

EMBEDDED PC MONTHLY SECTION

CIRCUIT CELLAR

INK®

THE COMPUTER APPLICATIONS JOURNAL

#93 APRIL 1998

EMBEDDED PROCESSORS

8x51 EPROM/Flash
Programmer

Fast DSP via a
Smart Boot Loader

Xilinx's New FPGAs

Embedding PC Card



\$3.95 U.S.
\$4.95 Canada

TASK MANAGER

Under the Hood



There's nothing like getting grease under your fingernails, oil up to your elbows, and skinned knuckles from a stubborn bolt to appreciate a nice, clean, climate-controlled desk job in front of a computer. I've just spent the past two weeks tearing down the engine in my '85 Dodge Caravan, finding the problem, fixing it, and putting the whole thing back together. All this during the dark evenings in a chilly garage after the kids were in bed.

While taking a break after struggling with the tenth head bolt, I started noting how much simpler engines were 13 years ago than they are today. Sure, the Caravan has an engine computer, but it too was pretty simple back then. As today's cars get more and more sophisticated, it's becoming harder to be a proficient do-it-yourself backyard mechanic. Bring one of today's cars to the shop, and you'll most likely see the mechanic (I'm sorry, "technician") hook it up to the shop's computer and have an intelligent conversation with the car.

"What's ailing you today, Mr. Ford?"

"I seem to have an ache right here under my fuel pump, and the oxygen content of my exhaust is a trifle too high. My owner also isn't paying enough attention to me."

We're also starting to see more smarts elsewhere in the car than under the hood. There's a microcontroller handling the antilock brakes, interior temperature, cruise-control speed, radio/cassette/CD player, and GPS receiver. Never mind the cell phone, alarm system, and collision-avoidance system.

Embedded processors continue their march into just about every piece of electronics we come in contact with daily. In the best implementations, you don't even know there's a computer inside. When the device does what it should naturally rather than try to show off what it could potentially do, then it's a successful design.

This year's Embedded Processors issue starts with an article by Tracey Lee on bootstrapping DSPs, making field modifications much more efficient. Next, G.Y. Xu constructs a universal programmer that can be used to burn code into microcontrollers with either EPROM or flash memory. Vincent Rikkink introduces us to a new low-power micro that shows a lot of promise. Finally, Tom Napier presents his crossbreeding of a Basic Stamp with a Forth engine to come up with a powerhouse.

In our columns, Joe DiBartolomeo continues his series on EMI with a discussion of protection components, Jeff revisits his old buddy the Z8, and Tom goes back even further to his free-wheeling hippie days while talking about Xilinx's latest FPGA.

In *EPC*, Chuck Lewin surveys what's available to aid in PC-based motion control, Ingo illustrates the similarities and difference between two real-time operating systems by implementing the same simple project on both, and Fred uses a PC Card to get a touchscreen up and running.

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 Fax: (860) 871-0411
(860) 872-3064 E-mail: bobbi.yush@circuitcellar.com

ADVERTISING COORDINATOR

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

CONTACTING CIRCUIT CELLAR INK

SUBSCRIPTIONS:

INFORMATION: www.circuitcellar.com or subscribe@circuitcellar.com
TO SUBSCRIBE: (800) 269-6301 or via our editorial offices: (860) 875-2199

GENERAL INFORMATION:

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

AUTHOR CONTACT:

E-MAIL: Author addresses (when available) included at the end of each article.
ARTICLE FILES: ftp.circuitcellar.com

For information on authorized reprints of articles,
contact Jeannette 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.

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

Entire contents copyright © 1998 by Circuit Cellar Incorporated. All rights reserved. Circuit Cellar INK is a registered trademark of Circuit Cellar Inc. Reproduction of this publication in whole or in part without written consent from Circuit Cellar Inc. is prohibited.

- 12** **DSP Boot-Loading Techniques**
Tracey Lee
- 18** **8x51 EPROM/Flash Microcontroller Programmer**
G.Y. Xu
- 22** **CoolRISC**
The Low-Power Microprocessor Solution
Vincent Rikkink
- 30** **Picaro**
A Stamp-like Interpreted Controller
Tom Napier
- 68**  **MicroSeries**
EMI Gone Technical
Part 3: Protection Components
Joe DiBartolomeo
- 74**  **From the Bench**
Rebirth of the Z8
Part 1: An Old Friend Comes to Visit
Jeff Bachiochi
- 82**  **Silicon Update**
VolksArray
Tom Cantrell

| | |
|--|-----------|
| Task Manager Ken Davidson Under the Hood | 2 |
| Reader I/O | 6 |
| New Product News edited by Harv Weiner | 8 |
| Advertiser's Index | 65 |
| Priority Interrupt Steve Ciarcia What You Get with a Handshake | 96 |

INSIDE ISSUE 93

EMBEDDED PC

- 42** **Nouveau PC**
edited by Harv Weiner
- 46** **PCs Move into Motion**
Chuck Lewin
- 53** **RPC Real-Time PC**
Software Development for RTOSs
Ingo Cyliax
- 61** **APC Applied PCs**
Embedding PC Card
Part 2: Getting in Touch
Fred Eady

www.circuitcellar.com

READER I/O

DON'T GET BURNED!

I've been reading *Circuit Cellar INK* off and on for several years now. Steve and Jeff's article, "Ground Zero" (*INK* 90), encouraged me to drop you this note. No matter where I live, it seems, vicious thunderstorms are always around at least some part of the year.

I love to watch lightning. I've experimented with high voltage from various sources including big Van de Graaffs. As a kid, I built a big mailing-tube Tesla coil from junk parts that would create a spark in excess of a foot and wipe out all radio and TV for blocks around! I also did some lightning experiments associated with my radio towers, but I never did get to the heart of the mystery. I did almost fry my butt off, though!

Long ago, I found out, when it comes to protection from lightning, make sure it's grounded! I always feed my antennas or towers in some kind of shunt-feed mode so the structure can go to ground as short as possible with as big a ground as I can get. I usually use 2" diameter thin-walled mast pipe and sometimes industrial braid!

Thanks for reminding us how much fun lightning can be!

D. D. Schendel
Scottsdale, AZ

AYE, AYE

I just finished yet another fine issue of *Circuit Cellar INK*, and as usual, I went from cover to cover nonstop. It's truly refreshing to read a magazine that contains actual substance and doesn't assault the reader with the hype found in most computer publications today.

I particularly enjoy Steve's observations in Priority Interrupt. His concerns (and gripes!) parallel mine more often than not. When he expressed concern over bloated PC code a while back (*INK* 89), it was gratifying to see that someone else had noticed. Why do I need 32+ MB of RAM and hundreds of millions of clock cycles per second just to get average performance? As an engineer, I have a hard time trading simple, elegant engineering for something that has been kludged up with gimmicks just to make it sell.

I also have to agree with Steve that the increasing Wintel presence in the embedded world is cause for concern. Personally, I'll be pretty upset if the day comes when I can't find a good, cheap 'HC11 SBC!

On a final note, "Inside the Box Still Counts" (as Steve wrote in *INK* 1 and reiterated in the tenth-

anniversary issue, *INK* 90) is true in my book, too. Somebody still has to understand the hardware before the software is going to fly.

Keep up the good work!

Paul J. Franklin
pfranklin@worldnet.att.net

A WORTHY REFERENCE

Congratulations on bringing to your readers a very important reminder of the real hazard of lightning and the need for well-designed protection ("Ground Zero," *INK* 90).

I have enjoyed Steve's writing since the early days of *BYTE* and always subscribed to *INK*. *Circuit Cellar INK* enjoys a special place in my library—it's one of only three magazine files that I moved to our new location a few years ago. The rest were hauled away in two five-ton trucks!

Robert Barbour
Eagan, MN

CRACK OPEN THE MARKET

I picked up the December issue of *Circuit Cellar INK* and wanted to comment on Steve's editorial ("The Best Kept Secret," *INK* 89). I've been a programmer for 15 years, and although I've never had the need to delve into embedded systems, I try to keep up with related disciplines.

I think Steve's views regarding the Wintel predominance in the marketplace are timely for the embedded-systems industry and quite correct. He has the luxury of learning from history in this matter. The PC market has been ultimately decimated, in my opinion, by Intel—but more by Microsoft, a company that somehow manages to absorb change but withstand progress. I've been frustrated by their (mostly) inferior and closed architecture for too long.

I'm so glad some alternatives are starting to appear. I manage to make a decent living existing solely in the Linux OS world, and I'm grateful an open, shared system is available for everyone to choose. I'm hopeful, this situation will remain available in the embedded-systems world as well.

Jim B.
jimbag@kw.igs.net

NEW PRODUCT NEWS

Edited by Harv Weiner

BRUSHLESS DC FAN MANAGER

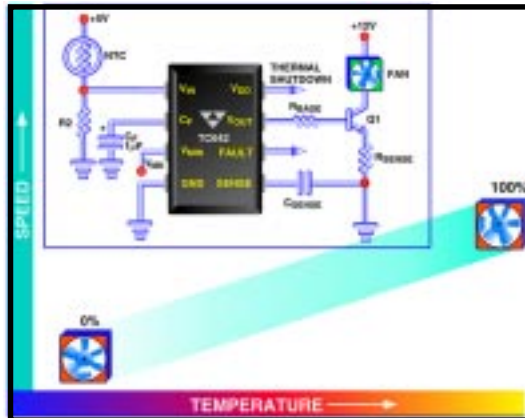
The **TC642 brushless DC fan manager** converts any standard two-wire brushless DC fan into an intelligent fan, controlling and monitoring brushless DC fans of any current or voltage. It consists of a PWM fan speed controller and integrated fan fault detector. Its efficient, low-frequency (30 Hz) PWM fan speed control reduces acoustic noise, extends fan life, and lowers average fan operating current.

Onboard fan fault-detection circuitry continuously monitors fan operation. When an abnormal condition is detected, the *FAULT output is asserted and fan operation terminated. This output is also asserted (but fan operation permitted to continue) when measured temperature exceeds a maximum limit, indicating a blocked air intake or system thermal runaway.

The TC642 lets the user program a minimum operating

fan speed with a simple resistor divider. An external fan shutdown input facilitates "green" system operation. Special onboard start-up circuitry ensures that even the most stubborn fans start and run reliably when exiting auto-shutdown and when power is initially applied.

Available in standard eight-pin plastic DIP and SOIC packages, the TC642 is ideal for personal-computer motherboards, power supplies, and other applications using brushless DC fans for forced-air cooling. Commercial and industrial temperature ranges are available. Pricing starts at **\$1.44** in 10,000-piece quantities.



TelCom Semiconductor, Inc.
1300 Terra Bella Ave.
Mountain View, CA 94043-1836
(650) 968-9241
Fax: (650) 967-1590
www.telcom-semi.com

#501

HIGH-PRECISION I/O CARD

Loughborough Sound Images has released a multi-channel, high-precision analog I/O card—the **PMC/16101**. This device is ideal for acoustic sensing in sonar and seismic equipment, noise and vibration analysis, and test and measurement systems, where an analog stimulus triggers the acquisition of data at high data rates.

The single-size PMC format PMC/16101 card offers 16 analog input channels and one analog output channel. It interfaces to the motherboard via a 32-bit PCI interface and features a sustained data rate of 55 MBps. When combined with a low-cost PCI/C42 motherboard, a complete data-acquisition-system front end can be built into a single PCI slot, enabling cost-effective DSP systems to be constructed around a standard PC chassis.



Both processors on the PCI/C42 motherboard, the memory, and the host and PMC interfaces are connected to the same synchronous bus to guarantee low-latency, high-speed data transfers. A programmable interrupt generator across the PCI interface can also be used to control the data flow.

All I/O channels use crystal delta-sigma converters with 20-bit accuracy. Each channel has an associated 1-KB FIFO for data buffering. Sample rates—up to 50 kHz—can be accommodated with the clock provided by an onboard 12.88-MHz crystal oscillator or from an external source. Four TTL-compatible signal lines, configurable as either input or output, are also provided.

Extensive software support is provided for the PCI/C42 motherboard under Windows 95 (Windows 98 ready) and NT. Add-ons include plug-and-play-compatible device drivers, interface libraries, and DSP utilities for daughterboards like the PMC/16101.

Loughborough Sound Images
Loughborough Park, Ashby Road
Loughborough, Leicestershire
LE11 3NE, U.K.
+44 1509-63-4444 • Fax: +44 1509-63-4450
www.lsi-dsp.com

#502

NEW PRODUCT NEWS

SOLAR POWER SYSTEM

The first portable solar power system capable of supplying full operating power to a laptop computer while simultaneously recharging its battery has been introduced by SunWize. This new **Portable Energy System**, featuring a high-efficiency solar panel just larger than a legal pad, can also operate or recharge other low-power products, such as a cell phone, two-way radio, portable stereo, or similar device.

In its basic configuration, the system consists of a solar panel, voltage controller, interconnect cables, and storage case. When used with one panel, the system delivers enough sustained electrical power to the computer (up to 9 W) to double or triple the run time of a portable computer operating with battery power, or it can recharge the battery when the computer is turned off. With a second panel, this power system generates sufficient sustained power to run the computer and simultaneously charge its battery.

The voltage controller supplied with the system can receive power from one or two solar panels. Two output ports on the controller send power either to the portable computer or to a second device. The output for the second

device can be set via a selector switch at 3, 6, 7.5, 9, or 12 V, as appropriate for most portable electronic products.

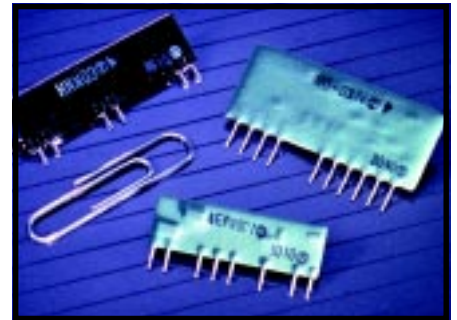
The solar panel is constructed using a proprietary process in which highest-efficiency single-crystal photovoltaic cells are permanently encased in rugged weather-resistant urethane plastic. Built into the case is a patented LCD meter—the Opti-Meter—which instantly measures sunlight intensity and enables optimum placement of the panel. The panel's 10' cord winds on a spool recessed into the back of the panel. A hinged metal stand folds flush into the panel's back side.

A diagram showing how to connect the SunWize Portable Energy System to a computer or other device is permanently printed on the back of the panel. The system weighs just over 2 lbs. and measures 15.5" × 10.5" × 0.675". The voltage controller weighs 4 oz. and measures 4" × 2.5" × 0.675".

The basic SunWize Portable Energy System package, which includes one solar panel, the voltage controller, interconnect cables, storage case, operating instructions, and other material, retails for **\$349.95**. The **System Doubler**, which consists of a solar panel with power cord and storage case, has a retail price of **\$279.95**.

SunWize Technologies, Inc.
1151 Flatbush Rd.
Kingston, NY 12401
(914) 336-7700
Fax: (914) 336-7172
www.sunwize.com

#503



SUPER-COMPACT DC-DC CONVERTERS

Xentek Power Systems offers ultra-compact, high-reliability DC-DC converters for use in LCD bias supplies. Three new models in the **RD Series** of 5-V input variable (positive or negative) bias modules offer wide-range DC outputs of +35 to +40 V, +24 to +40 V, and -30 to +38 V, respectively. A fourth model in the **EPN Series** provides a variable negative bias module with a wide range output of -13 to +24.5 V.

Other models are offered with nominal 5 VDC (4.5–5.5 V) or wide-range (10–25 V) DC inputs. DC output can be specified to be -22, -23, or -24 V. A +38-VDC version featuring adjustable output is also available. Models are available with remote operation (active high or active low).

The largest models in the line measure no greater than 35 mm (1.38") in length and 16 mm (0.63") in width, and all units have a rated MTBF of over 1,000,000 h. Prices range from **\$5 to \$7** per unit in OEM quantities.

Xentek Power Systems, Inc.
1770 La Costa Meadows Dr.
San Marcos, CA 92069
(760) 471-4001
Fax: (760) 471-4021
www.xentek.com

#504

NEW PRODUCT NEWS

SMART BATTERY MONITOR

The **DS2437** smart battery monitor offers a complete battery data-acquisition system on a single chip. It supplies all the real-time battery data needed to ensure that the battery does not overcharge or overdischarge. The chip fits into the battery pack, so it doesn't consume extra space. Applications include cellular phones, PCs, PDAs, portable fax/modems, handheld meters and instrumentation, palmtop computers, and more.

The DS2437 measures four key parameters—time, temperature, voltage, and current—to keep the battery pack operating within safe limits while charging and discharging. A 64-bit ROM code laser-engraved on each chip provides a unique ID to prevent the use of cloned batteries. As well, 40 bytes of EEPROM enable the DS2437 to function as a battery-pack ID device by storing a manufacturer's ID, battery-chemistry code, and charge and discharge param-

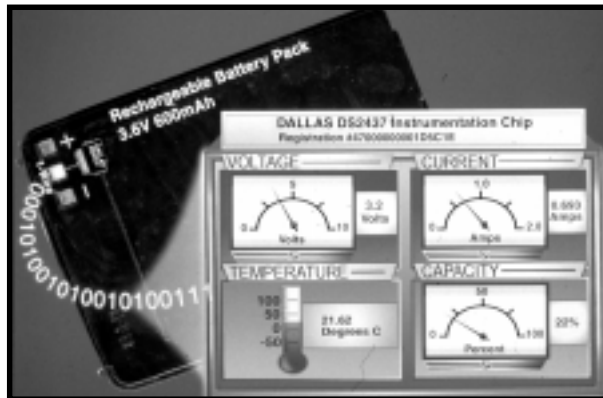
eters. This memory also permits optimization of the battery system for multiple chemistries, suppliers, capacities, and usage histories. The DS2437 is compatible with, but not limited to, these battery chemistries: NiCd, NiMH, SLA (Sealed Lead Acid), RAM (Rechargeable Alkaline Manganese), Li-Ion, and Li-Polymer.

