corresponding oscillator pin.

AT VS. BT CUT

If you take a look at the two temperature coefficient curves for the AT- and BT-cut crystals (see Figure 1), you find that the AT cut is preferable. For the BT, temperature changes always cause a resonant frequency decrease.

Unfortunately, you may not have a choice. For low-frequency crystals below 1 MHz, BT cuts dominate. Oscillators using these style cuts may not perform well in environments with large temperature fluctuations. But, if you have the choice, pick the AT cut.

MODE OF OPERATION

The crystal mode of operation largely depends on the operating frequency. Up to 50 MHz, the mode of operation is fundamental. Above 50 MHz, the mode is probably overtone.

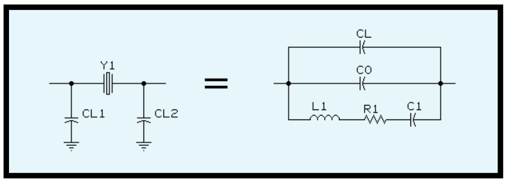


Figure 3—*When the external load capacitance* CL *is taken into account, it appears as a capacitor in parallel with* Co.

Note that overtone frequencies are not harmonics of the fundamental frequency, although they are close. Harmonics are exact integer multiples, while the overtones are not.

However, overtone frequencies are always odd multiples of the fundamental frequency. Both AT- and BT-cut crystals are available for overtone use.

Overtone operation is a nontrivial effort. Oscillators that run over 50 MHz must run in the overtone mode.

You don't see high-frequency fundamental crystals because the crystal becomes too thin. Crystal thickness is inversely proportional to resonant frequency, so high frequencies translate to thin crystals. And, thin crystals are expensive because they're difficult to manufacture and handle.

With overtone crystals, the thickness is greater than that of the fundamental crystal [3]. The overtone mode multiplies the thickness. As Figure 4 shows, a third overtone crystal is three times thicker than the comparable fundamental crystal.

There are a few disadvantages to using overtone crystals. The first is that the oscillator must be designed to specifically operate at the overtone frequency and not the fundamental frequency. Therefore, the oscillator must contain a filter to avoid the fundamental frequency.

Another disadvantage is that overtone crystals tend to be thicker than the fundamentals. This translates into larger ESRs, hence lower Q. Care should be take to ensure reliable oscillator startup and operation.

A third disadvantage is that overtone crystals can contain spurs (i.e., short for spurious mode, an unwanted type). The crystal manufacturer has to make sure that spurious modes are sufficiently suppressed. If they aren't, the oscillator can run at the wrong frequency.

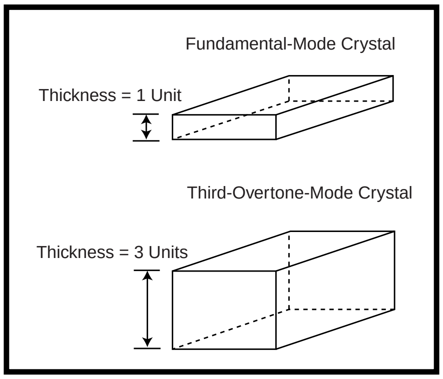


Figure 4—*The thickness of the third-overtone crystal slab is three times that of the fundamental-mode crystal slab.*

PACKAGE CONSIDERATIONS

Crystals are available in a variety of packages. There are many metal-can configurations, plastic packages, and surface-mount plastic.

Unlike ICs, the plastic version is probably the most expensive. This is because the plastic version is the metal version encapsulated in plastic. In other words, you pay for two packages.

Handling considerations should be the only reason to select one package type over another. Performance is the same.

CRYSTAL PLACEMENT

The old real-estate adage "Location, location, and location" applies to crystal placement. Closer to the oscillator is better.

You want to minimize parasitics introduced by long PCB traces. The board traces add to the *CL* value in the crystal model. Remember, a *CL* that does not match the *CL* in the specs causes a frequency shift (see equation 4).

Also, look around the crystal area to see if there are any other clock signals or otherwise frequently changing signals nearby. These signals can introduce noise into the oscillator. A good (quiet) ground plane under the oscillator can help eliminate noise problems, too.

Consider the distance between the two crystal lead traces as well. This distance adds to the *Co* term in the crystal model. Keep the traces apart by at least the same distance as the crystal width.

CRYSTAL CLEAR?

Hopefully, terms like BT cuts, third-overtone mode, and parallel resonance no longer send chills down your back. This brief overview of all the crystal specs should give you a bit more confidence when it comes to dealing with them.

As I mentioned, for accuracy, you need to consider four important specs. I discuss them in more detail in the "Crystal Specs" sidebar.

The sum of the errors contributed from each spec is the total system timing error. Since system cost is probably an issue, do not overspecify the crystal. Try to determine how much timing error your system can tolerate. Then, select the appropriate crystal using the information given here [3,4].

The other crystal parameters deal mostly with how the crystal is being used. For high-frequency applications, you'll almost certainly need an overtone crystal. For very low-frequency applications, it will be fundamental but a BT cut.

Basically, you want to design the best system you can at a particular price. Invest the time in planning for a good system. ■

Norman Bujanos is a member of the technical staff at Advanced Micro Devices in Austin, Texas. He is an analog circuit designer for the Logic Products Division. He received a B.S. in Physics from the University of Houston and an M.A. in Physics from the University of Texas at Austin. You may reach him at norman.bujanos@amd.com.

REFERENCES

[1] M.E. Frerking, *Crystal Oscillator Design and Temperature Compensation*, Van Nostrand Reinhold Co., New York, NY, 1978.
[2] T. Williamson, *Oscillators for Microcontrollers*, Microcontroller Technical Marketing App. note AP-155, Intel, June, 1983.
[3] Ecliptek Corp., www.ecliptek.com.
[4] Cardinal Components, www.cardinalxtal.com.

I R S

422 Very Useful
423 Moderately Useful
424 Not Useful

EMI Gone Technical

MICRO SERIES

Joe DiBartolomeo

Surge Suppression

Part 1 of 4

Caught in an EMI eddy? Joe's here to help you out. He begins by reviewing how to prepare for EMI threats, such as lightning, electromagnetic discharge, fast transients/bursts, and inductive load switching.

Last year, I wrote a MicroSeries on the most common electromagnetic compatibility (EMC) standards and tests mandated for digital equipment by the FCC and European Community (*INK* 79–82). As I pointed out, these EMC tests and standards have become de facto design specifications.

This MicroSeries builds on last year's. I begin by looking at common types of electromagnetic interference (EMI) transients.

I then present components used to both protect equipment from transient EMI and aid in passing the EMC tests. These components include metal oxide varistors (MOVs), zener diodes, transient voltage suppressor (TVS) semiconductors, and spark gap devices.

I'll end the series with a look at design philosophies and techniques for protecting electronics from EMI and passing the EMC tests.

These articles are intended as general design aids. Due to the nature of EMI, the equipment designer is the best judge of what techniques and components solve EMI problems.

USING EMC TESTS AND STANDARDS

Before discussing EMI threats, I'd like to take a moment and discuss my philosophy on the use of the EMC standards and tests.

I received several calls and E-mail messages regarding last year's Micro-

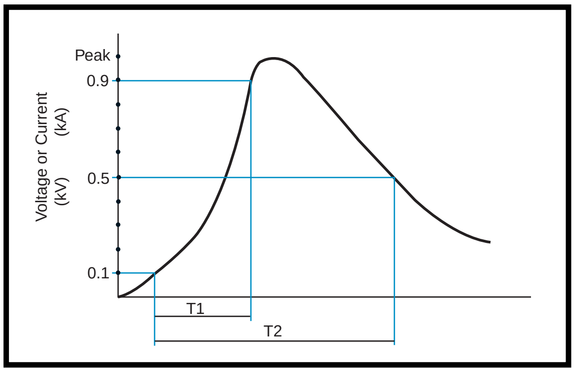


Figure 1—*This double exponential curve is representative of a lightning-induced transient. T1 represents the rise time from 10 to 90% of peak value. T2 represents the fall time to 50% of peak value. The most commonly used double exponentials are the 8/20 and 1.2/50. The 10/700 and the 10/1000 are double exponentials that are commonly used in telecom applications.*

Series. During these discussions, I noticed that many designers assume that if their equipment passes the prescribed EMC tests, then they won't have any EMI problems in the field. Unfortunately, this may not be the case.

EMI/EMC tests are general in nature. It's impossible to set tests and standards that apply to all equipment and operating conditions. Therefore, it's conceivable that your equipment will pass all the relevant EMI tests and have EMI problems during normal operation.

The first step in designing for EMC is to understand the tests mandated by the testing agencies. You can think of the EMC tests as the first EMI threat your equipment must endure. I treat the EMC tests and standards as minimum—but not necessarily sufficient—design specifications.

WHAT IS SUFFICIENT?