Communication between the central microprocessor and the DS2437 is achieved via a one-wire interface, which simplifies interconnection and preserves system resources. The battery pack uses three output connectors—battery power, ground, and the one-wire interface.

The DS2437K sells for **\$4.63** in 5k quantities.

Dallas Semiconductor
4401 S. Beltwood Pkwy.
Dallas, TX 75244-3292
(972) 371-4448
Fax: (972) 371-3715
www.dalsemi.com

#506



FEATURES

12 DSP Boot-Loading Techniques

18 8x51 EPROM/Flash Microcontroller Programmer

22 CoolRISC

30 Picaro

FEATURE ARTICLE

Tracey Lee

DSP Boot-Loading Techniques

Memory speed hasn't kept up with the speed increases of micros, especially DSPs, where EPROM acts as a bottleneck. Rather than slowing the DSP down, Tracey uses a PIC to load SRAM with external code and then wake the DSP up.



In today's microcontroller environment, memory speeds haven't kept up with the speed increases of the processing unit. Even the high-speed version of the 8051—namely, Dallas Semiconductor's 87C530—requires a chip with an access time of 55 ns or better to run at its maximum rated clock speed.

But, at least this chip has built-in wait states. Otherwise, you'd have to use it at a lower crystal frequency. Fast nonvolatile memory is expensive.

In this article, I examine how I solved this problem for a particular microprocessor. It should be fairly clear, however, that these techniques can be applied to a wider class of processors. I first discuss the standard techniques available and then present my solution.

I was working with a group of students to develop the Sound FX-26 board—an entry in the Texas Instruments DSP Solutions Challenge 1995

| | | |
|-------------------------|-----|------------|
| Built-in boot facility | C25 | C26 |
| RS-232 | No | Yes |
| Parallel I/O | No | Yes |
| EPROM width | No | 8 bit |
| Length of data transfer | N/A | 1536 words |

Table 1—C2x devices have basic boot-loading facilities that enable you to transfer code to fast SRAM.

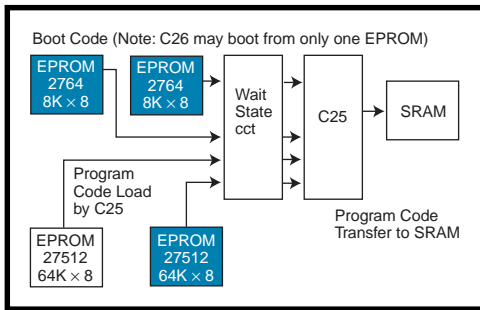


Figure 1—In a conventional design, here's how you would boot load the 'C25 from a 2 × 8 device.

in Singapore. It needed to be a general-purpose board capable of producing a wide range of sound effects, and the hardware had to be able to handle newer and more demanding algorithms.

BOARD DESIGN

In the design of the sound board, I used a 40-MHz Texas Instruments TMS320C25 DSP chip. In a more conventional design, the DSP chip would be wired up like the one illustrated in Figure 1.

Whereas most designers of conventional microcontrollers will just hang an EPROM off the address and data buses without a care, you have to note here that these TI fixed-point devices have several features that need to be considered.

First of all, the 'C2x features a 16-bit data bus. And, it operates at a clock speed of 40 MHz.

Additionally, this device has a Harvard architecture. In other words, programs execute out of a separate memory area (i.e., the code space), while data is accessed from data space. However, both spaces share a common address and data bus, whereas the CPU uses separate control pins to

indicate when a code or data fetch is taking place.

So, I had two choices. Either I'd need two eight-bit nonvolatile devices fast enough to keep up with the processor, or I'd have to introduce wait states. Herein lies the main challenge of implementing such a system. This basic concept involves exploiting the high performance of the Harvard architecture found in the DSP chip.

Chips with a Harvard architecture have problems in the code space. At 40 MHz, the TI manuals recommend that, for nonwait-state access, memory devices should have a 35-ns access time or better.

Since code space needs to be nonvolatile, devices that are high speed yet nonvolatile (like the 27C292, which has a 35-ns access time) tend to be costly and have a small capacity. Although larger capacity nonvolatile devices give better performance—some flash devices can provide 70-ns access time—they still fall short of the required access time.

In the search for high-capacity memories and fast access time, you can't beat SRAMs. Typical SRAMs come with capacities of 32 and 64 KB. Such SRAMs have declined in cost over the years, which may be attributed to the high demand for using them as memory caches in PCs.

A 64-/256-KB cache comes standard on today's PCs. So, it's possible to capitalize on these 20–25-ns access-time devices for our DSP chip.

In a typical setup, the 'C2x devices can address up to 64K words of code memory, which translates to 128 Kb,

enabling the use of 1-Mb devices like the 27C010 (EPROM) or 28010 (flash memory). The program stored within may be transferred to fast SRAM and the processor can then execute the program from there.

Another 64K words may be addressed as data memory. These use SRAM anyway.

There were two main options I had to consider. Let me go over each of these in turn.

First of all, I could use a standard boot. That is, I'd use standard EPROMs for code with wait states on the DSP chip. This solution has the lowest cost, and it uses conventional SRAM for data.

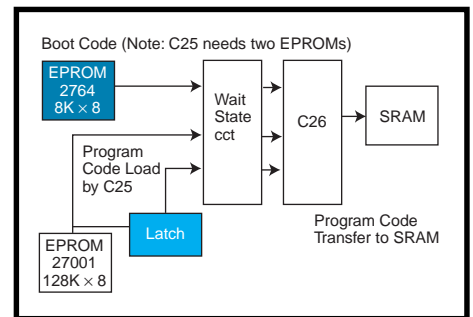


Figure 2—You can also perform a boot-load process using a transfer from 8-bit memory.

Then, all code accesses will be slower because of wait states. However, certain portions of code may be loaded into faster internal code RAM, which is a feature of these chips. The size is 1536 and 4096 bytes for the 'C25 and 'C26, respectively.

Another possibility is booting from SRAM. In achieving high performance, the 'C2x devices have basic boot-loading facilities that allow code to be transferred to fast SRAM. The boot-load facilities of the 'C26 are summarized in Table 1.

The CPU's firmware transfers some or all of the code from other devices to internal or external RAM. Therefore, we may use standard, small EPROMs with wait states, which the DSP can boot from.

This boot code then loads the actual program code from various sources, like a larger EPROM, serial device, or parallel device. After that, slow devices attached to the data bus have to be switched out of the address space if the memory location is to be used.

| Feature/Requirement | 'C25 | 'C26 |
|---|--|---|
| Boot code loader | 2 × 8 bit EPROM | 1 × 8 bit EPROM |
| Code storage | 128K × 8 bit (all memory usable) | 128K × 8 bit (boot code uses some space) |
| Boot config register decode | N/A | 1 PAL |
| Wait state and decode | 1 PAL | 1 PAL |
| Switch between code storage device (EPROM) and code memory (SRAM), retrieve 16-bit data from 8-bit device | 1 PAL, I/O address used or one I/O pin | 1 PAL, I/O address used or one I/O pin |
| Code address space | Some used by boot load | Some used by boot load |

Table 2—To implement the boot-to-SRAM feature, you need these features. Note the differences between what the two chips need.

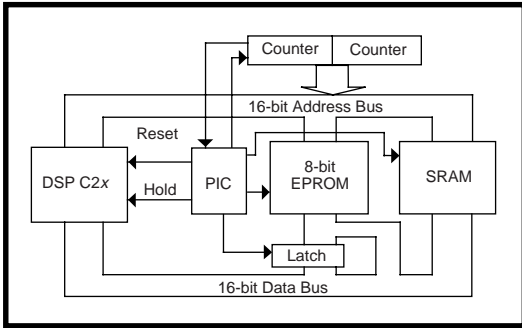


Figure 3—The PIC controller can boot load to SRAM using only 30 lines of code.

This option offers the most flexibility, but it takes up more hardware. In the conventional design of such a project, the lowest-cost solution is to boot from an eight-bit EPROM and transfer program code from the device.

To implement this, another set of hardware is needed, and other resources are taken up as well. Between the two, the 'C26 chip has better built-in booting facilities, such as being able to boot from a specially configured 8-bit EPROM or serial device.

Alternatively, you can load from a single eight-byte device. But, the 'C26 may need some hardware assist to read in two bytes and transfer it over. For example, a latch may be used as shown in Figure 2.

In a stand-alone embedded system, however, you may not be able to boot from a serial device. So, the solution providing both the lowest cost and highest performance is to boot from an eight-bit EPROM and transfer program code from such a device.

Extra resources are needed to implement the boot since extra decoding hardware must switch out the EPROM after booting and account for it in memory decoding. As well, the hardware needs to cater to the slower access time of the EPROM boot-load devices by adding wait states.

Extra software has to be written to account for the boot loading. As well, some addresses in the code memory are used by boot-loading software.

To switch out the EPROM, we need to use up some other resource, perhaps a device pin or an address in the 'C2x I/O address space. Table 2 summarizes the hardware and software requirements to implement the boot to SRAM option.

The PALs would be 20-pin devices, or a GAL could also be substituted. In other words, the additional hardware needed to implement native EPROM boot loading for a 'C25 are the two PALs (2×8 bit) and a $128K \times 8$ -bit EPROM. For a 'C26, I need three PALs and a $128K \times 8$ -bit EPROM.

Special boot code needs to be written, but that's a one-time job. Also, certain address spaces may be reserved. Although this overhead is small, it may be significant in certain cases. Be sure to watch out for this when porting code.

THE IMPLEMENTATION

After considering all these options, it was clear that there had to be a better way, especially in the use of code. What about leaving gaps in the code space to cater to the boot process?

In this implementation, I explored the possibilities of transferring data from the EPROM to fast SRAMs in a DMA-like operation. The general idea is that instead of letting the DSP do the code transfer, you switch it out of the circuit, let an auxiliary circuit do the transfer of data, then switch in the DSP.

If you consider how some dedicated DMA controllers (e.g., the Intel 8257) work, this rules out the use of PALs. The logic would be too complex.

Note that for this option to work, the address, data, and control buses must be tristate capable. Unfortunately, this rules out quite a few processors.

To transfer code from EPROM to SRAM, I next considered a microcontroller. Some may complain and say that to delegate another controller to such a lowly task is surely a waste of resources. But, I'll show you later that this isn't the case.

At this point in time, some widely used low-pin-count processors cost about the same as three GALs. These low-cost processors may be one-time programmable in the field and are available from chip manufacturers like Microchip, Zilog, and Motorola.

For this project, I decided to go with the PIC16C54 from Microchip.

This microcontroller had a windowed EPROM version and was easily available.

In the final design, no PALs are needed for wait-state generation or special decoding. There are no special addressing requirements as well. The boot-loading circuit effectively switches itself out of the DSP circuit when its task is completed and takes over the system again only when the reset switch is pressed.

So far in this system, the additional chips needed for boot loading are now two counters, one EPROM, one latch, and one microcontroller. Note that code is kept in 8-bit EPROM while the 'C2x reads code 16 bits at a time.

I used 8-bit SRAM devices connected up as two 8-bit devices. Code needs to be transferred from two addresses of the EPROM into one address of the SRAM.

MICROCONTROLLER FUNCTIONS

The microcontroller controls the RESET and HOLD pins on the DSP chip. These pins make sure that the DSP chip is inactive and its pins tristated, so that data may be transferred between EPROM and SRAM.

This microcontroller also clears and increments the counters for generating the 16-bit addresses used to access

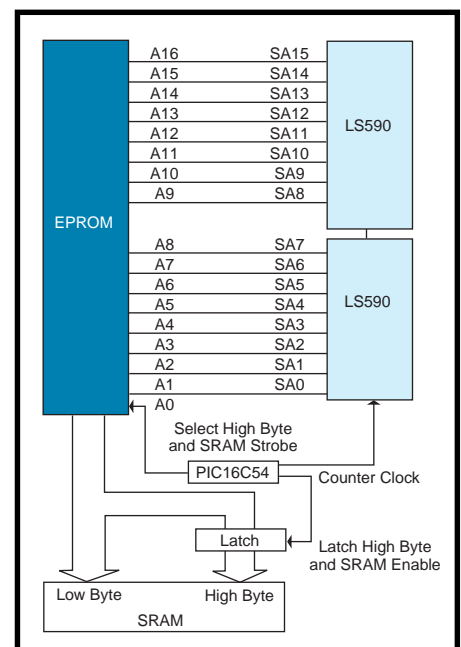


Figure 4—Here's how I transfer data from an 8-bit EPROM to a 16-bit SRAM system.

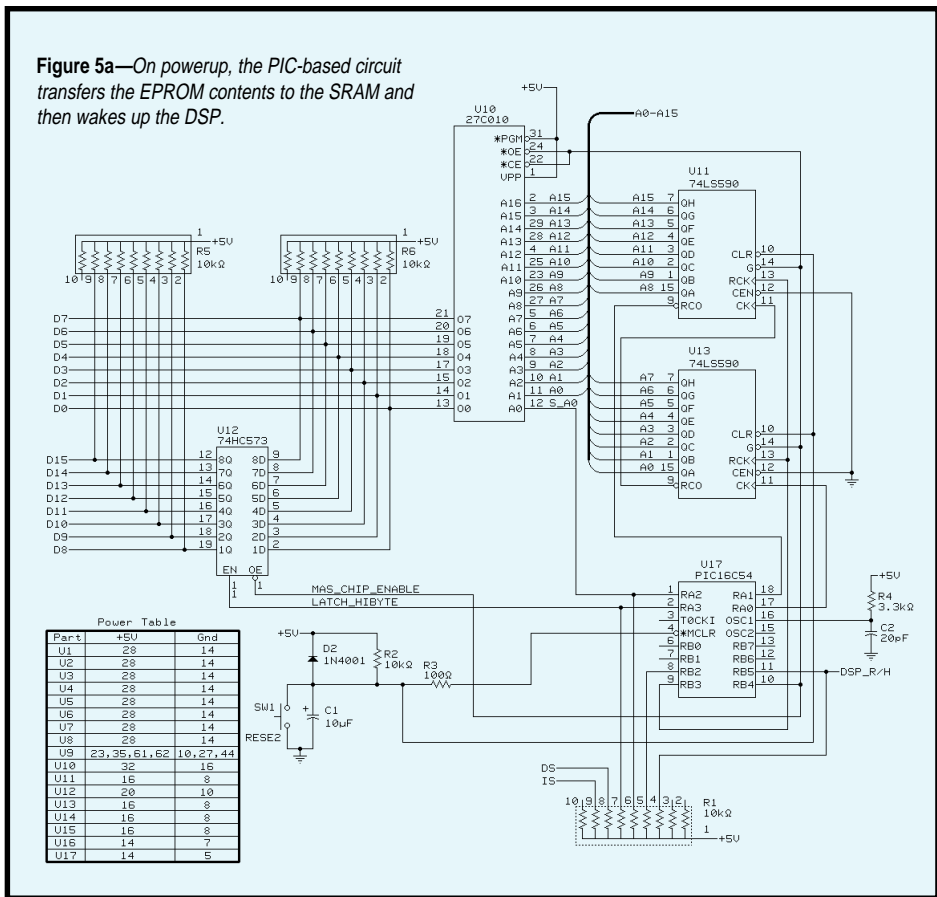
both the EPROM and SRAM. And, it generates the Enable and Read signals to the EPROM.

Another function of the microcontroller is to latch the higher eight bits from the EPROM and generate the address of the lower byte held by the EPROM, thus forming a 16-bit word which is held in preparation for the next step. The address remains the same for both SRAM and EPROM, except the lowest address bit is not used by the SRAM during the code-transfer process.

In addition, it outputs the Write signal to the SRAM chip and increments the counter and checks for overflow from it. When this task is done, the RESET and HOLD pins are released and the processor runs.

Figure 3 shows a basic block diagram of the system. The PIC does all this with only 30 lines of code, excluding comments. At this time, the OTP version of the PIC costs about the same as three GALs. The counters are tristate capable, as are the PIC and the latch.

Figure 5a—On powerup, the PIC-based circuit transfers the EPROM contents to the SRAM and then wakes up the DSP.