When I went to school, most of my classmates chose economics as their minor. Their rationale was that engineering and economics go hand in hand, cost versus performance curves, and so on. I, however, chose philosophy as my minor.

I'd love to say I chose philosophy for some esoteric reason like "taking the road less traveled," but that wasn't the case. I chose philosophy simply because it had the most women in the class.

One thing I did learn in those classes, however, is the concept of necessary and sufficient conditions. What is necessary to complete an objective may not be sufficient.

For example, the statement "I'm healthy because my diet is well balanced" is clearly wrong. Of course, ensuring that your diet is well-balanced is a necessary condition of good health, but it is not sufficient. Regular exercise is also necessary for good health, but that too is insufficient. You get the picture.

Getting back the EMC/EMI tests, it's clear that the design philosophy "Passing the EMI/EMC tests will ensure no EMI problems" is incorrect and may lead to trouble. The EMI/EMC tests are necessary and must be passed, but they may not be sufficient to ensure problem-free operation.

As an example, let's look at a piece of equipment being subjected to a lightning transient test (e.g., 1000-4-5). Assume the equipment passes the test.

However, keep in mind that you normally take new equipment to the test labs. Unfortunately, several of the components used to protect electronics from lightning surges degrade with use.

What happens in the field as protection degrades? The tests only deal with the equipment once—normally, when it's new.

The solution is to ensure that surge-protection devices that degrade are on a maintenance or replacement schedule. Here, obviously, the test standard is insufficient, and extra protection needs to be designed in.

I'm well aware of standard company policy when it comes to EMC tests: do the minimum for the equipment to pass the tests. There's no provision for adding extra protection. But remember the people in your company who set this EMC testing policy, usually the bean counters, are the first to run for the hills when and if an EMI problem arises.

I follow a general rule whenever I design something that I must sign off. I never take design advice from anyone who has less at stake than I do if something goes wrong.

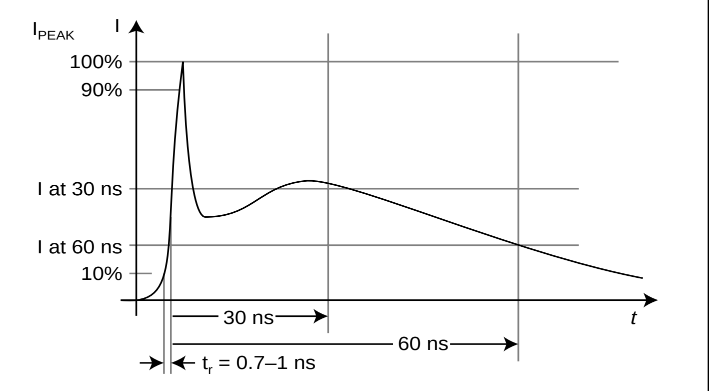
I refuse to lose my professional engineer's license because somebody wants to save a few bucks. I'm not trying to sound melodramatic, but as most designers know from experience, when something goes wrong, it's a lonely world.

I want to make this point early rather than waiting till the article on design techniques because it's important to understand the distinction between necessary and sufficient. Now, let's look at common EMI transient threats.

WHAT'S THE THREAT?

One of the most fundamental principles of protection is to define the nature of the hazard. I did some of that last year, but I want to expand on the EMI threats here.

By developing qualitative measures for EMI threats, you're able to



Severity Level	Voltage (kV)	First Peak (A)	Rise Time (ns)	Current @ 30 ns (A)	Current @ 60 ns (A)
1	2	7.5	0.7	4	2
2	4	15	0.7	8	4
3	6	22.5	0.7	12	6
4	8	30	0.7	16	8

Figure 2—*In a current waveform produced by an ESD gun, the charge is transferred in a very short period of time (i.e., <100 ns). The large, extremely fast spike at the start of the waveform simulates the initial charge transfer or spark that occurs when a charged human body comes in contact with a conducting object. The ESD-gun waveform parameters are established by the ESD test standard IEC 1000-4-2.*

	Rise Time	Fall Time	Peak Voltage	Peak Current
Lightning	1–10 µs	50-1000 µs	6 kV	10 kA
ESD	0.7–1 ns	60 ns	15 kV	>16 A
EFT/B	5 ns	50 ns	4 kV	N/A
Switching Inductive loads	1.2 µs	50 µs	4 kV	N/A
NEMP	5 ns	250 ns	100 kV/m	N/A

Table 1—*As you can see, the common causes of transient EMI present their own unique design problems. For example, lightning is slow but contains a great deal of energy, whereas ESD contains much less energy than lightning but has extremely fast rise times.*

understand the stresses that your electronic equipment will be subjected to and thereby ensure that the equipment is properly protected.

EMI is either conducted or radiated. It can originate from outside (i.e., lightning) or it can be an internal problem (i.e., when the microprocessor clock causes problems in an adjacent circuit).

When designing for EMI protection or solving EMI problems, it's useful to divide the EMI threats into these two broad categories. Here, I deal with conducted EMI threats. I'll address radiated threats in a later article.

CONDUCTED EMI

Conducted-EMI threats can be subdivided into two broad categories—transient events and steady-state signals. No doubt, you recognize this approach from circuit analysis.

I use the word "event" for transients because it points out the singularity of transients. Many similar transient events may occur one after the other or together, but they should still be regarded as individual events.

The term steady-state signal is also quite appropriate. Note that what I refer to as steady-state EMI (e.g., power-supply steady-state EMI), others may call circuit noise.

The transient EMI event (e.g., lightning) can be characterized as being a high-energy pulse of short duration. The transient EMI event is unpredictable and nonperiodic with very fast rise and fall times.

In contrast to transient EMI events, steady-state EMI signals tend to have much less energy content. They are generally periodic and predictable.

The fact that they are periodic means they normally have much lower frequency content than their transient EMI cousins. An example of a steady-state EMI signal is microprocessor clock noise on the power lines.

The solution to both transient and steady-state conducted EMI is to filter out or divert the interference away from the affected electronics. But clearly, the filter technique that works for transient EMI rarely works for steady-state EMI, and vice versa.

Filtering transients requires the filter to turn on quickly and absorb or divert a large amount of energy for a very short time. By contrast, when filtering steady-state EMI, the filter turn-on time is not very important and the power-handling capabilities are normally much less.

I'll concentrate on the transient EMI signals, as these are less well understood than the steady-state EMI signals and their effects are generally more dramatic.

CAUSES OF TRANSIENT EMI

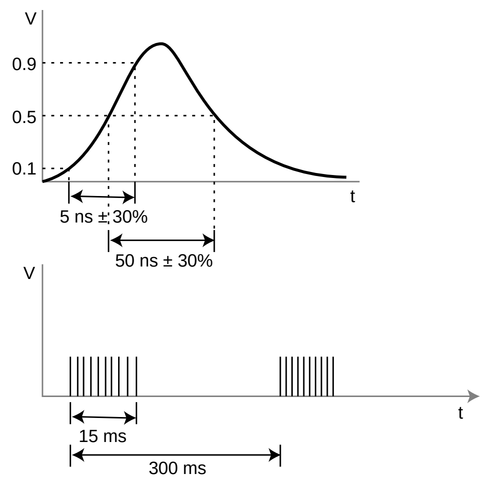
The most common transient EMI threats are lightning, electrostatic discharge (ESD), electrical fast transients/bursts (EFT/B), and the switching of inductive loads. A much less common EMI threat is the nuclear electromagnetic pulse (NEMP). Let's look at these threats one at a time.

As Steve Ciarcia and Jeff Bachiochi discuss ("Ground Zero," *INK* 90), lightning is caused by a separation of charge. When the potential is large enough to ionize a path, we get lightning.

The most common types of lightning are cloud to cloud, intercloud, St. Elmo's Fire, and cloud to earth. The cloud-to-earth lightning—although it only accounts for about 15% of all lightning strikes—is normally of greatest concern to designers of electronic equipment. However, all forms of lightning can cause problems for electronic equipment.

A cloud-to-earth strike occurs when the potential difference between the cloud and the earth is large enough to ionize the cloud-to-earth path. The lightning strike normally consists of two or more strokes (with a rare maximum of about 40 individual strokes).

The duration of each stroke is in the order of microseconds, and the peak current ranges from 2 to 200 kA with rise times to peak currents in the 0.1–10-µs range. The large amount of energy contained in a lightning strike, coupled with the rise times, makes it practically impossible to protect equipment from a direct light-



Severity Level	Power Supply Ports: Voltage Peak (kV)	Power Supply Ports: Repetition Rate (kHz)	I/O Ports: Voltage Peak (kV)	I/O Ports: Repetition Rate (kHz)
1	0.5	5	0.25	5
2	1	5	0.5	5
3	2	5	1	5
4	4	2.5	2	5
X	Special		Special	

Figure 3—*The EFT/B generator produces a train of pulses which lasts 15 ms during a 300-ms time frame. The individual pulses are well-defined. The voltage level of the applied voltage* V *depends on the severity level applied. The calling standard specifies severity levels for various test voltage levels and repetition rates.*

ning strike. Thankfully, direct strikes are rare.