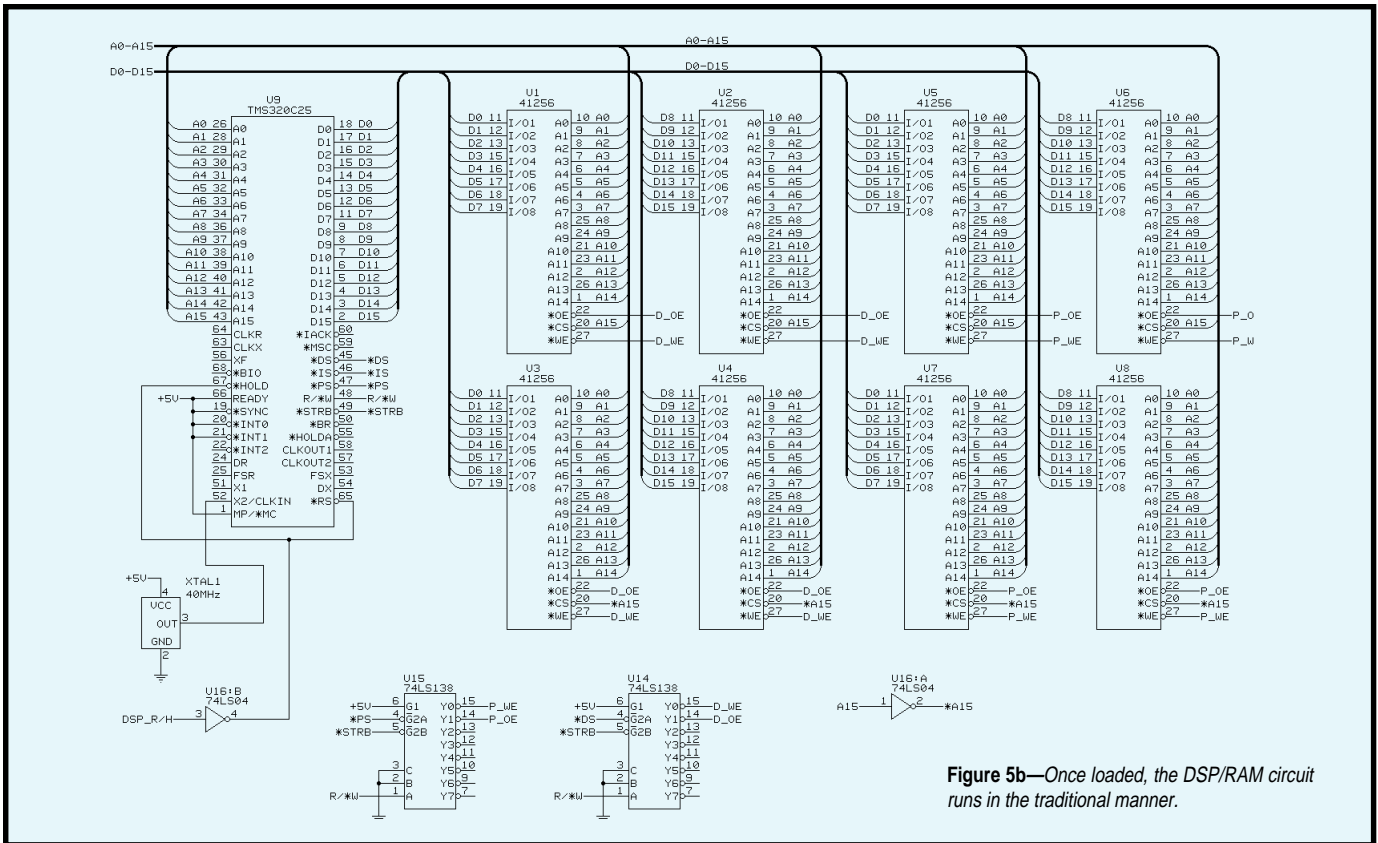


Figure 5b—Once loaded, the DSP/RAM circuit runs in the traditional manner.

DATA TRANSFER

Let's look next at how I transferred data from an 8-bit EPROM to a 16-bit SRAM system. Figure 4 illustrates this process. (Several other pins aren't described here.)

Also, the addresses of the EPROM and the tristatable counters are offset by one. The PIC latches the high byte and leaves the low byte to be output by the EPROM.

Because of a shortage of pins on the PIC, many of them had to do double duty. For example, the Latch High Byte signal is used elsewhere (see Figure 5) as an Enable signal to the SRAM. Likewise, the Select High Byte signal is used as a Strobe signal to the SRAM.

The processor used is the TMS-320C26 with the full 64K words of data and code space available for use. No wait states are employed.

The schematic in Figure 5 doesn't show the entire Sound FX-26 board—just the processor and memory portion. As the schematic has been modified for this article, the modified circuit has not been tested. However, I routed the Sound FX-26 on a two-layer board (7" × 4"), and it performed to specifications.

SRAM'S YOUR PAL

In summary, I've shown you how I used SRAMs to improve and exploit the performance of fast DSP chips or similar available devices. The cost of inexpensive voltage controllers and parts like counters and latches is less than the cost of using PALs.

How cost-effective this solution is depends on the type and cost of the microcontroller you use. Special considerations for gaps in the address space which must be set aside to support the boot-loading hardware are no longer required. This eases implementation complexity and enhances code portability.

I hope you now feel confident enough to try constructing a DSP or similar system at a reasonable cost and with high performance. Have fun. 📧

I want to acknowledge the contributions of Koh Beng Chuan and Poh Tze Koon in setting up the Sound FX-26 and Ong Kok Leong for the initial draft of this article.

Tracey Lee lectures at Singapore Polytechnic, where he specializes in micro-systems. He has a B.Eng. from the University of Singapore and a M.Eng.

from the Nanyang Technological University. You may contact him at tlee@sp.ac.sg.

SOFTWARE

Source code for this article is available via the Circuit Cellar Web site.

SOURCES

TMS320C25, TMS320C26

Texas Instruments, Inc.
MS 14-01
34 Forest St.
Attleboro, MA 02703
(508) 699-5269
Fax: (508) 699-5200
www.ti.com

PIC16C54

Microchip Technology, Inc.
2355 W. Chandler Blvd.
Chandler, AZ 85224-6199
(602) 786-7200
Fax: (602) 786-7277
www.microchip.com

I R S

401 Very Useful
402 Moderately Useful
403 Not Useful

FEATURE ARTICLE

G.Y. Xu

8x51 EPROM/Flash Microcontroller Programmer

With all the 8051s out there, it's nice to have one programmer that fits all. That's what Xu shows us—a universal 8051 programmer that can blank check, read, write, and verify EPROM- and flash-based MCUs.



The 8051 is one of the most popular 8-bit microcontrollers used in industrial control and the electronic world.

Since Intel first introduced it in the early '80s, numerous versions of the device (generally referred to as 8x51) have been manufactured by many semiconductor companies, including AMD, Atmel, and Philips, to name just a few. And, the market demand on these devices continues to grow.

A whole line of 8051 derivatives features memory built-in with programmable code. The 8751, for example, is an 8051 with 4 KB of EPROM code memory. The 89C51, by contrast, is an 8051 with 4 KB of flash memory, and the 89C55 contains 20 KB of flash memory.

These devices provide much more flexibility and versatility than the 8051. The EPROM version must be erased by an ultraviolet-light eraser before it can be reprogrammed, but the flash version doesn't require a special tool. It can be erased electrically in-circuit or by a programmer.

The programmer presented in this article is designed to program three kinds of DIP-style 8x51 devices. The first device is the venerable EPROM version 8751H, which requires a 21-V programming voltage and the normal programming algorithm (50-ms programming pulse for each byte).

The CMOS EPROM version 87C51, which uses a 12.75-V programming voltage and the Quick Pulse programming algorithm (1-ms programming pulse for each byte) is the second, and the third is the Atmel 89C51/55 flash MCUs, which use a 12-V programming voltage and 2-ms write cycle for each byte.

This programmer software is a menu-driven system. It contains a submenu for each kind of device, and it can blank check, write, read, and verify the device to be programmed.

The menu also provides erase capability for the flash MCUs. Additionally, it can program the lock bits of the device to protect the programmed source code. The programmer accepts both Intel hex and binary format files.

To keep the cost of the programmer as low as possible, I designed it under the constraint of using just two chips. This approach means that the device can program the AT89C55's flash memory only up to 16 KB, not the full

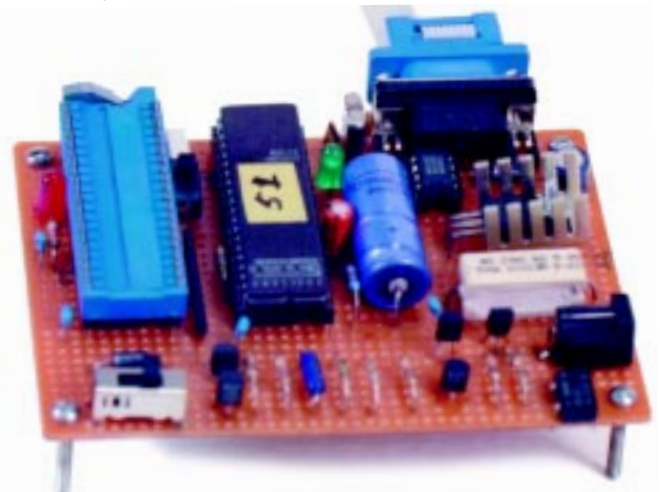


Photo 1—This 8x51 MCU programmer features minimum chip count design and is easy to use. It programs the popular 8751H and 87C51 and Atmel's AT89C51 and '55 (up to 16 KB). With an adapter, it also programs Atmel's 20-pin AT89C2051 and '1051.

20 KB. With an adapter unit plugged into the programmer's ZIF socket and some special software, it can also program Atmel's 10-pin 89C1051 and '2051 flash MCUs, which contain 1 and 2 KB of flash memory, respectively.

CIRCUITRY AND OPERATION

The complete schematic of the 8x51 programmer is shown in Figure 1. This device connects to the serial port COM1 of an IBM PC or a PC-compatible computer, which serves as the host.

As I mentioned, the programmer circuitry was designed with an emphasis on minimum chip count and low cost. As a result, it was implemented with only two chips—Dallas Semiconductor's DS1275 line-powered RS-232 transceiver chip (U1), which transfers the TTL signal level to RS-232 level and vice versa, and the 87C51/89C51 microcontroller chip (U2), which stores the system control firmware.

Photo 1 gives you a view of the programmer board. Its power-supply circuit consists of the bridge rectifier (BR1), which receives 24 VAC from a wall-mount transformer's output and transfers it to about 33 VDC with the filter capacitors C1 and C2.

There are two voltage regulators as well. VR1 accepts the dropped voltage from the power resistor R1 and provides +5-V output for V_{CC} , and VR2 provides multiple controlled regulated outputs for the target MCU's programming voltage, V_{pp} .

The transistor pair Q1 and Q2 form a solid-state switch to control the on and off states of V_{pp} . The base potential of Q2 is controlled by the system MCU's port pin P3.2. When Q2 base is applied to a high-level signal from the MCU, Q2 and Q1 turn on simultaneously.

Additionally, VR2 outputs a voltage V_{pp} that depends on the values of R5, R6, R7, R8, and the on and off states of transistors Q3 and Q4, which are controlled by the MCU's port pins P1.7 and P1.6, respectively.

The system MCU runs at a clock frequency of 3.6864 MHz, as determined by the crystal (XTAL) and capacitors C6 and C7. Capacitor C5 is used to reset the system to a known state during powerup.

RX and TX are the two pins used by the system to receive and transmit signals from and to the host PC. The communications software runs at 9600 bps. A reverse-biased Schottky

diode (D1) protects the line-powered DS1275 chip.

The 40-pin ZIF socket serves to hold the target 8x51 MCU. To program the target MCU, three groups of signals are derived from the system MCU. D0–D7 are data lines that use the entire port P0. As well, address lines A0–A13 use the entire port P2 and 6 bits of port P1.

Also, five control signal lines come from system-MCU pins P3.3, P3.4, P3.5, P3.6, and P3.7. The first pin (P3.3) provides the programming pulse signal to the target (*PROG), and the next four pins provide the control signals required for operations like device read, write, flash erase, and program the lock bits.

Under such a configuration, the circuit is capable of handling all the 4-KB EPROM and flash-memory 8x51 MCUs, with one exception. It doesn't handle Atmel's new 89C55. However, Atmel has announced that it will continue to develop more flash-memory MCUs with higher capacities.

You'll recall that the remaining two bits of port P1 (P1.7 and P1.6) and the remaining bit of port P3 (P3.2) are already used for V_{pp} controls. In other words, I'm running out of all the

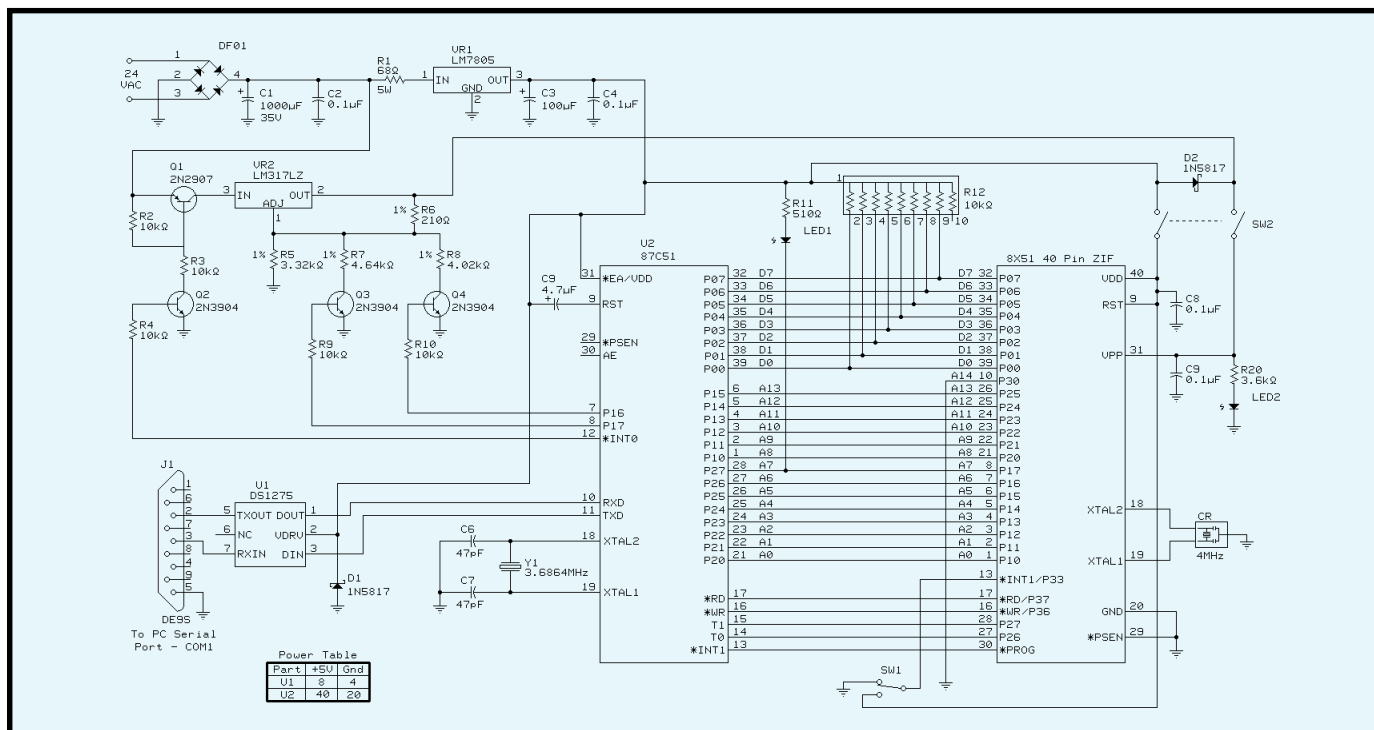


Figure 1—As you can see from this schematic of the 8x51 MCU programmer, the DS1275 provides an RS-232 serial interface to the host PC. The 87C51 or 89C51 controls the programming tasks and communications with the host. The respective power supplies V_{CC} and V_{pp} are generated by two voltage regulators.

available pin resources provided by the 87C51/89C51.

Because I wanted to keep the programmer's cost low, I didn't want to add chips just to achieve complete handling of the 89C55. And, I didn't have to—there are other options. The result is the compromised circuit shown in Figure 1.

Here, 14 address lines (A0–A13) are constructed from the system MCU, which can handle 16 KB of memory addressing. I grounded address pin A15 for the 89C55 (or other high-capacity target MCU) due to my limited resources.

Another issue that needs to be addressed here concerns the two switches SW1 and SW2. In the winter of 1995, Intel announced a new requirement on its 87C51 programming algorithm, which adds pin P3.3 as a control pin. For its 87C51 EPROM writing, this pin must be held high. For reading and verifying, it must be held low.

To comply with such a requirement, an SPDT switch (SW1) should be connected to the target MCU's pin P3.3. The user can manually set it to a high or low logic level when programming the Intel 87C51. But for other companies' 8x51 chips, this pin is a don't care, so you can leave the SW1 in any state.

The DPST switch SW2 helps you avoid the ZIF being a hot socket. This switch should be open (i.e., turned off) before the target MCU is put on the ZIF socket. This action means that the ZIF socket is isolated from both V_{CC} and V_{PP} when you're placing the target MCU.

The Schottky diode D2 serves two purposes. First, it provides a logic high level to the target MCU's V_{PP} /EA pin from the V_{CC} side while reading. It also blocks the V_{PP} to prevent it from entering the V_{CC} side during programming.

The 4-MHz ceramic resonator (CR) enables the target MCU circuit to work. During reading or writing on this MCU, it needs a clock signal for data transfer. When reading or verifying, the data buses D0–D7 transmit data from the target MCU to the system MCU. Such a situation requires 10-k Ω pull-up resistors.