Of greater concern to designers of electronic equipment are indirect lightning strikes. With about 100 cloud-to-earth strikes every second worldwide, chances are your equipment will be subject to indirect lightning strikes.

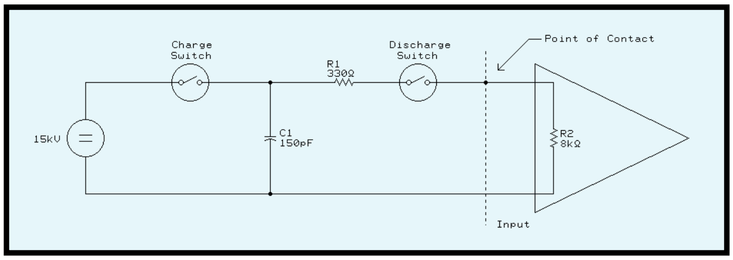


Figure 4—*Using the schematic of an ESD gun (Figure 2,* INK *81, p. 63), we can calculate the voltage and currents resulting from an ESD event. Here, an RS-232 input with an input impedance of about 8 kW is subjected to a 15-kV ESD. The result is virtually the full 15 kV appearing across the input.*

Unfortunately, when it comes to lightning strikes, defining the hazard is very difficult. Of course, we must come up with waveforms that define the transients that lightning produces. However, it's important to keep in mind their limitations and to understand that the test waveforms depend to some extent on your equipment.

Why are transients induced by lightning so hard to define? First, lightning is a natural phenomenon that occurs in an environment—outdoors—that has a great many variables. Therefore, quantifying lightning is difficult.

Note the values I gave above for lightning: peak current of 2–200 kA and a rise time of 0.1–10 µs. These are hardly ranges you can plug into your circuit-analysis program.

Another reason it's difficult to develop lightning transient waveforms is that the coupling path is unpredictable. Lightning can couple into a network via direct conduction or indirect conduction, where lightning induces a voltage on a wire that then conducts it into a circuit or system.

When lightning is directly coupled in, you can usually identify the entry point (e.g., a power cable or I/O line) but it's difficult to quantify the coupling path. When the lightning transient is coupled in via indirect conduction, the best you can do to quantify the coupling path is to say the farther away the strike, the better.

As a general rule of thumb, a lightning strike 1 km away from a 1-m piece of bare wire induces a voltage of 100 V across the wire. Since propagation is through the air, coupling behaves as a square law.

Without exact knowledge of the source or coupling path, it's difficult to come up with test waveforms that simulate transients caused by lighting. However, industry and standards groups have done a good job of creating useful test waveforms, which are based on observation as much as calculation.

Several industry-standard waveforms model lightning-induced surges. Virtually every one of these waveforms is based on the double-exponential waveform (see Figure 1).

The double-exponential waveform is characterized by an exponential rise from 10% to 90% of the peak and an exponential decay to 50% of peak. The most commonly used waveforms are the 8/20-current and the 1.2/50-voltage waveforms. The first number is the rise time, and the second number is the decay time in microseconds.

Lightning is modeled as a short-circuit event with a peak current up to 10 kA or as an open-circuit event with peak voltages up to 6 kV. The peak voltage of 6 kV is the point at which most household receptacles flash over.

Other standard double-exponential waveforms are the 10/1000 and 10/700 most commonly used in telecom applications. The European Community uses 1.2/50 for power lines and 10/700 for telecom lines to describe the transient threat posed by lightning (see 1000-4-5).

Voltages for these waveforms range from 500 to 4000 V. Depending on the class of equipment and peak currents, voltages can be as high as 1 kA.

As transients go, lightning is a relatively slow event and is normally handled using air or carbon spark gaps or gas discharge devices.

ESD

Electrostatic-charge buildup is caused when two nonconducting objects come in contact and then separate, one gaining and one losing electrons. This is known as the turboelectric effect.

The most common cause of ESD is humans coming in contact with electronic equipment. When the charged body comes in contact with a conducting path (e.g., electronic equipment), an ESD event occurs.

As you may recall, one of the immunity tests is the ESD test, in which the equipment under test is subjected to a simulated ESD with varied levels of severity. Figure 2 (reproduced from *INK* 81, p. 63) shows the transient waveform associated with ESD and gives typical specifications. Note that the energy contents are much less than that of a lightning transient, but the rise and fall times are much faster (i.e., in the nanosecond range).

Arcing-type surge protectors, such as gas discharge devices, are too slow for ESD events. Generally, metal oxide varistors (MOVs) along with transient voltage suppressor (TVS) semiconductors protect against ESD.

EFT/B

Electrical fast transients or bursts are usually caused by the repetitive switching of inductive loads. The industry standard for EFT/B waveforms is shown in Figure 3 (reproduced from *INK* 81, p. 68).

MOVs and TVS semiconductors protect from EFT/B. For more information on EFT/B and ESD, please refer to "Standards for Electromagnetic Compliance Testing—Part 3: Immunity and Susceptibility" (*INK* 81).

SWITCHED INDUCTIVE LOADS

The switching of inductive loads is a special case of the EFT/B where the transient is a single event rather than a burst, as shown in Figure 3. When the current flowing through an inductive load is interrupted, a transient voltage is produced by the collapsing magnetic field.

These loads are everywhere (e.g., refrigerators, air conditioners, motors, etc.). The voltage is characterized by:

$$V = -L\frac{di}{dt}$$

where V is the induced voltage, L is the inductance, and di/dt is the current's rate of change.

The industry standard for a switched inductive load is the 1.2/50 double exponential shown in Figure 1, with voltage peaks in the 0.5–6-kV range depending on the class of instrument.

Keep in mind that the point at which the current is switched is also important. If the current is switched near the zero crossing, much less voltage is induced than if the current were switched near the peak of the voltage waveform. This fact is important if you have any control over the switching of the inductive load.

Once again, MOVs and TVS semiconductors are the normal method of protection.

NUCLEAR EM PULSE

A nuclear electromagnetic pulse (NEMP) is, thankfully, not a common EMI threat and is therefore not of concern to most designers. However, NEMPs are of great concern to the designers of military equipment.

When a nuclear device explodes, it creates an impulse of electromagnetic energy. The pulse rise time is in the 5-ns range with a pulse duration of ~250 ns.

The pulse produces fields in the 100-kV/m range, which of course, has devastating effects on most electronics. This concept was used in the movie *Broken Arrow*.

TRANSIENT THREATS

Table 1 summarizes the most common transient-producing EMI threats. As you can see, every EMI transient threat presents its own special problems to designers.

Lightning is a relatively slow transient threat, but it has a great amount of energy. On the other hand, an ESD has a very small amount of energy, but its rise time is extremely fast.

It's important to keep in mind that the characteristics of the transient event—the current and voltage amplitudes, rise and fall times, and duration—depend as much on the characteristics of the affected equipment as on the source of the transient and the coupling path.

Telecom applications generally use a 10/100 or 10/700 double exponential, rather than the more common 8/20 or 1.2/50 waveform. This is due to the nature of telecommunications networks, which tend to distribute the transient over many lines, thereby reducing its rise time and increasing the decay time.

Now that you've seen the waveforms of the most common EMI transients, a useful experiment would be to apply these waveforms to your circuit without protective devices and calculate the voltage and current levels.

Figure 4 shows a circuit that simulates an ESD being applied to an RS-232 input. The input impedance of the line is about 8 kΩ (assuming input capacitance and lead inductance are zero).

If you assume contact resistance is zero, the result, using the voltage divider rule, would most likely be the destruction of the RS-232 input. Although this example was very simple, with the addition of a little more detail, this technique is a powerful design tool.

Now that you have a feel for the transient threats that your equipment will encounter, we can turn our attention to protecting circuits. In the upcoming months, I'll discuss the components used to protect equipment from EMI transients. ■

Joe DiBartolomeo, P. Eng., has over 15 years' engineering experience. He currently works for Sensors and Software and also runs his own consulting company, Northern Engineering Associates. You may reach Joe at jdb.nea@sympatico.ca or by telephone at (905) 624-8909.

REFERENCES

Encyclopedia International, Vol. 10, Grolier, New York, NY, 1972.
MTL, Surge-Protection App note, 1993–1994.
MTL, App note AN9009, 1990.
KeyTek, Surge-protection test handbook, 1986.

I R S

425 Very Useful
426 Moderately Useful
427 Not Useful

FROM THE BENCH

Jeff Bachiochi

Choose the Right Vehicle Before Riding the Air Waves

Even with all the TVs, tapes, CDs, and VCRs, radio still persists. Jeff takes a look at dual-state modulation techniques to help you ensure that your message gets through.

Everyone has their favorite. Chances are, either yours has changed format over the years or your tastes have changed. Yet, we all still listen to that old medium that just won't quit—radio.