LED1 is the system's indicator light. When the system is powered up, this LED goes on. It's interesting to note that it connects to the address pin A7 in this resource-tight system. Fortunately, that doesn't hurt the system performance at all because, when reading or writing, A7's logical level can also indicate that the system is operating properly.

For example, every time the LED1 changes its on and off state during reading or writing, the user is informed that another 256 bytes of data have been transferred. LED2 is an indicator light for the V_{PP} during MCU programming.

A double-sided, plated-through-hole PCB is needed for building this programmer. You can make your own PCB by taking a look at the information given at the end of this article.

USING THE PROGRAMMER

There are two programs to make the 8x51 programmer work with a PC—MP1.EXE, which resides in the PC's memory, and MP51.BIN, which resides in the 87C51 EPROM or the 89C51 flash memory of the programmer.

With the 8x51 programmer's power off, place the 8x51 MCU device to be studied on the ZIF socket. To avoid any hot sockets, you should keep the switch SW1 up and switch SW2 left at the beginning.

Connect the DB-25 male/female cable between the programmer and your PC's serial port COM1, and then turn on the plug-in 24-VAC power supply. You'll see LED1 light, but LED2 remains off.

A sample program file, LEDSHOW.BIN, is supplied with the source code that comes with the programmer. This simple program causes the two LEDs mounted on the programmer to light up and blink. When you first run the 8x51 programmer, it's best to give this program a try to see if the system works as expected.

To start using the 8x51 programmer, from your hard disk or any floppy drive containing the supplied programs, type MP1 and press Return. A root menu appears on the screen, prompting you to select the device you're going to work with.

Suppose you've placed an AT89C51 on the ZIF socket, you should select the option corresponding to AT89C51. The AT89C51 programming menu is then displayed onscreen.

There are several options to choose from—blank check, write, read, verify, erase, or program the lock bits for the AT89C51. You can try them one by one as you like.

Now suppose you selected the blank check and already got the "Blank Check Okay" response. Next, select the write option. The program will ask you about the filename and extension you're going to write. Type in LEDSHOW.BIN.

In less than a second, the "MCU Programming Finished" message appears, indicating that the programmer works fine. To verify that the code stored in the target MCU memory matches the code stored in LEDSHOW.BIN, you can choose the verify option.

To demonstrate your success, turn the 8x51 programmer's power off and use a chip puller or screwdriver to temporarily remove the system firmware MCU (U2) from its socket and replace it with the newly programmed MCU. Then, power your system up again.

If you now see the LED1 single blinking and LED2 double blinking, then you've been successful in programming the MCU. That's all the LEDSHOW program is supposed to do.

You don't have to build your own circuit. You can just use the programmer itself to demonstrate your result. That's a unique advantage of the 8x51 programmer.

GIVE IT A TRY

With so many different kinds of 8x51 devices available today, you can use this programmer to do a lot of interesting jobs.

For example, as an experiment (and only an experiment!), you can place the 8052AH/8052-BASIC chip on the programmer's ZIF socket and read its 8 KB of code memory into a disk file called BASIC_52.BIN.

Then, put an Atmel's 89C55 chip on the same socket and use the programmer's write option to transfer this file into the 89C55's flash memory.

From now on, the programmed 89C55 becomes a chip just like any other 8052AH-BASIC chip, and you can use this BASIC-52 chip directly or modify it to replace the original BASIC-52 chip.

Good luck getting your programmer to work just the way you need it. ☐

G.Y. Xu has many years of experience specializing in microprocessor and microcontroller systems design and development, both in hardware and software. He may be reached at (713) 741-3125.

SOURCES

Assembled and tested 8x51 programmer \$75
Complete kit (software, PCB, hardware, cable and 24-VAC plug-in transformer) \$65
PCB and software only \$29.95
Adapter for AT89C1051/2051 . \$30
Blank 8751H (4-KB EPROM) ... \$15
AT89C51(4-KB flash memory) \$15

G.Y. Xu
P.O. Box 14681
Houston, TX 77021
(713) 741-3125

89C1051/2051 flash MCUs, 87C51/89C51

Atmel Corp.
2125 O'Nel Dr.
San Jose, CA 95131
(408) 441-0311
Fax: (408) 436-4200
www.atmel.com

DS1275

Dallas Semiconductor
4401 S. Beltwood Pkwy.
Dallas, TX 75244-3292
(972) 371-4448
Fax: (972) 371-3715
www.dalsemi.com

80C52

Micromint, Inc.
4 Park St.
Vernon, CT 06066
(860) 871-6170
Fax: (860) 872-2204
www.micromint.com

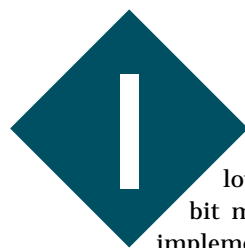
I R S

404 Very Useful
405 Moderately Useful
406 Not Useful

Vincent Rikkink

The Low-Power Microprocessor Solution

Standard 8-bit micros—the 68HCxx, PIC16/17xxx, and 8051/8031—offer low power consumption, but at a price: they need several clocks to execute one instruction. CoolRISC manages to offer both low power and fast execution.



Low-voltage and low-power eight-bit microcontrollers implemented as standard chips or embedded cores are becoming increasingly needed for portable products. The market today is dominated by conventional architectures like the 68HCxx, PIC16xxx and '17xxx, and 8051/8031.

The power consumption of these systems has been reduced via advanced low-voltage 0.8- and 0.6- μ m technologies. However, these architectures still need several clocks to execute a single instruction, resulting in a clock per instruction (CPI) of 4–20.

One measurement that needs to be taken into account when choosing an architecture is MIPS, which equals frequency divided by CPI. To provide good MIPS performances, these systems have to be clocked at relatively high frequencies.

But because power consumption is proportional to frequency, these systems present a MIPS-per-watt figure that isn't reduced to the extent it could be. The goal for the CoolRISC 816 microcontroller, then, was to achieve 10 MIPS and 2000 MIPS/W at 3.0 V.

Due to the RISC revolution, 32- or 64-bit pipelined microprocessors present CPIs between 1 and 2 for scalar archi-

tectures and less than 1 for super-scalar architectures. But, these architectures haven't been used for 8-bit microcontrollers due to software incompatibility.

RISC architectures also present drawbacks for 8-bit low-cost devices. The code size grows as 32-bit instructions increase the size of the program memory.

Additionally, the load and store mechanism requires more instructions to execute operations with the data memory. And, the pipeline, which includes prediction logic as well as the register set, results in more hardware.

Development tools—compilers, mainly—provide code with a minimum of executed instructions for a given task. Although the compiled code size is always bigger than handcrafted code, the number of instructions in the program memory produced by the compiler must be kept as low as possible. Code optimization is also achieved by reducing the number of memory operands via an optimal allocation of variables to internal registers.

Subroutine calls must be handled in priority by the hardware stack before the software call (branch and link) is used. If the compiler has the choice between various instructions, it has to pick the one with the less energy per instruction.

If a short subroutine isn't frequently called, its code can be inserted two or three times when it is called. So, the CALL and RETURN instructions can be removed.

As well, small loops executed only a few times can be unrolled. The compiler removes instructions that increment and test the loop counter, resulting in more instructions in the memory but fewer executed instructions.

These examples illustrate a general rule: less sequencing in the software comes at the expense of more hardware. In other words, fewer executed instructions means more instructions in the program memory. For this reason, tools estimating software power are based on the energy-per-instruction criterion.

ARCHITECTURAL CHOICES

The CoolRISC features a Harvard RISC-like architecture with one-word

instructions (16–22 bit) and separate program and data memories. It also has a register-memory architecture (i.e., one operand of an arithmetic instruction can be fetched in the memory and another one in the register bank).

With requested performances ranging from 1 to 10 MIPS, only a 1–10-MHz clock is required when a CPI of 1 is achieved. And, a simple three-stage pipeline minimizes the amount of hardware and power consumption.

In advanced technologies, a clock cycle of 100 ns to 1 μ s is long enough to execute several functions in series (e.g., fetch and decode), resulting in rather long combinatorial paths and short pipelines.

CoolRISC ARCHITECTURE

The CoolRISC is a three-stage pipelined core, as illustrated in Figure 1. One instruction is provided to the pipeline at each clock. Arithmetic and load instructions are executed in the three pipeline stages.

Data from the preceding instruction is bypassed to the next instruction if needed, which results in no load delay. This bypass occurs in the hardware.

The store instruction is executed in the two first stages of the pipeline. The branch instruction is executed in only one clock in the first pipeline stage, and the condition provided by the preceding instruction is always available.

This way, no branch delay occurs in the CoolRISC core, resulting in a strict CPI of 1. A 1-MIPS performance is therefore achieved at 1 MHz. But, the same doesn't hold for other 8-bit pipelined microprocessors like the PIC, Nordic μ RISC, and MCS-151 and '251.

As Table 1 demonstrates, reducing CPI is key to high performances. Here, you see the number of instructions, bits in the ROM memory, executed instructions, and clocks used to execute the same small routine on various microcontrollers.

For each instruction, the first half clock is used to precharge the ROM program memory. The instruction is read and decoded in the second half of the first

Table 1—Compared with conventional architectures, CoolRISC (with a CPI equal to 1) has a significantly lower number of executed clock cycles.

| | Instruction Code | Bits Code | Executed Instructions | Executed Clocks | CPI |
|--------------|------------------|-----------|-----------------------|------------------|----------------|
| ST62xx | 12 | 152 | 60 | 2704 | 45 |
| COP800 | 12 | 120 | 60 | 2000 | 33 |
| 8048 | 8 | 112 | 35 | 1125 | 32 |
| Z86Cxx | 8 | 168 | 35 | 692 | 20 |
| 68HC05 | 11 | 160 | 59 | 226 ¹ | 4 ¹ |
| PIC16C5x | 11 | 132 | 59 | 300 | 5 |
| Punch | 12 | 216 | 74 | 296 | 4 |
| CoolRISC 81 | 12 | 192 | 58 | 58 | 1 |
| CoolRISC 88 | 10 | 180 | 58 | 58 | 1 |
| CoolRISC 816 | 10 | 220 | 58 | 58 | 1 |

¹ Refers to the internal *E* frequency that is two times slower than the oscillator frequency.

clock. A branch instruction also executes during this half, which is long enough at 1–10 MHz to perform all the necessary transfers.

For a store instruction, only the first half of the second clock is used to store data in the RAM. For an arithmetic instruction, the first half of the second clock is used for reading an operand in the RAM or register set, the arithmetic operation is performed in the second half of this second clock, and the first half of the third clock is used to store the result in the register set. Figure 2 shows the CoolRISC 816 architecture with 16 registers.

A hardware stack of 1–8 levels handles CALL instructions. This setup is more efficient (i.e., has fewer executed instructions) than using RISC-like branch and link instructions, which implies that the return address must be saved by software. But in case the hardware stack is full, this branch and link instruction is also available.

Figure 3 depicts the generic instruction set of the 8-bit CoolRISC 816 with 16 registers. Instructions are 22 bits wide. Branch and CALL instructions provide a 16-bit direct addressing of

the 64-KB program memory as well as an indirect addressing through the 16-bit IP ROM index. The hardware stack can be read via POP and PUSH instructions.

Load and store instructions are available with a NOP as an arithmetic logic unit (ALU) operation. They can be executed conditionally on the carry flag.

Only 16 ALU operations can be specified for the ALU instructions with immediate 8-bit data. The other ALU instructions provide a 5-bit field to specify the ALU operation. There are two types of ALU instructions—RISC-like (use only registers) and register memory (the operand comes from the data memory).

The 816 has a number of addressing modes. One goes directly to the first page of data memory, and several others use one of the four 16-bit index registers to address the 64-KB data memory.

The index registers can be post- or premodified while adding or subtracting a 7-bit offset. Another addressing mode is based on one of the four index registers, with the 8-bit R3 register as an offset.

The 816 ALU operations with the post-fixed A (arithmetic) modify the flags so that the signed numbers in the two's-complement representation are supported. The most-significant-bit result of an 8 \times 8 multiplication (MUL) is stored in the destination register, and the least significant bits are stored in the accumulator.

The MSHx instructions can shift several bits at a time using a register and

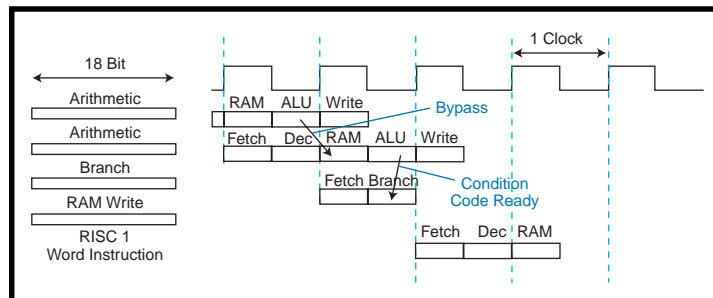


Figure 1—CoolRISC features three-stage pipelined scheduling and instruction execution. A CPI of 1 is always respected even for a branch instruction. The evaluation of the branch condition, as well as the possible execution, is performed in the second half of the clock period.

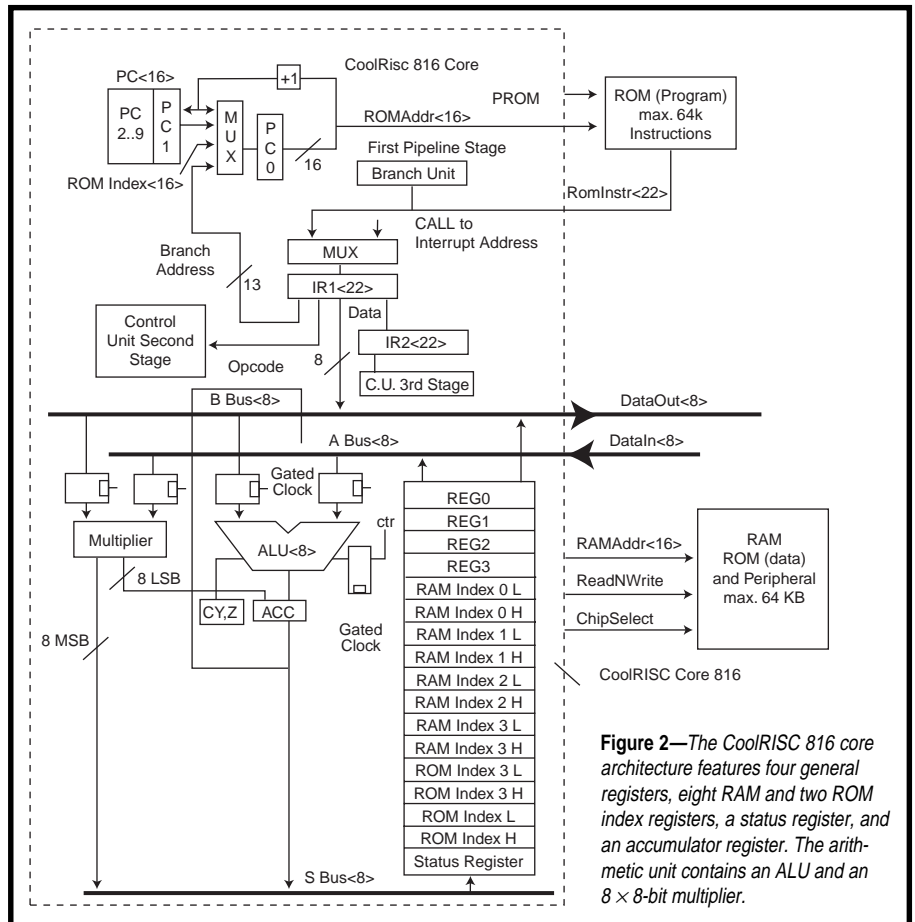


Figure 2—The CoolRISC 816 core architecture features four general registers, eight RAM and two ROM index registers, a status register, and an accumulator register. The arithmetic unit contains an ALU and an 8 × 8-bit multiplier.

the accumulator. The bit instructions (i.e., TSTB, SETB, CLRB, and INVB) test, set, reset and invert a single bit in a register.

The index registers can also be used as 8-bit general-purpose registers if the ROM index isn't used or if less than four index registers are used for the data memory. The accumulator stores the last ALU result, which can be used as an intermediate result for the next ALU operation.

LOW-POWER DESIGN TECHNIQUES

Another important issue in designing 8-bit microcontrollers is power consumption. The CoolRISC cores rely extensively on the gated-clock technique.

The ALU, for instance, features input and control registers that are loaded only when an ALU operation has to be executed. During the execution of another instruction (e.g., branch or load and store), these registers are not clocked, so no transitions occur in the ALU. Thus, power consumption is further reduced.

A similar mechanism is used for the instruction registers. In a branch instruction, which executes only in the first pipeline stage, no transitions occur in the second and third stages of the pipeline.

Note that gated clocks can be advantageously combined with the pipeline architecture. The input and control registers implemented to obtain a gated-clocked ALU are naturally used as pipelined registers.

Another interesting low-power feature is the support of hierarchical memories. Frequently used codes are stored in a small, fast ROM, whereas infrequently used parts of the code sit in a large but slow memory.

Most of the time, the small ROM is read, which reduces power consumption. This mechanism can be combined with an automatic reduction of the operating frequency when the large memory is read.

Via software, the internal frequency can be reduced by a factor of 2–16 while using a `FREQ` instruction in which the division factor is programmed. The

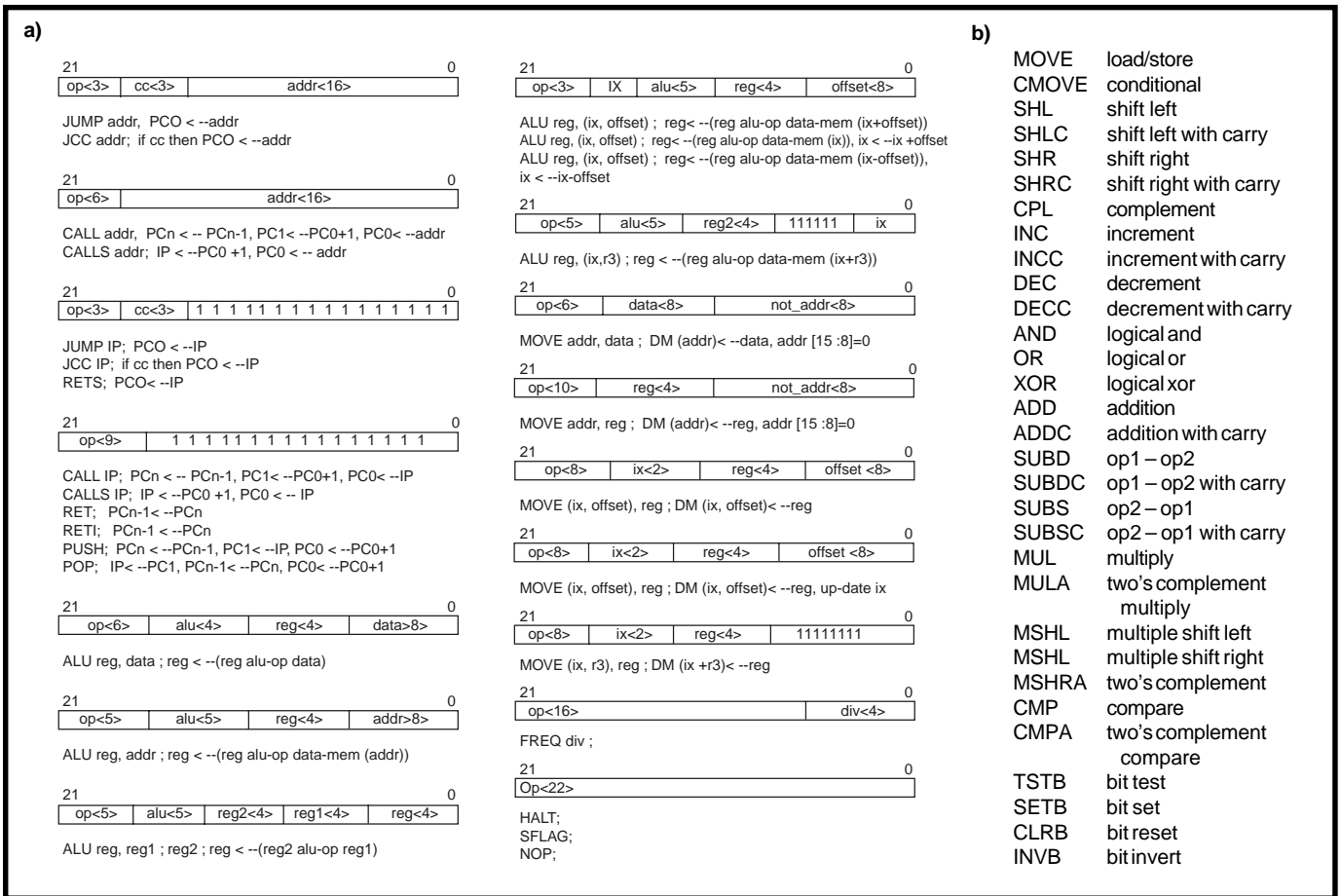


Figure 3a—The eight-bit CoolRISC 816 uses 22-bit instructions with variable-width fields. **b**—A great deal of the 816's reduced instruction set is devoted to arithmetic-logic unit operations. Operations with a post-fixed A modify the flags so the signed numbers in the two's complement are supported.

microcontroller can also be in standby mode during a HALT instruction.

Also noteworthy is a multiprocess mechanism that shares data address spaces with several microprocessors or intelligent peripherals. During clocks in which the microcontroller doesn't access its own data memory, other microcontrollers connected in parallel have free access to this memory.

Management of shared memories is handled by a request/acknowledge mechanism, which can also be used for intelligent peripherals like DMAs.

AT THE CORE

The CoolRISC 816 core comprises 16 registers, 22-bit wide instructions, a maximum ROM program memory of 64K × 22, and a maximum RAM data memory of 64 KB. It also has one ROM index register and four index RAM registers, eight RAM addressing modes, three interrupts, and two events to wake up.

The core provides a branch and link instruction as well as a multiprocessor mechanism you can use for DMA purposes. It also contains an 8 × 8 parallel-parallel multiplier.

With its 19,000 transistors, CoolRISC achieves 70 μA at 1 MIPS at 3.0 V in a 1-μm process (i.e., 4700 MIPS/W at 3.0 V). At 1.5 V, it achieves 32 μA at one MIPS in a 1-μm process (i.e., 21,000 MIPS/W at 1.5 V).

These figures are given for the microcontroller cores only, and they differ from the performances given in Figures 4 and 5, where small embedded low-power memories are taken into account.

Keep in mind, though, that the CoolRISC 816 was integrated in a test chip with small embedded low-power memories.

DEVELOPMENT TOOLS

The development tools are based on the GNU (Gnu's Not Unix) toolset, including a GCC (GNU C compiler),

assembler, instruction-set simulator, debugger, and hardware emulator.

The CoolRISC IDE (Integrated Development Environment) consists of three parts—the target configuration, project manager (including source code editors), and tools for debugging source-level code. The target-configuration editor, for instance, is used to choose a given CoolRISC core.

At the project-manager level, the tools include:

- a project browser and configuration editor
- a source-code editor
- a C compiler (cc)
- macro-assemblers (as)
- linkers (ld) and librarians (ar)

As I mentioned, the C compiler for the CoolRISC 816 is based on the GNU toolset. The GNU toolset, available in the public domain, was chosen for the quality of the generated code, and the GCC is a parameterizable compiler

which needs a description of the microprocessor and its resources.

The C compiler must take low-power aspects of the system into account by minimizing the number of executed instructions for a given task. There are several ways to do this. You can minimize the memory accesses, unroll loops, minimize the number of CALLS and RETURNS, and choose instructions with the lowest energy per instruction.

To use the GCC, the original CoolRISC 816 had to be modified. Some instructions were changed to facilitate the creation of stacks in data memory.

Four index registers are available (some were added due to the compiler) to produce better generated code, primarily for routines accessing the data memory. These modifications help lower the number of executed instructions, which saves power.

The C compiler calls routines written in assembly language. Several arithmetic routines are available, such as single, double, and quadruple precision unsigned or signed multiplication and division, as well as 24- and 32-bit floating-point routines.

The CoolRISC GCC provides many GNU-based optimizations of the generated code. These improvements reduce the number of instructions, while using a good register allocation to the variables and optimized instruction scheduling.

During testing, the first version of the CoolRISC C compiler generated good-quality code. It was also fully tested with an ANSI/ISO validation suite.

The powerful macro-assemblers (as) were developed with the GNU toolset. The assembly process consists of two phases—preprocessing, which expands macros and loops, and assembly.

The assembler generates relocatable code. A set of assembly routines were written to provide the user with a CoolRISC library of arithmetic functions.

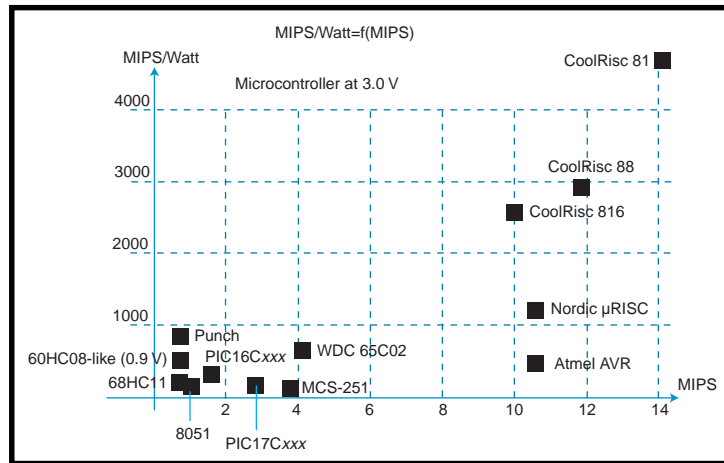


Figure 4—This graph shows the MIPS-per-watt numbers for different cores at 3 V. A CPI equal to 1 results in better MIPS-per-watt figures, which are key for low-power applications.

A linker (ld) enables you to insert the needed routines into your code. The linker provides an absolute object file that you can extract data ROM content from via GNU binary utilities software.

The source-level debugging tools include:

- instruction-set processor simulator
- GDB debugger
- processor hardware emulator

The instruction-set simulator is a C++ software model of the CoolRISC core that can execute each instruction of the CoolRISC set. This simulator is interfaced with GDB, the GNU debugger.

The debugger can be used with both source-C and source-assembly code in various modes (e.g., step by step, continuously, or with break-points). The contents of the C variables, registers, program counter, flags, call stack, and data memory can be displayed in several windows.

The GDB can also interface with a hardware emulator containing a ROMless CoolRISC chip. When you're using the 816 C compiler, pointers on the C code as well as on the assembly code are available during the debug of the application program. You can also use these debuggers for the processor peripherals.

These GNU-based development tools pro-

vide text windows with the Emacs editor. These tools can be run on a Sun/Solaris or a PC running under Windows 95 or NT.

The next version of the CoolRISC development tools will operate on a PC running under Windows 95 or NT with an enhanced user-friendly graphical window-based environment named CoolRIDE. It will provide advanced facilities for project management as well as enhanced editors.

PERFORMANCE COMPARISON

Figure 6 shows MIPS performances as a function of the operating frequency. With a large CPI, 1-MIPS performance requires a large frequency. But, that's not the case with the CoolRISC family with a CPI of 1.

Note that the MIPS-per-watt figure does not depend on the operating frequency. As Figure 4 shows, the performances in MIPS per watt of the microcontrollers with a large CPI are quite low. This data was computed from various datasheets, so these results are only approximates. Only orders of magnitude are important here.

Compared to most existing 8-bit microcontrollers, an improvement of the MIPS-per-watt figure by a factor of about 10 can be reached with new 8-bit architectures designed with low-power techniques.

This factor improves even further if the power supply can be reduced while satisfying speed performances. And as you see in Figure 5, the MIPS-

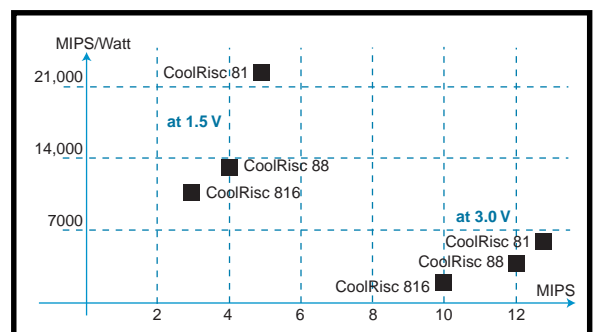


Figure 5—The MIPS-per-watt figures can be improved further by operating at a lower voltage level.

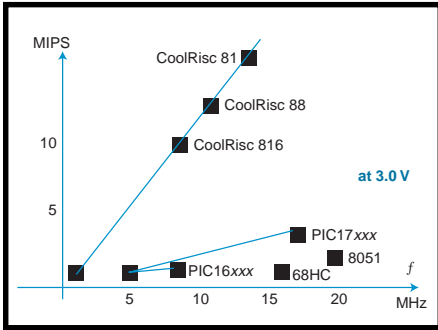


Figure 6—Compared with other architectures, the CoolRISC architecture with a CPI of 1 (always) has a very good MIPS figure as a function of the operating frequency.

per-watt performances of the CoolRISC core improve when the device runs at 1.5 V.

LOW POWER = MORE POWER

More and more often, low-power and low-voltage embedded microcontrollers are required in portable applications.

Power reduction should be addressed at the software and architecture levels. Lower power requirements are also achieved by reducing the number of executed instructions for a given task.

In this article, I've introduced you to one 8-bit RISC-like microcontroller family that achieves 1 CPI. It employs several techniques such as gated clocks, hierarchical memories, and pipelines to further reduce power consumption while maintaining speed performance.

The result: a MIPS-per-watt figure that exceeds conventional architectures by at least an order of magnitude. 📦

Some development tools for the CoolRISC microcontrollers were designed in collaboration with Logical System Laboratory (LSL), EPFL, Lausanne, Switzerland (Professors E. Sanchez and D. Mange). The development of the C compiler was financed by the MicroSwiss Groupe National de Support, project TR-CM-006.

Vincent Rikkink is the manager of the microcontroller and DSP business division at Xemics, and he has also worked for Philips Oscilloscopes as well as the Swiss Center for Electronics and Microtechnology (CSEM).

Vincent's research interests are in design automation and digital ICs for telecom applications.

REFERENCES

- M.J. Flynn, *Computer Architecture, Pipelined and Parallel Processor Design*, Jones and Bartlett, 1995.
- J.-M. Masgonty et al., "Low-Power Design of an Embedded Microprocessor," ESSCIRC'96, Neuchâtel, Switzerland, September 16-21, 1996.
- G. Myklebust, "The AVR Microcontroller and C Compiler Co-Design," 3rd European Microprocessor and Microcontroller Seminar, Heathrow, UK, November 6, 1996, 164-170.
- C. Piguet et al., "Low-Power Low-Voltage Microcontrollers: Architectures and Performances Comparison," Low-Power Low-Voltage Workshop at ESSCIRC'96, Neuchâtel, Switzerland, September 20, 1996.
- H.S. Stone, *High-Performance*

Computer Architecture, Addison-Wesley, Reading, MA, 1987.

A. Wild et al., "A 0.9V Microcontroller for Portable Applications," ESSCIRC'96, Neuchâtel, Switzerland, September 17-19, 1996.

SOURCES

CoolRISC

Xemics
Maladière 71
CH-2007 Neuchâtel
Switzerland
+41 32 720 56 83
Fax: +41 32 720 5770
www.xemics.ch

Sulzer Microelectronics, Inc.
44350 Grimmer Blvd.
Fremont, CA 94538
(510) 656-2033
Fax: (510) 656-0995
www.sulzermicro.com

I R S

407 Very Useful
408 Moderately Useful
409 Not Useful

Picaro

FEATURE ARTICLE

Tom Napier

A Stamp-like Interpreted Controller

For years, Tom's been itching to control the instruction sets of processors. Using a PIC, some memory, and an interpreter, he bypasses the processor hurdle and writes his own language. He shows you how to do it, too.



or as long as I've been using microprocessors—and I was designing the Intel 8080 into radiation-monitoring equipment in 1977—I've always itched to have control over the instruction set. The makers always seem to leave something out.

When I first encountered Microchip's PIC microcontrollers, I realized that a chip with a built-in EPROM and a 200-ns instruction time could emulate, and outrun, other microprocessors. And if you write an interpreter, you can design any instruction set you wish.

Two years ago, I started developing a PIC-based tokenized Forth engine, but that idea went on the back burner. Then, inspired by Sojourner, I started designing a computer to control models. It would store its software in an EEPROM, so I could change its program on the move. Since the EEPROM

has a serial interface, execution would be relatively slow but adequate for real-time control applications.

I didn't want to reinvent the Basic Stamp and I wasn't sure implementing another Forth interpreter would be all that interesting, so I designed my own machine language.

Since this little computer would be more adventurous than most, I decided to call it "Picaro." (I was tempted to call it "Picard" but decided I couldn't afford a fight with Paramount's lawyers.)

HARDWARE IMPLEMENTATION

Picaro's hardware couldn't be much simpler. As you see in Photo 1, it's built in a 2.1" × 1.35" × 0.6" plastic box on a piece of perf-board cut to fit.

The box contains an 18-pin PIC16C56 in a low-profile socket, a 24C16B EEPROM (also from Microchip), 16-pin I/O connector, stereo jack socket, a few resistors, a crystal, and a couple other parts, as shown in Figure 1. There's room left over for a DIP chip having up to 20 pins, so you could add, for example, a serial DAC or ADC chip.

The I/O connector carries the power, reset, and timer pins and is compatible with a 16-pin ribbon cable connector. It has eight pins that can be programmed to be inputs or outputs.

Each pin has a 10-kΩ pull-up resistor and a 1-kΩ series resistor, and connects to one pin of the PIC's eight-bit ports. This combination enables outputs to supply a voltage or current, and it protects the microcontroller from accidental overload.

When specified as an input, a connector pin can be connected directly to a switch contact (e.g., a limit switch on a moving part) without external components. As an output, each pin can drive an LED directly, but it can also drive a Darlington transistor and switch several amps on and off.

Table 1—When Picaro is in system mode, it interacts with the user with single-letter commands. When it's ready for a command, it outputs ">" and waits for input to be typed in.

| | | |
|---|-----------|-------------------------------------|
| D | Addr | Display 16 bytes from the EEPROM |
| E | Addr | Display and edit the EEPROM code |
| F | Addr Byte | Fill 16 locations with the byte |
| G | Addr | Go to address and start interpreter |
| I | | Display the input port |
| O | Byte | Output the byte to the port |
| R | Addr | Read and edit any byte from RAM |
| U | | Upload a program file (see sidebar) |

Two pins of the PIC's four-bit port connect to the EEPROM, which stores the user's program. The other two pins, with the addition of some discrete components, make up a full-function serial port, which is wired to the stereo jack.