It was comforting growing up with your favorite station. That special DJ was like a good friend you could always count on.

Even with the availability of cassettes, CDs, TVs, and VCRs, radio continues to be a popular medium. And now, talk radio has its own following. Radio just won't die.

AM VS. FM

For a signal to be transmitted and received over a particular medium, the characteristics of the signal need to be modulated or changed.

Our vocal cords modulate the air, creating air-pressure waves that travel outward from our mouths. If these waves happen to enter our ear's cavity and in turn vibrate the eardrum, the pressure waves are demodulated into the original signal.

We can control the amplitude or overall amount of the air pressure. Loud sounds have large pressure waves, and quiet sounds have smaller pressure waves.

And, we can control the frequency of air-pressure waves. Low frequencies produce waves stretched apart, while higher frequencies create waves that are packed closer together.

Our vocal cords modulate air using both AM (amplitude modulation) and FM (frequency modulation). I could use the term AM (angular modulation) instead of FM, but it gets too confusing.

Angular modulation can be either frequency or phase modulation. Frequency modulation changes the carrier's frequency in direct proportion to the modulation's amplitude. Phase modulation (PM) changes the carrier's phase in direct proportion to the modulation's amplitude.

These differences are subtle. The FM carrier is at its maximum frequency deviations during the maximum and minimum amplitudes of the modulation signal. The PM carrier is at its minimum frequency deviations during maximum and minimum amplitudes of the modulation signal.

Since both FM and PM carriers have the same spectral properties, I'll use FM, one form of angular modulation, to lessen the confusion with AM (amplitude modulation).

AUDIO VS. DATA

AM and FM radio broadcasts are forms of analog modulation. When AM and FM are used for data transmission, the modulation format can be simplified.

AM becomes a carrier-present/carrier-absent function, where the presence of the carrier indicates one data state and the absence equals the opposite state.

FM can have two states as well, but a carrier is always present. Using FM, the two states are two frequencies equidistant from the original carrier—one above it and the other below.

Data transmission over the perfect medium (or close to it), a LAN, or phone line often uses multilevel modulation (i.e., four or more states, instead of two) to double the data transmitted in one bit time.

Both AM and FM can even be used together to increase the data/bit time. That's how we can get more and more throughput using the same antiquated POTS (plain old telephone service) delivery service.

Improving technologies enable us to get better at detecting multilevel transmissions. In this discussion, I won't delve into these more complex modulations.

Instead, I concentrate on a more error/interference-prone transmission medium—radio. And, I will compare only the simplest dual-state modulation techniques.

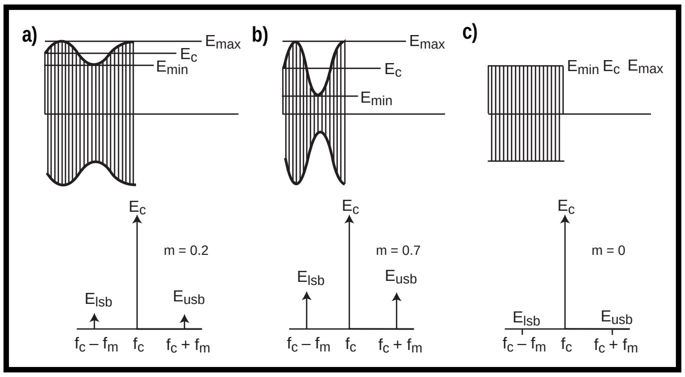


Figure 1—*The amount of amplitude modulation affects the amplitude of the sidebands and not that of the carrier. At 30% modulation, the sidebands are small* ***(a)****, while at 70%, the sidebands are large* ***(b)****, and at 0%, there are no sidebands* ***(c)****.*

ASK VS. SPECTRUM

Amplitude Shift Keying (ASK) was first used commercially by Marconi in 1896, when his Wireless Telegraph Company established the world's first permanent wireless installation at The Needles on the Isle of Wight, Hampshire, England.

Marconi used Morse code, developed earlier by Samuel Morse, to send wireless messages. (Actually, Morse amplitude modulated current flow in his telegraph system, but the medium was copper wires.)

The maximum modulation rate is generally a factor of 10–100 times less than the carrier frequency. The upper data rate for the AM radio band is about 15 kHz, and the lowest AM carrier is 36 times that (i.e., 540 kHz).

As audio modulates a carrier, upper and lower sidebands are created and transmitted along with the carrier. These sidebands are spaced at a deviation from the carrier equal to that of the modulation frequency, where:

$$\text{sideband}_f = \text{carrier}_f \pm \text{modulation}_f$$

as you see in Figure 1a.

When audio modulates a carrier between 0 and 100%, the sidebands increase in amplitude in relation to the percentage of modulation from no sidebands, at 0% modulation, to half the carrier amplitude, at 50% modulation (compare Figures 1a and 1b).

When data is transmitted using ASK modulation, no sidebands are created since the modulation varies between 0% and no carrier (see Figure 1c). Receivers require only a narrow bandpass filter. However, in-band noise has a large effect on the signal since it is playing one-on-one with the carrier.

FSK VS. SPECTRUM

Frequency modulation became a commercial institution in 1940 when Zenith Radio's Eugene McDonald started an FM radio station. However, it was Edwin H. Armstrong who first demonstrated FM's superior static-free radio reception.

Unlike AM, where legal modulation is between 0 and 100%, FM is expressed as a modulation index or the ratio of peak frequency deviation to modulation frequency.

As the amplitude of the modulation increases, the frequency deviation (and the modulation index) increases with respect to the modulation frequency, moving the sidebands further from the unmodulated carrier (see Figures 2a and 2b).

As the modulation frequency falls, the sidebands become rich in overtones. Remember, in FM, the center frequency (or unmodulated carrier) is not part of the FM transmission, unless of course, there is no modulation.

Unlike AM, where the total power transmitted was proportional to the data, the total power with FM is always the same. FM's change is one of sideband deviation.

In FM data transmission, this change consists of a increase or decrease in the overtones of the sidebands sharing the transmitted power.

POWER RESTRICTION

The FCC regulates all radio-frequency (RF) devices.

RF devices are defined as "…any device which in its operation is capable of emitting radio-frequency energy by radiation, conduction, or other means," whether it be incidental, unintentional, or intentional transmissions. Part 15 of the FCC regulations explains the conditions under which devices can be operated, licensed, and unlicensed.

Among the regulations is the definition of digital devices (formerly labeled computing devices) as a "…(device or

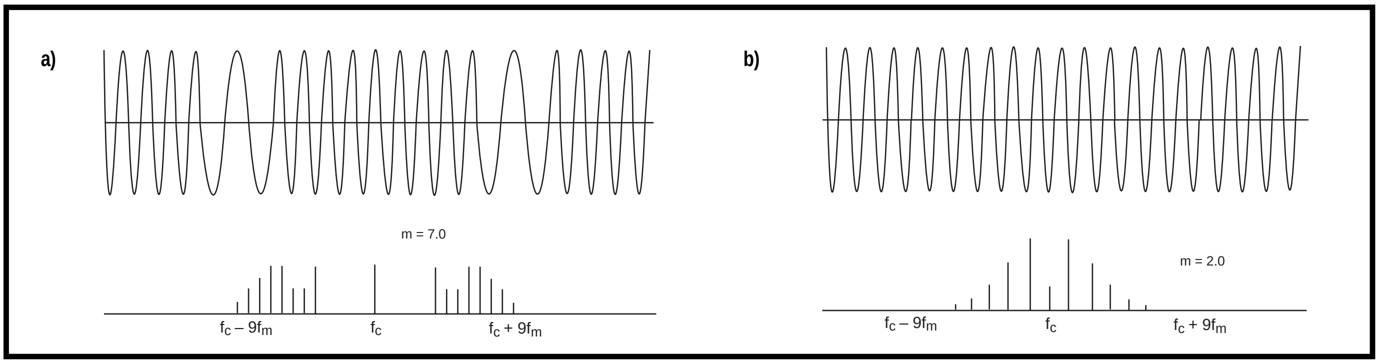


Figure 2a—*In FM, large modulation amplitudes move sidebands further from center frequency, while small modulation amplitudes* ***(b)*** *require less bandwidth.*

a)

Frequency of Emission	Unintentional Radiators (Class A digital device)
9–490 kHz	2400 µV/m
490 kHz–1.7 MHz	24000 µV/m
1.7–30 MHz	30 µV/m
30–88 MHz	90 µV/m
88–216 MHz	150 µV/m
216–960 MHz	210 µV/m
960+ MHz	300 µV/m

b)

Frequency of Emission	Intentional Spurious Radiators
40 MHz	225 µV/m
70–130 MHz	125 µV/m
130–174 MHz	125–375 µV/m
174–260 MHz	375 µV/m
260–470 MHz	375–1250 µV/m
470+ MHz	12,500 µV/m

c)

Frequency of Emission	Continuous Intentional Radiators (Fundamental)	Continuous Intentional Radiators (Harmonics)
902–928 MHz	50 mV/m	500 mV/m
2400–2483 MHz	50 mV/m	500 mV/m
5725–5875 MHz	50 mV/m	500 mV/m
24+ GHz	250 mV/m	2500 mV/m

Table 1—*Part 15 regulates maximum allowable emissions based on the signal types of unintentional radiators* ***(a)****, intentional spurious radiators* ***(b)****, and intentional continuous radiators* ***(c)****.*

system) which generates and uses timing signals or pulses at a rate in excess of 9000 pulses (cycles) per second."

Also of note is the ever-popular electronics kit, described as "any number of electronic parts, usually provided with a schematic diagram or printed circuit board, which, when assembled in accordance with instructions, results in a device subject to the regulations of this Part, even if additional parts of any type are required to complete the assembly."

Part 15 characterizes intentional radiators as those devices that emit radio-frequency energy. And, RF energy is defined as electromagnetic energy at any frequency in the radio spectrum between 9 kHz and 3,000,000 MHz. That just about covers everything we'd be talking about, huh?

Subpart B of Part 15 discusses unintentional radiators. That's where most devices and appliances fit (see Table 1a for the band vs. field-strength limits).

Subpart C reviews intentional radiators (i.e., transmitters), and these range from 9 kHz to well over 38 GHz. Within this range, there are many bands in which only spurious emissions are permitted, and radiated energy must fall within strict guidelines.

Table 1b lists the bands limited to these transmissions. For continuous operation, the bands are even more restricted (see Table 1c).

So, you can see that field strength isn't the only limitation put on a transmitter. The type transmissions are also restricted to certain bands. This means if you intend to operate an unlicensed transmitter, you need to identify where you fit in the FCC's regulations before you can determine what transmitter frequencies are available to you.

	Advantages	Disadvantages
AM	lower cost lower power consumption smaller size potentially twice the allowable output power	lower data rate poor noise immunity noncontinuous carrier
FM	higher data rate better noise immunity continuous carrier	higher cost higher power consumption larger size lower allowable power output

Table 2—*AM and FM techniques should be considered based on need since each has its own advantages or disadvantages.*

Why am I talking about the FCC regulations when I started off talking about AM and FM?

First, it's important to realize that you just can't arbitrarily choose any transmission frequency if you want your product to be legal. Second, because the FCC regulations have radiated-energy limits, the transmission distance depends on the type of transmission modulation you plan to use.

So, let's wrap this up by looking at the advantages and disadvantages of AM and FM.

WHICH WAY?

Like anything else in life, it's one big tradeoff. If the lowest cost, power consumption, and/or size is the ultimate goal, then AM looks like the best choice (see Table 2).

However, if a higher data rate and/or better noise immunity is of the utmost importance, then it might be better to go with FM. Then again, if it's not legal, it doesn't matter much, does it?

The moral is, when you think you found the solution, make sure to check the regulations. After all, while it might seem as if the FCC's sole purpose is to make life miserable, really, they're just here to protect us from each other. ■

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

REFERENCES

Text

American Radio Relay League, *The Radio Amateur's Handbook*, Newington, CT, 1998.

FCC, *Code of Federal Regulations*, Title 47, Part 15, 1995.

Internet

Spectrum Analysis, Amplitude & Frequency Modulation, Test & Measurement App note 150-1, www.tmo.hp.com

williams.cs.ncat.edu/modulate.htm

robotics.eecs.berkeley.edu/~sastry/ee20/modulation.html

www.physics.udel.edu/wwwusers/watson/student_projects/scen167/thosguys

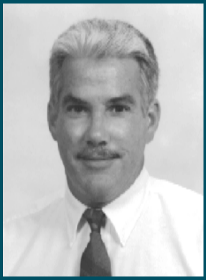
I R S

428 Very Useful
429 Moderately Useful
430 Not Useful

SILICON UPDATE

Tom Cantrell

Double-Duty DSP



What first appears as just another DSP turns out to be much more. Zilog's Z89*xx*3 DSP offers the usual DSP goods and acts like many highly integrated controllers for a fraction of the cost.

The earnest marketing fellow from Zilog didn't get more than a few foils into his presentation about their DSP to Ken and me before I butted in.

I'm definitely a left-coast kind of guy (er, dude). You may think you're giving me a presentation, but I see every meeting as a potential rap session. Meanwhile, Ken is one of those New England "salt of the earth" types who use words sparingly, as if stocking up for winter.

Anyway, the poor Zilog fellow said something about RISC and DSP that convinced me my comments would, or at least should, be welcomed. I suppose my two cents could have been helpful, but I have to admit I ran up the tab to at least a quarter.

Eventually realizing I was chewing up the allotted meeting time at a prodigious rate, I finally deigned to allow the poor guy to rush through the rest of his pitch. The usual stuff about MIPS and MACs, high performance, low cost, and so on. To tell the truth, I was already mentally filing under "Yet Another DSP."

Ken hadn't said much during the meeting—not that he could have gotten a word in edgewise. It was only as we were leaving and I mentioned my somewhat ho-hum assessment that Ken finally spoke, "You know, there might be something to that."

DSP STEREO(TYPING)

It was one of those moments that prompts a breakthrough in self-awareness. I realized I was guilty of the same sin of conventional wisdom complacency that I always warn against.

In the CPU versus DSP debate, I've been inspired by the work of professor Michael Smith. He demonstrated way back in '92 (including "To DSP or Not to DSP," *INK* 28) that a decent CPU-pipelined Harvard RISC enhanced with key DSP features (e.g., fast multiplier) could match the signal-processing performance of DSPs.

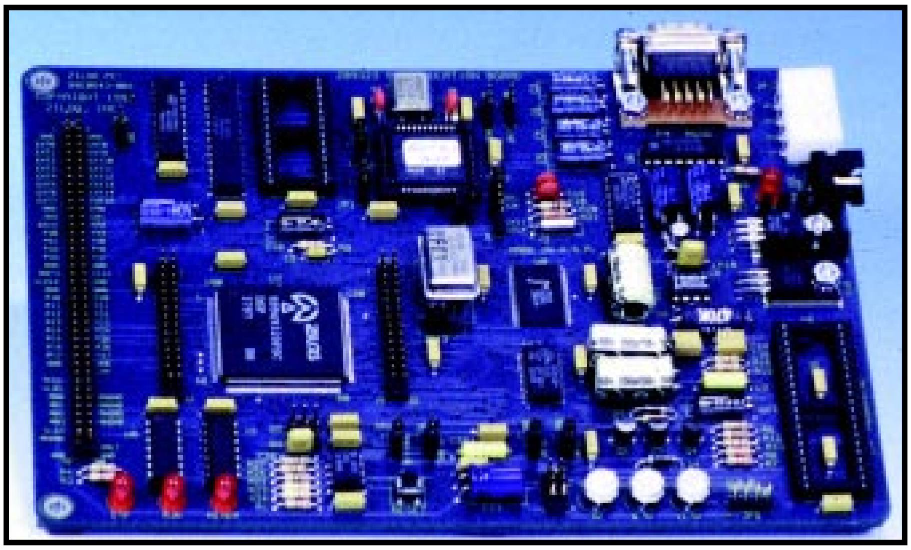


Photo 1a—*Whether the new Zilog 'x3 is a DSP or a micro, you've still got to hook it up and get it working. Zilog offers a low-cost evaluation and emulation board that connects to a PC via a serial port and includes an OTP programmer.*

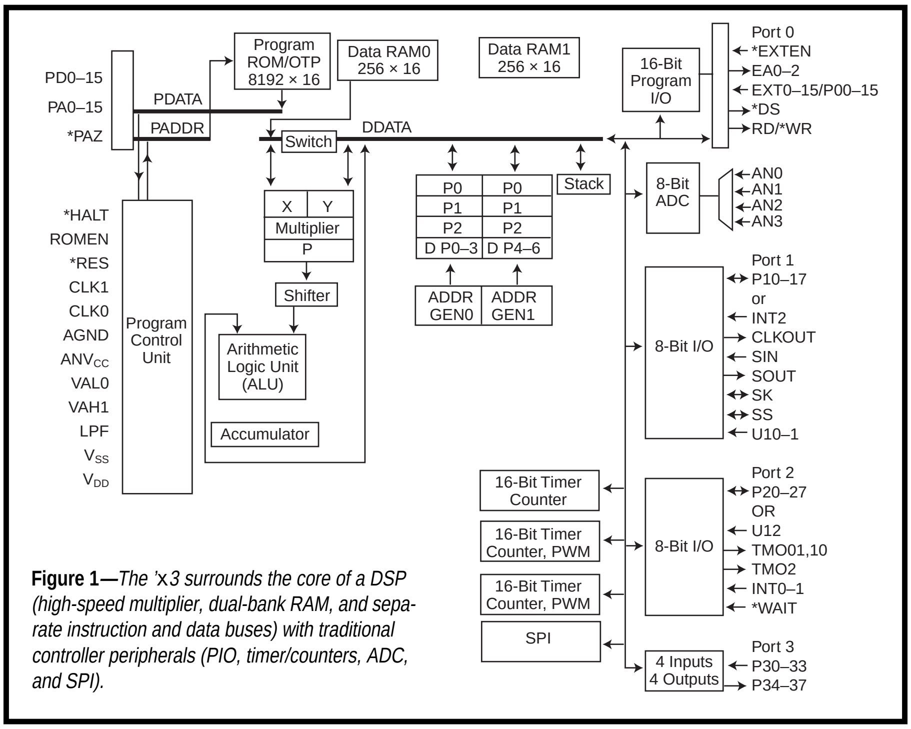


Figure 1—*The 'x3 surrounds the core of a DSP (high-speed multiplier, dual-bank RAM, and separate instruction and data buses) with traditional controller peripherals (PIO, timer/counters, ADC, and SPI).*

Today, you needn't look further than your own desktop, where MMX and similarly pumped CPUs crunch through audio, image-processing, compression, and modem algorithms formerly deemed the province of DSPs.