This jack lets you program and control a model from the serial port of a portable computer (in my case, an ancient Tandy Model 100). No RS-232 driver chips or negative voltage converters are necessary. The negative output voltage comes from the device being driven, while a diode blocks the incoming negative signal voltage.

When the jack plug is removed, the serial output switches to the 16-pin connector, enabling the model to have its own communications facility by radio, ultrasound, or infrared. The two communication links can run at different data rates.

BUILDING PICARO

If you've ever assembled a Swiss watch, this project should be easy. Naw, I'm kidding, but you do need a delicate touch with a fine-pointed soldering iron.

To queue the parts in, I cut short one row of pins on the I/O connector and wired them to cordwood-mounted series resistors. The 10-k Ω SIP is mounted under the connector pins and the eight-bit port pinout is out of order just to avoid lumpy wiring (shades of the S-100 bus).

I also drilled out the board so the PIC socket can be pushed into it, and

then I cut down the protruding pins on the underside. Now, the lid closes tightly even when a windowed chip is used. The crystal and bypass capacitor fit inside the socket.

I bought the 6.144-MHz crystal, PIC, and EEPROM from Digi-Key. They also have suitable SIPs and right-angle connectors. The plastic case and serial I/O jack came from Radio Shack. The serial output transistor can be just about any small PNP type. I used an old European one with a low profile.

FIRMWARE IMPLEMENTATION

Picaro runs two languages. It can operate in system mode, responding directly to commands entered at a terminal connected to the serial port. These commands, listed in Table 1, enable the user to load and modify a program, read and modify memory, read or drive the I/O port, and initiate interpretation of a program.

The Edit command displays the byte at the current address and waits for user input. If it's a hex byte, it is written to that address, then the next address, and its contents are displayed. Entering a space steps to the next address without changing anything, and Return exits you from the editor.

The RAM editor uses the same syntax, except that it doesn't step. It exits on any non-hex character.

The on-chip EPROM also contains the interpreter. In interpreter mode,



Photo 1—The Picaro fits in a small space, but its communication and control capabilities make it a giant.

Picaro fetches and executes eight-bit instructions from the user's program.

This program is stored in EEPROM and retained when the power is off. You can change it whenever you wish, even while the model is operating. On powerup, the user's program, which starts at address 0 of the EEPROM, is executed until either a serial input is detected or a "jump to system" instruction is encountered.

I modeled this interpreted language on the machine languages of many small microprocessors, and almost all instructions are coded as single bytes. It has a regular structure, partly to make it easy to write and partly to make the interpreter as simple as possible.

Despite its simplicity, however, this language can perform almost any programming task. I even threw in an 8- \times 8-bit multiply instruction.

USER-PROGRAM MEMORY

The user program is contained in a Microchip 24C16B EEPROM. This 8-pin DIP chip has 2048 programmable locations, each containing 1 byte. It uses a serial input and output that can run at 400 kbps.

With a 6.144-MHz crystal driving the PIC, the transfer rate is 192 kbps. Reading and interpreting an instruction takes around 60 μ s (i.e., the computer executes about 17,000 instructions a

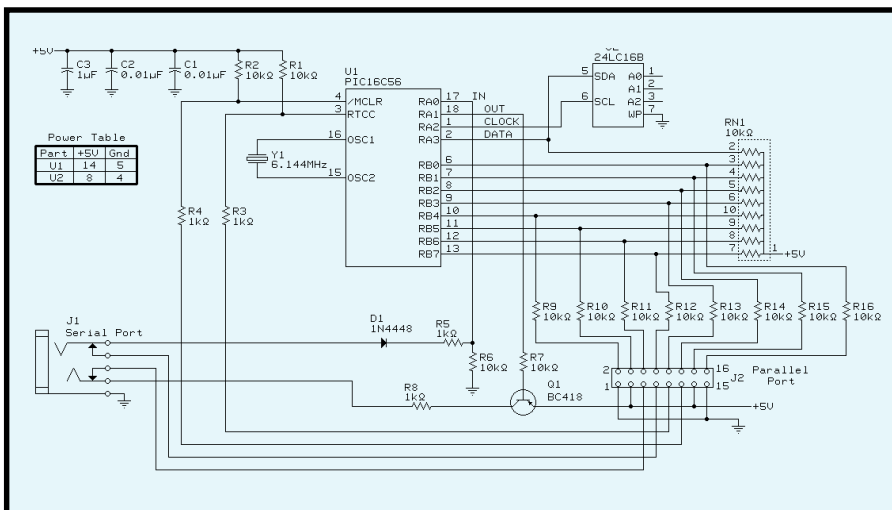


Figure 1—On the Picaro, when the serial connector is unused, serial I/O is present on the parallel port.

| Register | Function |
|----------|---------------------------------------|
| 0 | The accumulator; where results end up |
| 1 | Can be used as an index register |
| 2 | Can be used as a multiplicand |
| 3 | Can be used as a multiplicand |
| 4 | General-purpose register |
| 5 | General-purpose register |
| 6 | General-purpose register |
| 7 | General-purpose register |
| 8 | General-purpose register |
| 9 | General-purpose register |
| A | General-purpose register |
| B | General-purpose register |
| C | General-purpose register |
| D | General-purpose register |
| E | Low byte of return address |
| F | High byte of return address |

Table 2—Registers 0–7 can be incremented and decremented.

second), which should be fast enough to control even a complex model in real time.

Writing to the EEPROM is much slower than reading it. To change one instruction takes up to 10 ms. Luckily, up to 16 bytes can be loaded sequentially and written simultaneously, so uploading a program takes ~0.5 ms per byte.

In practice, the 9600-bps serial link from the terminal is the bottleneck. Memory locations can be rewritten at least 100,000 times, so memory life isn't a problem.

READ/WRITE MEMORY

The PIC16C56 has 32 one-byte RAM registers, which Microchip confusingly calls "files." (Did someone copyright "register"?)

Eight of these have special functions. For example, one is the program counter and others are I/O ports.

I allocated the remaining 24 registers as eight system registers and 16 interpreter registers. That is, the interpreted program has access to 16 RAM

locations, which are listed in Table 2.

The interpreter is denied access to the system registers. However, it can read and write Port B, access the register storing the interpreter's carry and zero flags, and indirectly access the two system registers that store the interpreter's program counter. As you'll see, the latter are rarely used.

INSTRUCTION SET

This language has 37 instruction types, listed in Table 3. It is accumulator based and has the usual complement of memory fetch and store instruc-

tions. It has two arithmetic and three logical operations as well as register-increment, -decrement, and constant-loading instructions.

It also has bit-set, bit-clear, and bit-test instructions. Its program-flow instructions include skip on test, conditional and unconditional jump, and a subroutine call.

The high four bits of each eight-bit instruction specify its type. Most instructions use their lower four bits to address one of the 16 RAM registers. For example, there are 32 move instructions—16 copy the accumulator to another register, and 16 copy another register to the accumulator.

ARITHMETIC AND ACCUMULATORS

The accumulator serves as the destination as well as a source for all two-parameter operations except multiplication. That is, both arithmetical operations and all three logical operations between two variables use the accumulator as one input. Any of the 16 registers can act as the other input. The result ends up in the accumulator. If it needs to be placed in another register, a specific move instruction is required.

Accumulator bits can be changed one at a time by bit-set and -clear

| Sixteen unique instructions | | | Register move instructions | | |
|---|-------|---|---|-----------|--|
| 00000000 | NOP | Does nothing | If RRRR = 0, the move is to or from the flags register | | |
| 00000001 | PRTD | Set port direction from next byte | 0110RRRR | MOVR A R | R := A no flag effect |
| 00000010 | PRTO | Move A to port | 0111RRRR | MOVA R A | A := R no flag effect |
| 00000011 | PRTI | Move port to A | Move immediate instructions | | |
| 00000100 | SERO | Send A to serial out | 1000XXXX | MVIL | A := 0000XXXX no flag effect |
| 00000101 | SERI | Put serial input in A | 1001XXXX | MVIH | A := XXXX0000 exor A no flag effect |
| 00000110 | TYPE | Print counted string from EEPROM | Bit set, clear and test | | |
| 00000111 | SWAP | Swap upper and lower nibbles of A | 1010FBBB | SET/CLR | Set bit BBB of A to the value F |
| 00001000 | SHL | Shift A left | 1011FBBB | SKIS/SKIC | Skip two bytes if bit BBB = F |
| 00001001 | SHR | Shift A right | Register increment/decrement, affect zero flag | | |
| 00001010 | INXO | Move A to indexed register | 1100FRRR | INC/DEC | Increment RRR if F=0, decrement if F=1 |
| 00001011 | INXI | Move indexed register to A | Program control, two-byte instructions, second byte is address | | |
| 00001100 | WAIT | Use next byte as wait period | PPP is the destination page. | | |
| 00001101 | MULT | Multiply registers 2 and 3 | 1101FPPP | JMP/CALL | Unconditional Jump (F=0) or Call (F=1) |
| 00001110 | RET | Return from subroutine | 1110FPPP | JNC/JC | Jump if carry flag = F |
| 00001111 | EXIT | Return to System | 1111FPPP | JNZ/JZ | Jump if zero flag = F |
| Arithmetic and logic instructions | | | RRRR is a register address | | |
| If RRRR = 0, the second operand is the next program code byte | | | | | |
| 0001RRRR | AND R | A := A and R affects zero flag | | | |
| 0010RRRR | OR R | A := A or R affects zero flag | | | |
| 0011RRRR | XOR R | A := A xor R affects zero flag | | | |
| 0100RRRR | ADD R | A := A + R affects zero and carry flags | | | |
| 0101RRRR | SUB R | A := A - R affects zero and carry flags | | | |

Table 3—The Picaro supports 37 instructions. In most, the upper four bits specify the instruction type while the lower four bits reference a register.

instructions. There are eight of each, one for each bit. Accumulator bits can also be tested. Eight instructions specify to skip the next instruction if the corresponding bit is a 0, and another eight test for a 1.

Four- and eight-bit constants can be loaded into the accumulator by the 32 `move immediate` instructions. Of these, 16 set the accumulator to a four-bit value and the other 16 XOR a four-bit constant with the upper four bits of the accumulator. So, loading an arbitrary eight-bit value requires two instructions, but a four-bit constant requires only one.

In theory, since the accumulator is also Register 0, it's a legal input operand. Since the only instruction that's really useful is `ADD 0` (which doubles the accumulator), I preferred using the five Register 0 codes to perform immediate operations on the accumulator. That is, the next byte in the program code is the second operand.

Move operations between the accumulator and Register 0 are also useless, so the flags register is treated as Register 0 for fetch and store instructions only. Therefore, the flag bits can be set, cleared, and tested under program control, thus implementing skip-on-carry and skip-on-zero operations. The accumulator can be shifted left and right through the carry bit, and its upper and lower four bits can be swapped.

INCREMENT AND DECREMENT

One irregularity in the instruction set is that there are only eight register-increment instructions and eight register-decrement instructions. (I ran out of codes.)

Only the lower eight registers can be incremented and decremented, but that shouldn't be a problem. There's also no specific register-clear instruction. You need to load the accumulator with 0 and move it to the register.

SPECIAL CODES

Unique instructions, such as port input and output, are carried out by the 16 opcodes whose high four bits are 0. Allocation of the port bits to inputs or outputs is done on a bit-by-bit basis by the `PRTD` instruction. It takes the next code byte and writes it to the PIC's port direction register. A 1 bit sets an input and a 0 bit an output.

Two instructions of special interest are the indexed read/write pair. By using Register 1 as an index, not only can these instructions access any of the 16 RAM locations, but they can also read and write the upper 240 bytes of page 7 of the EEPROM. Since this area is addressable as program memory, it can be loaded with tables of constants at upload time.

The indexed write instruction, `dare` I mention it, lets the program modify itself. Since anything written is unaf-

ected by a powerdown, it's the ideal place to store measurement results. However, when a byte is written to this area, there is an 8-ms timeout before normal operation resumes.

The `TYPE` instruction reads the next code byte and uses it as a count to output a string from the next n bytes of program memory to the serial port. This allows the program to communicate with people. "Take me to your leader?"

`WAIT` also reads the next code byte. It uses it as a time delay in half-millisecond units, so you can implement a time delay from 0.5 to 128 ms.

The `MULT` instruction multiplies Registers 2 and 3. It puts the 16-bit result back in the same registers with the more significant byte in Register 3.

PROGRAM-FLOW CONTROL

There are six program-flow instructions—unconditional jump, unconditional call, jump on carry set, jump on carry clear, jump on zero, and jump on not zero. They all require two code bytes. The first specifies the type of jump and contains the three-bit page code for the EEPROM, and the second specifies the address within a page.

As well, the bit-test instruction conditionally skips the next two instruction bytes. That is, you can skip a jump or call instruction, an immediate instruction, a two-byte macro, or just a single-byte instruction plus a `NOP`.

Uploading a Program

You can send a code file to Picaro whenever it's in system mode and waiting for a command. This uses the `U` command, which heads the code file to be uploaded. The file format is:

```
Ulf
:0AA0lf
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXlf
- - - more code lines - - -
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXlf
XXXXXXlf
lf
```

where `lf` is a line-feed character and `0AA0` is the program's start address.

All uploads must start at an address whose lowest four bits are zero. Each code line except the last contains 32

characters, 16 hex bytes. The last line contains 1–16 bytes. All lines end with a single line-feed character. The upload terminator is a blank line (i.e., two successive line feeds). Carriage-return characters may be used in place of line feeds. The system ignores all characters between the initial `U` and the `:`, so this is the place to insert a title or version information.

The upload link runs at 9600 bps unless you program it to be slower. To avoid halting the source device, each successive 16-byte line is stored in the PIC's RAM as it is received, overwriting any existing contents. The stored bytes are then quickly transferred to the EEPROM while the line-feed character is arriving. The EEPROM then goes offline and writes them while the next 16 bytes are being transmitted. Picaro echoes a `U` to indicate that the file has been received and stored. This is followed by the `>` prompt for the next command.

| Bit | Function |
|-----|---------------------------------------|
| 0 | Interruptenable. Normally set |
| 1 | Data rate. Set = revert to 9600 bps |
| 2 | Carry flag, set by overflow or borrow |
| 3 | Zero flag, set by result = zero |
| 4 | Low bit of the data-rate setting |
| 5 | Middle bit of the data-rate setting |
| 6 | High bit of the data-rate setting |
| 7 | Reserved for timer control |

Table 4—The flags register contains the zero and carry bits and also sets the bit rate of the serial port.

SUBROUTINES

The ability to call subroutines was worth incorporating, but anything more than a single level of nesting was too much trouble in a machine with limited RAM space. Thus, when a subroutine is called, the next instruction address is stored in Registers 14 and 15. When a subroutine return executes, the contents of these registers become the new execution address.

This arrangement has two side effects. One is that Registers 14 and 15 can be used as normal registers until you call a subroutine. The other is that you can jump to any instruction address by loading it into Registers 14 and 15 and executing a return. This lets you execute a computed jump as you would need, for example, when implementing a case statement.

THE FLAGS REGISTER

The interpreter's zero and carry bits are stored in the flags register and set by the result of an operation. (The carry bit uses the Intel convention, not the Microchip one. It is cleared if the result of a subtraction is positive.)

The flags register also has an interrupt enable flag. Clearing this bit causes unexpected characters at the serial port to be ignored. If it's set, an unexpected character acts as a user interrupt (i.e., it causes an automatic return to the system).

Another bit controls the data rate used when returning to the system. If it is set, the system default rate is used (i.e., 9600 bps). If it is cleared, the data rate remains at the preset rate.

Three bits of the flags register store this data rate. This can be set from 300 to 38,400 bps by setting these three bits from 111 to 000. Writing to the flags register (via `MOVR 0`) sets up

the new data rate. Table 4 summarizes this information.

PROGRAM COUNTER

The EEPROM has the useful property that once you have sent a memory address to it, you can read sequential bytes indefinitely without sending a new address. You can execute instruction after instruction without worrying about the current program address. To execute a jump, send a new address to the EEPROM since you don't care where it was before the jump.

Unfortunately, when you call a subroutine, you must know where to return to. The bad news is that there's no method of reading back the address counter of the EEPROM, so you have to track the current program address.

Tracking is done via a phantom program counter—two bytes of the system RAM which track the internal counter of the EEPROM. Every time an instruction is executed, the phantom program counter is incremented, and every time a jump is made, the new address is loaded into the EEPROM and phantom counter. When a subroutine is called, the phantom counter's contents are saved as the return address.

The only other use for the phantom counter is when an indexed read or

write to the EEPROM occurs. The EEPROM's address counter is changed to point to the indexed address on page 7. After the operation, the EEPROM's program address counter is automatically restored from the phantom counter.

The interpreter has no direct access to the phantom counter, but if you want to implement a program-counter-relative operation, you can read it indirectly by executing a call to the next instruction. This action copies the program counter into Registers 14 and 15.

TIMER REGISTER

The microcontroller has a timer register which can count prescaled clocks or external events. In this implementation, it controls the timing of the serial port.

The prescale ratio is changed to set different data rates. Therefore, the timer input pin of the parallel port has no function. You can change this if you want to use the timer for something else. However, the interpreter's `WAIT` instruction performs some of the same functions.

PROGRAMMING PICARO

If you have experience in programming a register-based eight-bit micro-

Listing 1—First upload or type in the Picaro test program, and then enter `G 000`.

| Addr | Mnemonic | Code | Function |
|------|----------|------|---------------------------------|
| 000 | PRTD | 01 | Use next byte as port direction |
| 001 | | F0 | Low four bits are outputs |
| 002 | MVIL 5 | 85 | Set accumulator to 5 |
| 003 | PRT0 | 02 | Write it to port |
| 004 | CALL | D8 | Subroutine call, page 0 |
| 005 | | 4A | Call address |
| 006 | PRTI | 03 | Read port |
| 007 | SHL | 08 | Shift MSB to carry |
| 008 | JC | E8 | Jump if carry, page 0 |
| 009 | | 13 | Destination address |
| 00A | TYPE | 06 | Send a string |
| 00B | | 05 | String length |
| 00C | | 48 | "H" |
| 00D | | 6F | "O" |
| 00E | | 77 | "w" |
| 00F | | 64 | "d" |
| 010 | | 79 | "y" |
| 011 | JMP | D0 | Carry on |
| 012 | | 1A | |
| 013 | TYPE | 06 | Send a string |
| 014 | | 05 | String length |

(continued)

computer, such as any of the Intel 80xx series, learning and using this assembly language should be straightforward. Once you figure out what you want your model to do, the easiest way to proceed is to break the functions down into manageable chunks you can pass parameters to.

So, to turn 10° right, call the subroutine Turn Right with a parameter of 10. A sequence of subroutines, combined with tests and jumps, programs the desired operations. (Forth is the ideal language for this type of application, but that's another article.)

Once your program is written in assembly language, you need to translate it into a list of instructions. I do this by hand, which is a strong motivation for keeping the language simple. The biggest job is tracking the instruction addresses since you need to specify where all the jumps are headed.

The listing must be converted into a text file using the 16 bytes per line format specified in the sidebar "Uploading a program." Next, the computer can be turned on, put in system mode, and the program uploaded. The program can be examined and modified while in system mode. The G system instruction starts execution from any address.

On powerup or reset, Picaro uses the highest EEPROM byte, at address 7FFH, to tell it whether it should start executing the user's program immediately or go to system mode and wait for instructions. If this byte is 0, it jumps to the user's program. Any other value tells it to enter system mode.

Since blank EEPROMs seem to contain mostly 1s, Picaro should power up in system mode the first time you switch it on, so you can upload or edit a program. Once your program works, you can edit byte 7FFH to 0 so your program starts every time the model turns on.

Listing 1 is a short sample program for testing Picaro. It can be typed in byte by byte in system mode or sent as a text file in upload format. If port bit 7 is high (default), the output is:

```
Hello
ABCDEFGHJI
OK
```

Listing 1—continued

```
015          48  "H"
016          65  "e"
017          6C  "l"
018          6C  "l"
019          6F  "o"
01A  CALL    D8  Send CRLF
01B          4A
01C  MVIL 10  8A  Specify count
01D  MOVR  4   64  Store count
01E  MVIL  1   81  Specify low nibble
01F  MVIH  4   94  Specify high nibble
020  SERO   04   Send character
021  INC   0   C0  Increment character
022  DEC   4   CC  Decrement count
023  JNZ   F0   Jump back unless done
024          20   Destination address
025  CALL    D8  Send CRLF
026          4A
027  MVIL 13  8D
028  MOVR  1   61  Set index to 13
029  MVIL 10  8A  Accu = 0AH
02A  MOVR  3   63  Set counter to 10
02B  MVIH  5   95  Accu = 5AH
02C  INX0   0A   Write to indexed register
02D  ADD   0   40
02E          29   Add 29H to accu
02F  DEC   1   C9  Decrement index register
030  DEC   3   CB  Decrement counter
031  JNZ   F0   Jump to loop
032          2C
033  MOVA 13  7D  Start testing arithmetic
034  XOR   8   38
035  AND   6   16
036  ADD  11  4B
037  OR   4   24
038  SUB   6   56
039  SWAP  07
03A  XOR   0   30  Check result
03B          67
03C  JZ    E8
03D          45
03E  TYPE  06
03F          04
040          4F  "0"
041          6F  "o"
042          70  "p"
043          73  "s"
044  EXIT  0F  Return to system
045  TYPE  06
046          02
047          4F  "0"
048          4B  "K"
049  EXIT  0F
Subroutine, print CRLF
04A  MVIL 13  8D  CR to Accumulator
04B  SERO   04   Send serial character
04C  MVIL 10  8A  LF to Accumulator
04D  SERO   04   Send serial character
04E  RET    0E   Return
Upload file
U
:0000
01F08502D84A0308E8130605486F7764
79D01A060548656C6C6FD84A8A648194
04C0CCF020D84A8D618A63950A4029C9
CBF02C7D38164B2456073067E8450604
4F6F70730F06024F4B0F8D048A040E
```

If port bit 7 is held low, Hello is replaced by Howdy. If the last line is Oops, then something went wrong.

It would be easy to write an assembler for this language. I didn't need to myself, and in any case, my software runs on an Amiga and wouldn't be much use to Wintel users. Perhaps you can fill the gap.

USING THE SERIAL PORT

Normally, the serial port provides direct interaction with the user in system mode. However, the user's program can also transmit and receive serial data, and it can use a different data rate from the 9600 bps used in system mode. This allows the use of longer range, narrower bandwidth communication paths.

Ideally, serial input only takes place when the computer is waiting for it (e.g., when it has sent a request for new instructions). But, you have to decide how to handle unexpected input. You could ignore it via an interrupt disable flag, but then you have to do a hardware reset if you want to plug in a terminal and diagnose a problem.

If you don't ignore unexpected inputs, you have to decide whether they'll come from the low-speed communication link or the terminal. In the first case, you set the rate reversal flag to "preset" and in the second case to "default."

The same flags control what happens when the program executes a return-to-system instruction. "Tell me what to do next" may be more easily handled by uploading new code over the communications link in system mode, rather than by choosing from a limited number of preset actions in interpreter mode.

GETTING STARTED

To make Picaro work, the system firmware must be written into the EPROM of the PIC chip. There are two ways to do this.

If you have a UV eraser, a PIC assembler, and a programmer, you can buy your own windowed PIC chip and download the firmware from the Circuit Cellar's Web site. Or, you can purchase the preprogrammed parts from me.

WRITING ASSEMBLY LANGUAGE

Picaro's assembly language is supposed to be general-purpose, but your application may require an operation that can't be readily synthesized from combinations of the existing instructions but that's still within the PIC's capabilities. In this case, you're welcome to modify the source code, but please do not make commercial use of this code without my permission.

The firmware is divided into two blocks—system and interpreter—and occupies ~70% of the 1-KB EPROM. A gap between the blocks keeps the interpreter and system code on separate EPROM memory pages to avoid having to swap the page-select bit too much. So, you can change the interpreter's instruction set by modifying only the second page.

In operation, the interpreter reads an instruction, saves its lower four bits as a possible register address, and uses the upper four bits to make a calculated jump to one of 16 handlers. These handlers use the saved lower four instruction bits as a register address, immediate data, or a flag bit and partial program jump address.

If the upper four bits are zeros, the handler makes a second calculated jump, this time based on the lower four instruction bits. This jump ends up at one of the 16 subhandlers containing the code for the 16 unique instructions. After every instruction, the interpreter returns to a housekeeping routine, which sets the zero and carry flags and advances the phantom program counter.

I ended up with no spare codes. At one point, I even implemented an escape instruction, which used the following byte as a pointer to (potentially) 256 further instructions. But, I found I didn't need it and took it out again. If you want to add a function, you need to insert an escape code or replace an existing function by changing its handler.

The only reason to change the system firmware on the first EPROM page is to use a different crystal frequency. Then, the constants controlling the data rate have to be changed, and if the crystal frequency is higher than 6.144 MHz, NOPs need to be inserted into the EEPROM read and write routines.

DESIGN YOUR OWN

So, I finally got to write my own assembly language. I haven't done as much with it as I had expected, but I'm passing it on in the hope that you'll find it useful.

If you have the tools to program PIC chips, you can follow my lead in designing your own custom computer. Picaro makes a pleasant change from systems with 16-MB minimum memories and operating systems no human being can comprehend. ☒

Tom Napier has worked as a rocket scientist, health physicist, and engineering manager. He has spent the last nine years developing spacecraft communications equipment but is now a consultant and writer. In his free time, he develops neat test instruments, debunks pseudoscience, and writes in Forth on an Amiga 3000.

SOFTWARE

Complete documentation and source code for Picaro is available via the Circuit Cellar Web site.

SOURCES

EEPROM and PIC microcontroller

Microchip Technology, Inc.
2355 W. Chandler Blvd.
Chandler, AZ 85224-6199
(602) 786-7200
Fax: (602) 786-7277
www.microchip.com

Digi-Key Corp.
701 Brooks Ave. S
Thief Falls, MN 56701-0677
(218) 681-6674
Fax: (218) 681-3380

Picaro kit

Preprogrammed, nonwindowed PIC16C56, 24LC16B EEPROM, 6.144-MHz crystal, and disk with Picaro firmware and manual ... \$15
Tom Napier
P.O. Box 3155
Maple Glen, PA 19002-8155

IRS

410 Very Useful
411 Moderately Useful
412 Not Useful

42 Nouveau PC
edited by Harv Weiner

46 PCs Move into Motion
Chuck Lewin

53 Real-Time PC
Software Development for RTOSs
Ingo Cyliax

61 Applied PCs
Embedding PC Card
Part 2: Getting in Touch
Fred Eady

EMBEDDED

APRIL 1998



Photo courtesy of
Performance Motion Devices, Inc.

EMBEDDED MODEMS

WinSystems has introduced two 33.6-kbps modems that give PC/104 and STD-bus embedded systems Web access over standard telephone lines.

Application areas include Internet access for embedded Web servers, program download, site-status information upload, and remote-site monitoring.

The **PCM-33.6M** and **MCM-33.6M** offer effective data throughput up to 115,200 bps by using error correction and data compression. These units are offered in either 33.6- or 14.4-kbps data rates. The modems support V.34bis, V.34, V.32bis, V.32, V.22bis, V.22A/B, V.23, and V.21, plus Bell 212A and 103 communications standards.

Both devices integrate the functions of a stand-alone modem and PC-compatible COM port to provide a single-card modem. Each supports the industry-standard enhanced AT command set, which works with off-the-shelf communications software for 'x86 systems. The boards support the Microcom Networking Protocol (MNP) for detecting and correcting errors in high-speed modem transmissions. V.42 LAPM, MNP 2-4, and MNP 10 error correction is supported, as well as V.42bis and MNP 5 data compression.

An onboard FCC Part 68 registered Data Access Arrangement provides the required isolation and protection, allowing direct connection to the Public Switched Telephone Network (PSTN) without additional circuitry. The phone-line connection is through an RJ-11C jack.

The PCM-33.6M is a PC/104-bus module measuring 3.6" x 3.8". It has a 16-bit PC/104 interface and requires a single +5-V supply. The MCM-33.6M is an STD-bus card (4.5" x 6.5"),



containing STD-bus and external serial RS-232 interfaces. A speaker and eight LEDs indicate the status of the modem handshake and data lines. A software-programmable amplifier drives an onboard piezoelectric speaker for monitoring the phone-line signal.

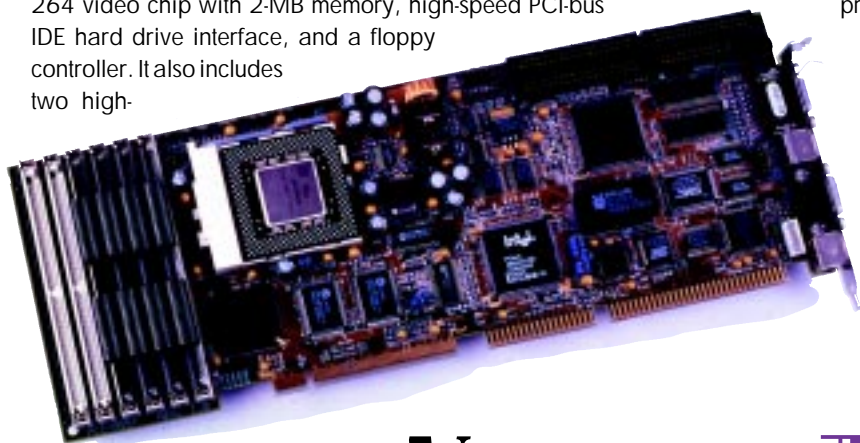
List price for the PCM-33.6M module is **\$250**. The MCM-33.6M module sells for **\$350**.

WinSystems, Inc.
715 Stadium Dr.
Arlington, TX 76011
(817) 274-7553
Fax: (817) 548-1358
www.winsystems.com

#510

300-MHz MMX PENTIUM SBC

The new **PMXVX** single-board computer with MMX and video supports up to 300 MHz of Pentium processing power coupled with a maximum of 384 MB of FPM-, EDO-, or BEDO-type DRAM. This single-board PCI Pentium computer integrates ATI's Mach 264 video chip with 2-MB memory, high-speed PCI-bus IDE hard drive interface, and a floppy controller. It also includes two high-



speed 16550 serial ports, bidirectional parallel port with ECP/EPP, mouse and keyboard ports, real-time clock, watchdog timer, and the new USB and infrared communications port. The Award flash BIOS with plug-and-play is autoconfigurable and provides a full selection menu for custom setups.

The board sells for less than **\$500**.

Interlogic Industries
85 Marcus Dr.
Melville, NY 11747
(516) 420-8111
Fax: (516) 420-8007
sales@infoview.com
www.infoview.com

#511

Nouveau NPC

edited by Harv Weiner



KEYPAD AND LCD INTERFACE

The **AIM104-KEYDISP** is designed to interface to third-party keypad and display products that form the front panel of an industrial control system. The PC/104-sized board can connect to an STN alphanumeric 4 x 20 LCD (68130 controller) or an STN graphics 240 x 64 LCD (68130 controller). A DC-to-DC converter supplies the negative voltage for the LCDs, and a connection for external contrast adjustment is provided.

The board also accommodates up to two 4-/12-/16-key keypads or one 36-/40-key QWERTY keypad. An integrated I/O software library gives instant start-up functions in the form of C function libraries and executable demonstrations.

The AIM104-KEYDISP is priced at **\$65**.

Arcom Control Systems
13510 S. Oak St.
Kansas City, MO 64145
(816) 941-7025
Fax: (816) 941-7807
icpsales@arcomcontrols.com
www.arcomcontrols.com

#512

DEVELOPMENT TOOL

Phar Lap's **TNT Embedded ToolSuite V.9.0** now supports the Microsoft Developer Studio for Visual C++ 5.0 for embedded systems development. Embedded Studio Express provides transparent access to the Developer Studio Integrated Development Environment (IDE) and lets you edit, compile, link and debug embedded and real-time software.

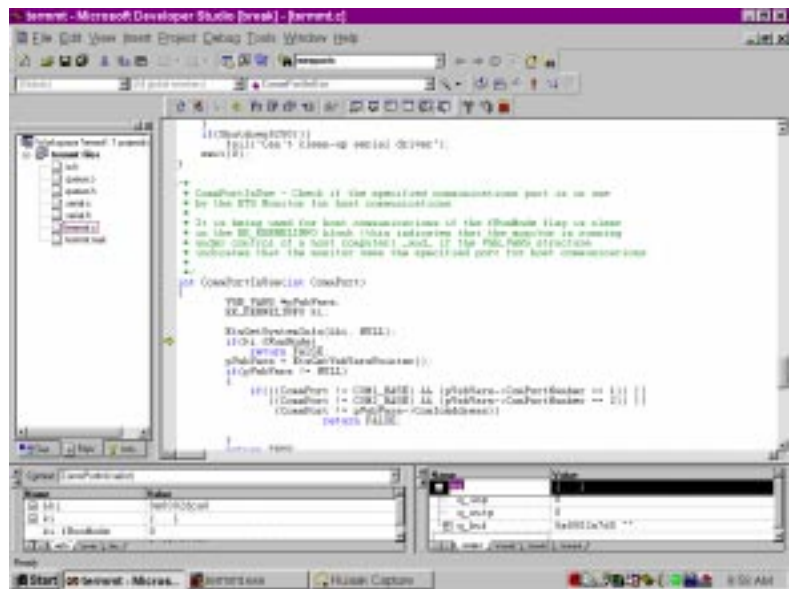
Developer Studio targets Phar Lap's Realtime ETS Kernel, a deterministic, multitasking, small-footprint kernel that supports a Phar Lap-defined subset of the Win32 API. Due to this Win32 API support, this kernel is compatible with off-the-shelf 32-bit compilers and tools and standard 32-bit 'x86 hardware.

The Developer Studio IDE uses a multiple windowing development environment and Embedded Studio Express. It also features the ETS Mail Client, multihoming (two or more networks) and reading internal thread information, and a thread-aware debugger to debug multithreaded real-time applications. The software includes thread timing, reading internal thread information, and naming threads, semaphores, mutexes, events, and pipes. The ToolSuite contains card enablers for ATA flash disks, Ethernet adapters, serial ports, and modems, as well as source code for writing enablers.