Reinforcing my bias, I had a number of negative conceptions about DSPs. Pricey, hard to program and debug, weak handling anything besides DSP loops, finicky hardware interface, and so forth.

Ken's comment prompted me to stick to my own advice: Question authority, including your own. If CPUs are becoming more like DSPs, doesn't that mean DSPs are automatically becoming more like CPUs? I decided to give the Zilog Z89*xx*3 16-bit single-chip DSP a closer, and impartial, look.

Even without popping the hood, a glance at the sticker dispels price misconceptions. Though high-end DSPs that I've covered in the past (like Analog Device's SHARC and TI's VelociTI) come with double- or even triple-digit price tags, the Zilog parts are less than $10 for low-volume OTPs and dive below $5 for high-volume masked ROMs.

TWO CHIPS IN ONE

Along with the price tag, the '*x*3 block diagram in Figure 1 further highlights the similarity with mainstream MCUs. Certainly nothing odd about the ADC, counter/timers, SPI, parallel I/O, and expansion bus.

Indeed, the high-speed multiplier and dual data RAMs are the only distinctly DSP features. They're also the focal point for critical design tradeoffs that achieve good performance at such a low price.

For instance, 16-bit operands would normally call for a 32-bit ALU and accumulator to avoid overflow. However, when determining the resolution required, consider the source—most likely an on-chip four-channel eight-bit ADC.

No sense demanding accuracy beyond the precision of the input. So, instead of a full 32-bit result, the '*x*3 only totes the most significant 24 bits.

Though the 24-bit fractional format (i.e., sign bit followed by fractional power of two bits, such as ½, ¼, ⅛...) can't represent 1.0, it gets darn close with 0.9999999. Furthermore, 8-bit data and 24-bit resolution mean we can multiply and accumulate up to 256 samples before worrying about overflow, which just happens to perfectly match the 256-word depth of the data RAMs.

Similar economizing takes place on the shifter front. Instead of the full barrel shifter found on expensive chips, the '*x*3 gets by with conventional single-bit left and right shifts, supplemented by a three-bit right-shift option. Of course, simply multiplying by the appropriate fractional power of two duplicates the function of the right-shift portion of a barrel shifter (e.g., multiply by 1⁄32 to shift right five bits).

Remember that clever design may offset left-shift laments. For instance, one Zilog app note discusses connecting a single eight-bit SRAM to the 16-bit expansion bus. Since the chip has no built-in bus matching or sizing capability, this calls for software to morph 16-bit transfers into dual-byte accesses. It seems there's no escape from chugging through eight left-shift instructions.

The trick? Connect the SRAM to every other expansion-bus bit, and a single shift is all it takes to switch between the even and odd bits.

LINGUA FRACTAL

As with the block diagram, only a small portion of the architecture (see Table 1) uniquely brands the '*x*3 as a DSP, most notably the one-clock multiply (`MLD`), multiply and accumulate (`MPYA`), and multiply and subtract (`MPYS`) instructions.

There's also a bit of looping support with the Loop address-mode option

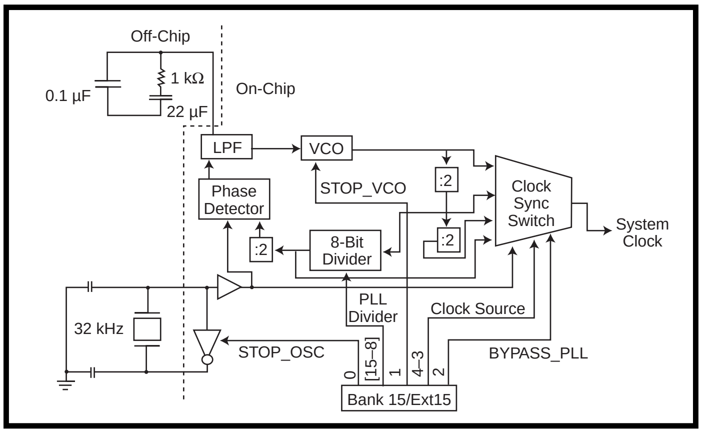


Figure 2—*The 'x3 clock generator is about as good as it gets, certainly far superior to the plain oscillator found on most controllers.*

that tweaks conventional pointer auto-increment and -decrement with modulo rollover. Cost consciousness dictates a bit of a compromise. That's why only power-of-two loop sizes between 2 and 256 are supported.

Since the '*x*3 is a true Harvard architecture, the `<memind>` addressing mode is provided to access data (e.g., constants, tables, and coefficients) stored in program (EP)ROM.

Other than these overtly DSPish features, there's little to distinguish the '*x*3 from conventional controllers. It has kind of a "Z8 on steroids" flavor (in fact, the $99 assembler offered by Zilog supports both the Z8 and '*x*3). Otherwise, it's business as usual (e.g., `LD`, `JP`, `CALL`, `AND`, `OR`, etc.).

Of course, the '*x*3's claim to fame is that most of the instructions are single word and execute in a single clock. Two exceptions are when the long (16 bit) immediate (two word, two clock) and indirect (one word, three clock) address modes are used. Since they're relatively rarely encountered, that leaves two clock branching (i.e., `JP`s, `CALL`s, `RET`s) as the main MIPS versus megahertz derater.

`JP`s and `CALL`s are conditional, but timing is the same whether the branch is taken or not. Besides the usual flags (`Z`, `C`, `OV`, etc.), two pins (UI0 and UI1) and their inversions are patched into the condition codes. The '*x*3 also supports conditional execution for typical accumulator operations (`INC`, `DEC`, `NEG`, shifts, and rotates), cutting the number of `JP`s required.

The interrupt scheme furthers the cause of fast and predictable response. It isn't fancy, mind you. No intricate programmable priority or nesting schemes, and the stack is just six deep. However, it only takes two clocks (after completing the current instruction) to save the PC and vector to the handler.

All interrupt sources (including on-chip I/O and three external programmable edge inputs) are mapped to one of three internal interrupt vectors. The idea is to pick the two most critical interrupt sources and map them to vectors 0 (highest priority) and 1.

Then, any or all of the remaining sources are assigned to vector 2 (lowest priority), recognizing some polling will be required if more than one is enabled. Any source that isn't assigned to one of the three internal vectors is de facto disabled.

MY KINGDOM FOR A CLOCK

Before going further, I'd like to single out the '*x*3 clock generator, shown in Figure 2, for special praise. Often overlooked, clocking is a key issue with intrinsic relation to system cost, reliability, power consumption, EMI, and timing capability.

Zilog could have gotten by with the conventional crystal oscillator, since 20 MHz arguably doesn't push the envelope. Nevertheless, they went to the trouble of providing a PLL-based clock generator that works with a cheap, rugged, low-EMI 32-kHz watch crystal. While the '*x*3 isn't the first (and won't be the last) to exploit PLL, Zilog's design is notably pragmatic and versatile.

As shown in Figure 3, a variety of dynamically programmable options comprise a four-tier low-power regime.

a)

Symbolic Name	Syntax	Description
<pregs>	Pn:b	Pointer Register
<dregs> (points to RAM)	Dn:b	Data Register
<hwregs>	X, Y, PC, SR, P EXTn, A, BUS	Hardware Registers
<accind> (points to Program Memory)	@A	Accumulator Memory Indirect
<direct>	<expression>	Direct Address Expression
<limm>	#<const exp>	Long (16 bit) Immediate Value
<simm>	#<const exp>	Short (8 bit) Immediate Value
<regind> (points to RAM)	@Pn:b	Pointer Register Indirect
	@Pn:b+	Pointer Register Indirect with Increment
	@Pn:b–LOOP @Pn:b+LOOP	Pointer Register Indirect with Loop Decrement
<memind> (points to Program Memory)	@@Pn:b	Pointer Register Memory Indirect
	@Dn:b	Data Register Memory Indirect
	@@Pn:b–LOOP	Pointer Register Memory Indirect with Loop Decrement
	@@Pn:b+LOOP	Pointer Register Memory Indirect with Loop Increment
	@@Pn:b+	Pointer Register Memory Indirect with Increment

b)

Register	Description	Register	Description
P	Multiplier output (24 bit)	EXT1	External data port 1 (16 bit)
X	X multiplier input (16 bit)	EXT2	External data port 2 (16 bit)
Y	Y multiplier input (16 bit)	EXT3	External data port 3 (16 bit)
A	Accumulator (24 bit)	EXT4	External data port 4 (16 bit)
SR	Status register (16 bit)	EXT5	External data port 5 (16 bit)
Pn:b	Six RAM address pointers (8 bit)	EXT6	External data port 6 (16 bit)
PC	Program counter (16 bit)	EXT7	External data port 7 (16 bit)
EXT0	External data port 0 (16 bit)		

c)

Instruction	Description	Instruction	Description
ABS	Absolute value	NEG	Negate
ADD	Addition	NOP	No operation
AND	Bitwise AND	OR	Bitwise OR
CALL	Subroutine call	POP	Pop stack
CCF	Clear carry flag	PUSH	Push stack
CIEF	Clear interrupt enable flag	RET	Subroutine return
COPF	Clear overflow protect flag	RL	Rotate left
CP	Compare	RR	Rotate right
DEC	Decrement	SCF	Set carry flag
INC	Increment	SIEF	Set interrupt enable flag
JP	Jump	SLL	Shift left logical
LD	Load	SOPF	Set overflow protection flag
MLD	Multiply	SUB	Subtract
MPYA	Multiply add	XOR	Bitwise XOR
MPYS	Multiply subtract		

Table 1a & b—*The speedy multiply and program memory address modes are the only clues that the '*x*3 is a DSP.* **c**—*The instruction set is quite terse, indeed more "reduced" than that of many RISCs.*

	Z89223-ROM Z89273-OTP	Z89323-ROM Z89373-OTP		Z89393-EXT.ROM
Pin Count	44-Pin	68-Pin	80-Pin	100-Pin
Package	PLCC/QFP	PLCC	PQFP	PQFP
P0[15:8]	EXT, P0, P1	EXT, P0	EXT, P0	EXT, P0
P0[7:0]	EXT, P0	EXT, P0	EXT, P0	EXT, P0
P1[7:0]		P1	P1	P1
P2[7:0]	P2[4:0]	P2	P2	P2
P3[7:0]			P3	

Table 2—*The ’x3 lineup starts with the ’223/’273 in 44-pin packages and moves up to the ’323/’373 with 68 or 80 pins. The 100-pin ’393 supports external ROM up to 64K × 16 with a separate bus (16 address and 16 data lines).*

The most miserly Stop clock mode shuts everything down.

However, those who’ve dealt with PLLs before know that slow wakeup is the price to pay for deep sleep. The ’*x*3 is no exception, needing a leisurely 10 ms to get up and running.

If that’s a problem, select the option that leaves the PLL running for fast recovery. Another choice has the ’*x*3 run directly off the 32-kHz crystal, again with or without the fast-recovery option. Notice the selection of wakeup sources, including INT0, the SPI SS (Slave Select), a PIO bit (U10), or all three.

KITCHEN SINK

The ’*x*3 may be brainy, but does it have the I/O brawn to match highly integrated controllers? Though the chip has a complete selection of peripherals, the devil may be in the details, so let’s take a closer look.

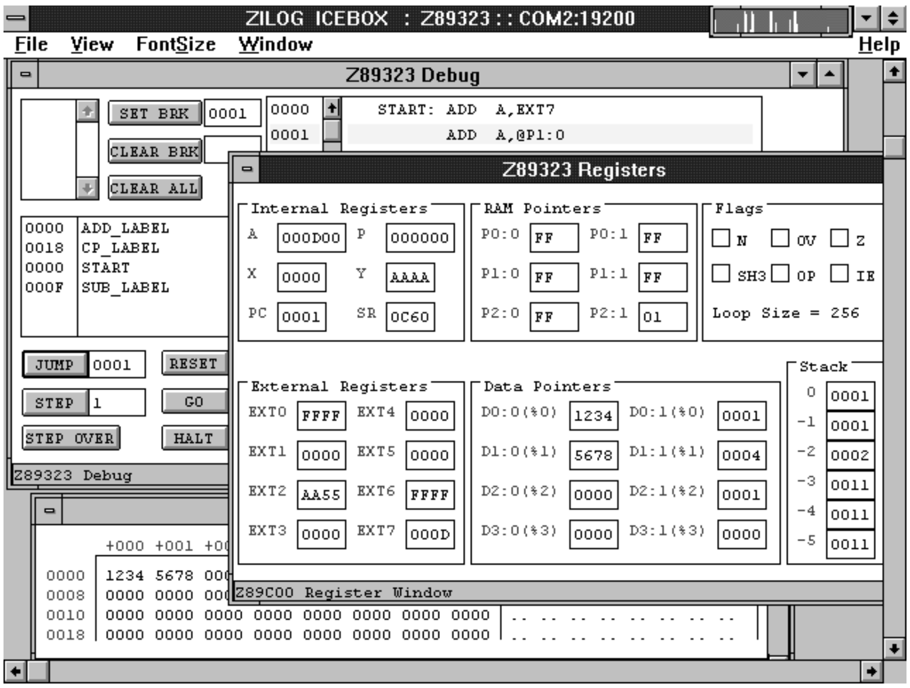
The ’*x*3 is offered in five different packages (listed in Table 2), which basically fit into two categories—big (68+ pins) and small (44 pins). Though the 100-pin version does offer 64K word program expansion, it’s the 16-bit EXT (external) bus that’s intended to accommodate application-specific add-ons.

The three-address-line limit (EA0–2) isn’t simply a case of pin shortage. In fact, EXT0–EXT7 are all the ’*x*3 knows about because they’re hardwired into the opcode map.

Since EXT is also used for accessing the on-chip I/O, the ’*x*3 relies on a banking scheme. Various combinations of internal and external registers are mapped into 16 banks, with EXT7 (which selects the bank and consolidates interrupt status) common across all banks.

The EXT bus, mapped directly into the register file, is fast (i.e., one clock). If timing is tight, take advantage of the on-chip one-wait state generator and *WAIT pin. Of course, you can always configure one or both bytes of EXT as regular PIO (outputs configurable as push-pull or open-drain) and do everything in software, especially with so many MIPS on tap.

DSPs are known for delivering bus bandwidth, but how does the ’*x*3 fare on controllers home turf (e.g., the ADC, timers, SPI, etc.)? With relatively modest expectations, I was surprised to find that these functions


```
ZILOG ICEBOX : Z89323 :: COM2:19200
File  View  FontSize  Window                                   Help

Z89323 Debug
SET BRK   0001        0000   START: ADD  A,EXT7
CLEAR BRK             0001          ADD  A,@P1:0
CLEAR ALL

0000  ADD_LABEL
0018  CP_LABEL
0000  START
000F  SUB_LABEL

JUMP  0001    RESET
STEP  1       GO
STEP OVER     HALT
Z89323 Debug

Z89323 Registers
Internal Registers      RAM Pointers            Flags
A  000D00  P  000000    P0:0  FF   P0:1  FF     □ N    □ OV  □ Z
X  0000    Y  AAAA      P1:0  FF   P1:1  FF     □ SH3  □ OP  □ IE
PC 0001    SR 0C60      P2:0  FF   P2:1  01     Loop Size = 256

External Registers      Data Pointers                           Stack
EXT0 FFFF  EXT4 0000    D0:0(%0) 1234   D0:1(%0) 0001           0  0001
EXT1 0000  EXT5 0000    D1:0(%1) 5678   D1:1(%1) 0004          -1  0001
EXT2 AA55  EXT6 FFFF    D2:0(%2) 0000   D2:1(%2) 0001          -2  0002
EXT3 0000  EXT7 000D    D3:0(%3) 0000   D3:1(%3) 0000          -3  0011
                                                               -4  0011
                                                               -5  0011
Z89C00 Register Window

      +000 +001 +00
0000  1234 5678 000
0008  0000 0000 000
0010  0000 0000 0000 0000 0000 0000 0000 0000 | .. .. .. .. .. .. .. ..
0018  0000 0000 0000 0000 0000 0000 0000 0000 | .. .. .. .. .. .. .. ..
```

Photo 1b—*Traditional controller users will feel right at home with the 'x3 Windows-based development software included with the evaluation and emulation board pictured in Photo 1a.*

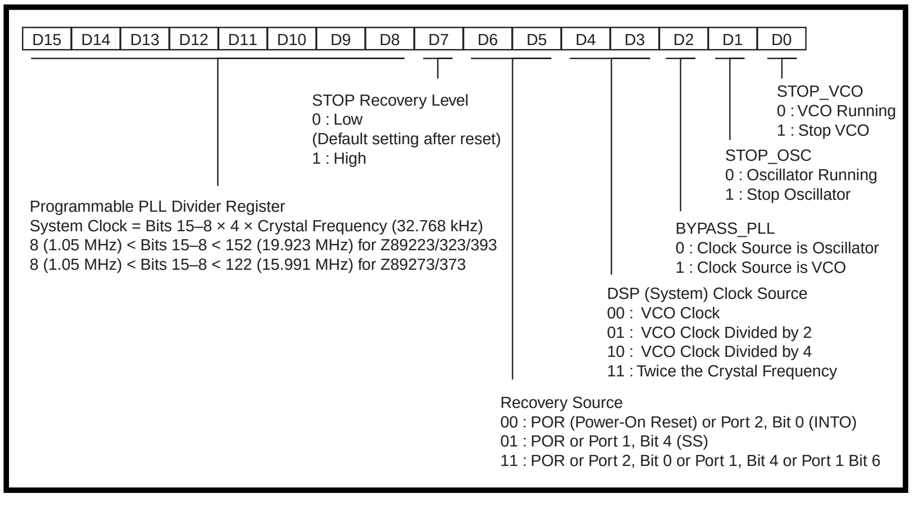
are not only competitive with controllers, but quite a bit better than most.

For instance, the four-channel eight-bit ADC offers separate high and low reference voltage inputs. Of course, you can just connect them to digital power and ground, achieving the equivalent of chips without dedicated pins. However, separate references are often preferred for higher accuracy, especially considering noise and drift on the digital side.

The ADC is quite fast (2.0 µs) and is rather versatile and easy to use. For instance, besides program control, a conversion can also be initiated by an interrupt input (INT1) or countdown of one of the timers. Conventional single channel conversion is supplemented with modes that gather four readings from a single channel or one from each channel.

The three 16-bit timers prove that '*x*3-peripheral know-how extends into the digital realm as well. Two of the units feature eight-bit prescaler, auto-reload, and a variety of modes including square wave, PWM, one-shot, gated-count, triggered count, and period measurement. There are even two watchdog variants—one retriggered by software and one by pin input.

The third 16-bit unit isn't as whizzy but does offer the unique advantage of being able to run off the 32-kHz time-



STOP_VCO
0 : VCO Running
1 : Stop VCO

STOP Recovery Level
0 : Low
(Default setting after reset)
1 : High

STOP_OSC
0 : Oscillator Running
1 : Stop Oscillator

Programmable PLL Divider Register
System Clock = Bits 15–8 × 4 × Crystal Frequency (32.768 kHz)
8 (1.05 MHz) < Bits 15–8 < 152 (19.923 MHz) for Z89223/323/393
8 (1.05 MHz) < Bits 15–8 < 122 (15.991 MHz) for Z89273/373

BYPASS_PLL
0 : Clock Source is Oscillator
1 : Clock Source is VCO

DSP (System) Clock Source
00 : VCO Clock
01 : VCO Clock Divided by 2
10 : VCO Clock Divided by 4
11 : Twice the Crystal Frequency

Recovery Source
00 : POR (Power-On Reset) or Port 2, Bit 0 (INTO)
01 : POR or Port 1, Bit 4 (SS)
11 : POR or Port 2, Bit 0 or Port 1, Bit 4 or Port 1 Bit 6

Figure 3—*The ability to completely program the clock generator provides the basis for 'x3 low-power modes.*

base (other options are clock/2 and an external input), which the rest of the chip is running at high speed.

Even the lowly SPI port earns Brownie points with lots of programmable options—master or slave, clock source, divide ratio, slave select (SS) polarity, and data transfer (SIN/SOUT) on the clock's rising or falling edge (SK).

MCU + DSP = MSP?

Zilog also offers low-cost tools including an emulator/programmer board with Windows front-end (see Photos 1a and 1b) and a full-featured assembler, while PLC offers a C compiler for those so inclined.

I didn't have a lot of time to check the tools out thoroughly, but they seemed to work fine, notably passing my ancient PC reality check. If the stuff works with my 33-MHz '386 Win3.1 box of distinctly dubious pedigree, I suspect your PC can handle it.

Experimenting with the tools was the final nail in the coffin of my preconceived notions. The look and feel of the setup was no different than for any number of micros I've dealt with.

If the '*x*3 is any indication, what a chip is labeled will soon be more a matter of marketing than what's under the hood. Zilog may call it a "DSP," but it's also a fast, low-cost, high-integration 16-bit controller that just happens to know how to multiply.

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.

CONTACT

Z89*xx*3
Zilog, Inc.
210 E. Hacienda Ave.
Campbell, CA 95008-6600
(408) 370-8000
Fax: (408) 370-8056
www.zilog.com

I R S

431 Very Useful
432 Moderately Useful
433 Not Useful

PRIORITY INTERRUPT

Techno-Jargon

Every industry and occupation has its jargon. At one time, it was just the military that came up with acronyms and abbreviations that stumped the conscious mind. These days, every technical discussion is interspersed with so much techno-speak, you need an acronym dictionary to participate.

There's no rational approach to its creation either. We create new acronyms as if coining one is a necessary marketing tactic. At one time, we just used the acronym with a prefix or suffix to keep things simple. A 1200-bps modem (MOdulator-DEModulator) and a 56.7-kbps modem served the same purpose, but you easily knew the difference. A new Jeep (I forgot what it stood for) isn't called a Jeep XR5. It's an HMMWV (High Mobility Multipurpose Wheeled Vehicle). Of course, the guy who coined the acronym and the people who tried to pronounce it had some disagreement. Now, it's spelled Humvee, but it still has the original description. Talk about muddying the waters.

The worst part about using explicit technical jargon and acronyms in conversation is that unless everyone is clued into exactly the same definitions, everyone can come away with different understandings of what was communicated. An editor attending a news conference had better know that this latest computer widget with ESP has an Enhanced Serial Port—not Extra Sensory Perception.

Keeping up on the latest techno-speak is no easy task. It's a language with no regulations other than use three times as many syllables as would normally suffice, forget all the grammar rules, and try to arrange it into a catchy acronym if possible. The only guide to actual meaning is often the context in which it's used. Even then, you might have to listen very carefully. You can't blindly assume that IDE always means Integrated Device Electronics (hardware), when the reference may in fact be to an Integrated Development Environment (software) instead.

The downside of all this language modification is that we end up with a different language for every expertise. I had a very real demonstration of this during a meeting at a recent computer conference. Typical of tradeshows where time is at a premium, marketing people, engineers, editors, and advertising representatives all meet together. Invariably, as the engineers and editors discuss the latest technical attributes of a new product, you can watch as the other meeting attendees start counting the holes in the acoustic ceiling tiles. As the marketing and advertising people banter in their equally obscure terminology, the editors and engineers fidget with their electronic notepads or just glaze over. As for myself, I nodded confidently at most of the presentation. That was true until their development software description included the sentence, "…and the whole system is cued from the RMB."

Say what? I had familiarized myself with most of the terms in their new product literature just in case. This was one I didn't remember. I could feel a little sweat forming as I tried to inconspicuously open the product brief and scan the block-diagram notations. I couldn't help feeling like a school kid hoping to be invisible to the teacher. All I needed was for the speaker to say, "So Steve, how do you think *INK* readers will feel about our use of the RMB?"

The meeting broke up with everyone feeling that a lot had been accomplished. I frowned as I walked away muttering to myself, "What is this RMB triggering logic?" Later, after a few drinks in the restaurant, I decided to probe the question without directly admitting ignorance. "Should we be investigating the significance of their RMB cueing?" I asked.

One of my editors laughed and said, "I hardly think an exposé on using the Right Mouse Button will excite our readers."

"Right mouse button! Why the hell didn't he just say you start the development program by clicking the right mouse button?"

I know I really shouldn't be criticizing the use of technical acronyms, even techno-speak. After all, when it's used correctly, it's an expedient means for rapidly communicating concepts and ideas. Surely, if everyone at the table understands that BDM means Background Debug Mode, it can more easily be used in the descriptive explanation of another concept. The downside of the constant and arbitrary use of vague acronyms in technical descriptions (especially proprietary ones), however, is that invariably they often serve only to obscure a simpler explanation.

Before you cave in to language intimidation, realize there's no requirement that technical explanations include a dozen six-letter acronyms. Simply presuming that someone who uses a lot of techno-jargon has a better technical understanding is an oversimplification and often incorrect. And unfortunately, even when you know exactly what's being discussed, it can still be very difficult to determine the dividing line between Babel and genius. I certainly discovered that.

Steve

steve.ciarcia@circuitcellar.com