TNT Embedded ToolSuite, V.9.0 has a base price of **\$4995**. Upgrade pricing is also available. The host system requires a PC running Windows 95 or NT and Microsoft Visual C++ 5.0. The Realtime ETS Kernel requires 100 KB of memory (minimum) and a serial or parallel port for debugging.

Phar Lap Software, Inc.
60 Aberdeen Ave.
Cambridge, MA 02138
(617) 661-1510
Fax: (617) 876-2972
www.pharlap.com

#513



Nouveau PC

PCI-ISA BACKPLANE

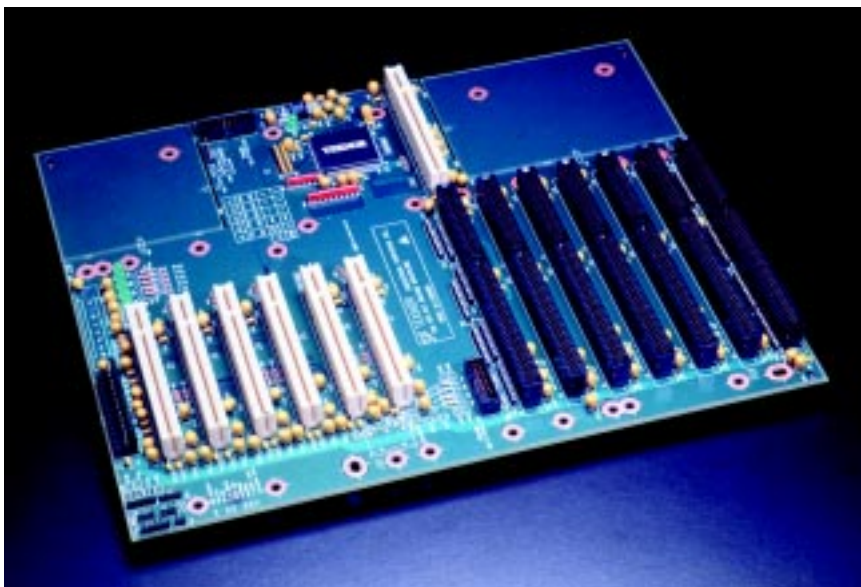
The **PCI-972**, a PCHSA industrial bridge backplane with 12 expansion slots, is designed for OEMs and systems integrators involved in developing and managing PICMG PCI-ISA industrial computer applications and systems. The PCI-972 features six 32-bit PCI slots, one dedicated 32-bit PCI-ISA CPU slot, and six ISA slots. Additionally, each PCI slot provides bus-mastering capability. The unit conforms to IEEE P996 and PICMG Rev. 2.0 specifications.

The six-layer board is designed to minimize EMI and features a 65-Ω impedance (±15%) on PCI and ISA signals. It also provides Baby-AT standard connectors for hard-disk LED, power-on LED, keyboard inhibit, reset, speaker, and a keyboard interface. An industry-standard 12-pin power-supply connector is included as well, and five power-on LEDs monitor all voltages.

PCI-bus loading limits are overcome via PCI-to-PCI bridging. This setup permits a higher PCI-bus slot count—essential in many industrial and telecommunications applications—and enables concurrent bus operations on each PCI bus.

Teknor Industrial Computers, Inc.
7900 Glades Rd. • Boca Raton, FL 33434
(561) 883-6191 • Fax: (561) 883-6690
www.teknor.com

#514



PC/104 DEVELOPMENT MODULE

The **Proto-8** is an extended-length PC/104-compatible board for assembling and evaluating circuits for the PC/104 bus. The large prototyping area enables circuit evaluation prior to PCB etch, so programmers can get involved earlier in the design cycle by writing and testing software well in advance of the final hardware. Potential problems that occur during the merging of hardware and software are revealed sooner and cost significantly less to resolve than if they were addressed later.

The Proto-8 is available in two versions. The B version features a solderless breadboard and easy access to all 8-bit J1/P1 signals. The S version provides a solder-pad/wire-wrap grid with 2000+ holes on 0.1" centers. It is intended for permanent construction, particularly

when only a few articles are needed. Marks are provided so that the length can be trimmed to standard PC/104 module size.

Both versions offer buffered data lines, clearly labeled signal designations, and I/O decoding logic contained in a single reprogrammable device. An optional J2/P2 connector permits access to all 16-bit PC/104 signals.

The B and S versions sell for **\$230** and **\$135**, respectively.



Scidyne
649 School St.
Pembroke, MA 02359
(781) 293-3059
Fax: (781) 293-4034
members.aol.com/
scidyne

#515

Nouveau NPC

PCs Move into Motion

PCs are reshaping motion control. Many specialized motion-language cards are being replaced with a mix of desktop PCs, motion chipsets, and motion peripherals, with the PC carrying much of the load.

PCs are used on everything from the space shuttle to undersea submersibles. As a control platform, the PC has gained popularity because it's cheap, highly accessible, and available in many shapes and sizes.

Motion control, which is the art and science of precisely moving mechanical devices through some path or to some exact position, hasn't escaped this trend. A variety of vendors today provide motion-control products for the PC.

Motion cards are available that provide one, two, four, or even eight axes of motion control. These cards can control DC brush, step, and brushless DC motors.

All of this hardware doesn't necessarily result in a control system that's right for your application, however. The missing ingredient is software. Software means not only the motion-control language or instruction set you write your application in, it also includes tuning tools and exercisers that let you quickly prototype your application.

There are several different types of motion cards, each with its own type of motion-instruction set that is implemented. It's important to understand the differences between them so you can make the right controls choice.

In this article, I look at the different types of PC-based control cards and discuss their architectural differences and their pros and cons. I also discuss some example applications.

Additionally, I give you a look at distributed control, which is gaining popularity as it applies to motion control for the PC.

IT'S ALL IN THE CARDS

Figure 1 shows an overview of a typical PC-based motion-control system. The major elements of the system are the PC, motion card, motor amplifiers, and motors.

For most motion-card products, the type of the motor you use affects the choice of card because the motion peripherals must match the motor type.

For example, step motors may not require quadrature decoding, whereas servo motors always do.

Essentially, however, there are just three types of motion-control cards available for the PC today. This small number of card options may be difficult to determine from the myriad of motion-card vendor claims. Unfortunately, there isn't any standard industry jargon in existence for these three types of cards.

For now, I'll refer to them as motion-language, motion-engine, and motion-peripheral cards. Let's go over each of these in turn.

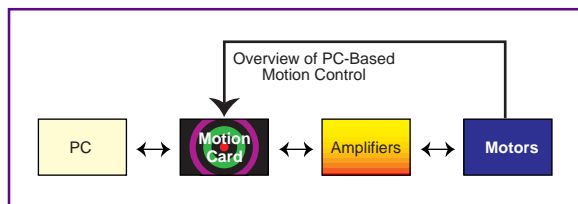


Figure 1—PC-based motion control cards send an analog signal ranging from -10 to +10 V to external amplifiers. These in turn drive the motor at the requested torque level, and encoders feed position information back to the controller.

MOTION-LANGUAGE CARDS

Figure 2 shows the control flow for a PC-based card that executes a complete motion language on the card. The motion language goes beyond merely the ability to perform trapezoidal profile and servo control. It includes constructs for branching and looping, just like C++ or BASIC.

There are two major elements in such a card. The first is a motion-engine chipset, which performs the raw motion calculations such as trajectory generation, servo loop closure, commutation, and step and direction generation for step motors.

The second major element is a program microprocessor that executes the motion language. Generally speaking, this microprocessor interprets the motion language downloaded from the PC into the motion card.

The advantage of this type of architecture is that the motion card can operate totally autonomously from the PC. This capability is a big advantage when the PC has a lot of activity to perform for display, keyboard processing, and so forth.

One disadvantage is that there are no standards for these motion languages, so several vendor-specific languages exist. They run the gamut from BASIC-like symbol-based languages that use two-letter codes to more elaborate systems that borrow from C or Pascal.

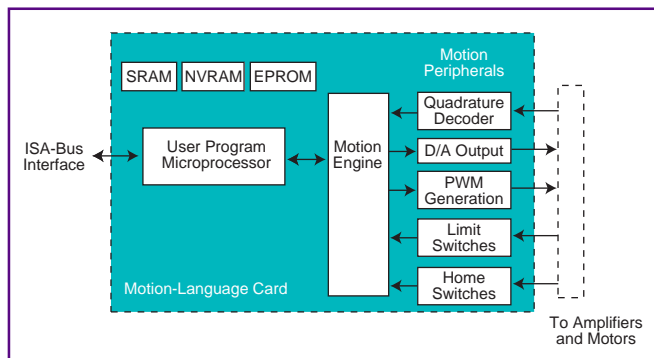


Figure 2—Motion-language cards execute a complete self-standing motion program downloaded by the PC.

Some motion-language cards let the user program the motion application in the native instruction set of the program microprocessor. This blurs the distinction of a motion-language card and a motion-engine card.

What is significant here is that the language is loaded in the card even though it's a standard computer language like C or assembler. So, if the motion card can autonomously control a machine with decision making, branching, and looping, you can consider it a motion-language card.

MOTION-LANGUAGE APPLICATIONS

A wide variety of applications can benefit from a motion-language card approach. Nevertheless, since these cards are slightly more expensive than the other card types, it's worth knowing where these cards really excel.

Ironically, there aren't any applications that require a motion-language-card approach because, when properly programmed, the PC can handle almost any number of simultaneous tasks. Since this is the defining characteristic of a motion-language card, the real implementation question for the designer is: Do I need or want the motion card to run as a separate, parallel task?

The answer to this question may be yes in systems that are not multitasking and require relatively simple interactions between the motion card and system software.

One application that often executes on this type of card is CNC (computer numerical control) and other multidimensional contouring applications. The motion-language card is ideal here because these applications use special file formats that can be interpreted directly by the card. In this mode, the motion-language card essentially runs a shrink-wrapped application that can process file formats like DXF (AutoCAD output), HPGL (plotter control language), and G&M codes (CNC control language).

When running in this mode, the motion card is loaded with the standard format file. The card then manages the details of controlling each motion axis to follow the contour specified in the file.

Motion Processors Off-the-Shelf

ICs that provide dedicated motion-control functions are available off-the-shelf from a variety of vendors. These so-called motion processors, like the ones shown in Photo i, provide canned motion functions like S-curve profiling, servo control, and pulse and direction generation.

One of the advantages of buying a motion-engine card that uses an off-the-shelf motion processor is that you can change card vendors at a later time as long as the card uses the same motion processor. Another advantage is that if your product volume increases to the point that you want to consider designing your own custom card, then by buying the motion processor and building your own card, you eliminate the need to rewrite your software.

Motion processors have been around for a while. When selecting a specific product, be sure that development kit cards and ample software and application notes are available. Designing your own card is more complicated, but the potential cost savings of an embedded approach to motion control can pay off quickly.

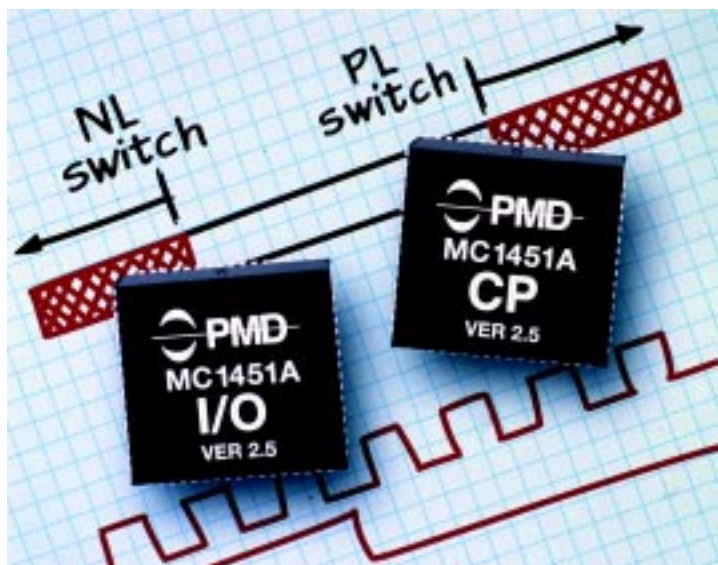
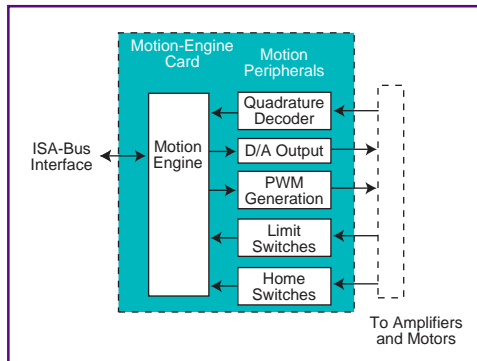


Photo i—The MC1451A chipset is not only used by designers constructing their own embedded motion controllers, but it also serves as the basis of several motion-engine PC-based cards.

Figure 3—Motion-engine cards execute high-level motion sequences sent by the PC. The PC provides the looping and branching language facilities.



MOTION-ENGINE CARDS

Figure 3 shows a different architecture known as a motion-engine card. The primary distinction between this card and the motion-language card is that there is no program microprocessor.

In this type of card, the motion engine is usually an off-the-shelf motion-processor chip that directly accepts instructions from the PC (see sidebar "Motion Processors Off-the-Shelf"). The motion engine provides a motion-instruction set, and the PC provides the high-level language control.

This setup is an advantage when the control application will be written on the PC. The PC application simply sends a sequence of motion instructions to the motion-engine card, and the motion-engine card executes them, notifying the PC when the moves are complete. The PC coordinates a variety of resources on the bus, motion and otherwise.

By comparison, in a motion-language card, this type of coordination is more difficult because the card operates as a sort of parallel processor which must be explicitly synchronized to the PC code.

Typically, motion-engine cards offer complete point-to-point moves, servo control, and most functions that motion-language cards provide. However, the motion-engine card doesn't attempt to create an environment to execute the user's application code. This code resides in the PC.

Motion-engine cards have become more popular recently because they're less expensive than motion-language cards (there's less hardware) and because they encourage the use of standard languages.

Many vendors of motion-engine cards provide C- or BASIC-compatible libraries which, in effect, become their own sort of motion language, except that the resulting language, by definition, is known to many people and is not vendor specific.

MOTION-ENGINE APPLICATIONS

Motion-engine cards are used in an increasing variety of applications, particularly

as the desire for standard computer languages increases and as the availability of multitasking OSs improves. Semiconductor capital equipment, medical automation, laboratory automation, packaging, printing, and scientific instruments are just a few areas that frequently use these cards.

With most of these applications, the system software is written on the PC. So, why not write the motion-language code there as well? To these designers, writing part of the application in C or BASIC and part in a vendor-specific motion language is cumbersome.

CNC is also a popular application, although with this type of card, the PC performs much more of the contour-file processing than for a motion-language card. Motion-engine cards cannot directly process CNC codes, so the PC is enlisted to perform this function and the motion engine is downloaded with an array of move vectors.

Although it sounds complicated, this approach is quite popular because a PC is an ideal platform to compute complex shapes and paths. These calculations are performed in floating point—something the PC excels at!

MOTION-PERIPHERAL CARDS

As you can see in Figure 4, the key distinguishing characteristic of motion-peripheral cards is that there's no motion

Programming for PC-based Motion Control

How do you program a PC-based motion card? The answer depends on the type of card. Motion-language cards have facilities for branching and looping and executing the user's code directly on the motion card. One popular vendor implements a complete motion language using two-letter codes that implement a BASIC-like language.

Motion-engine cards require that the PC provide the language while the card provides motion instructions for tasks such as trapezoidal profiling, servo loop closure, and so on. Most of these cards implement some sort of packet-oriented commands.

Programming these cards is easy once you are familiar with the commands. To load and execute a trapezoidal profile for axis 1 on a PMD MC1401A chip-based motion-engine card, you need to program:

```
SET_1
SET_POS 12345 ; set final-destination position
SET_VEL 3344 ; set maximum velocity
SET_ACC 456 ; set acceleration value
UPDATE ; make the move
```

The next example loads servo parameters and makes them active (the parameters currently being used) for axis 4.

```
SET_4
SET_KP 123
SET_KD 1234
SET_KI 100
UPDATE
```

Motion-engine cards provide far more capabilities than just these examples show. Nevertheless, the basic concept of sending the card motion instructions, which are buffered by the card and then executed, is at the heart of all motion-engine type cards.

Which language approach is best for you depends on your application and your control architecture. Motion-engine cards enable you to construct elaborate control systems of which the motion card may just be one component. Motion-language cards are adept at offloading chunks of the control problem, so the PC can focus on other tasks.

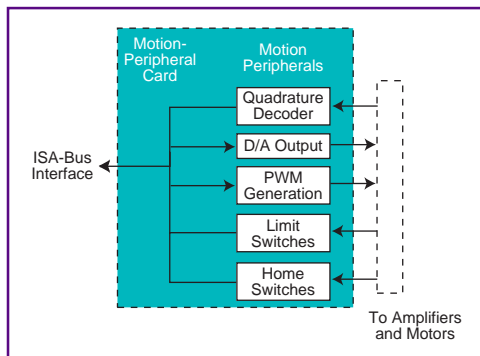
Figure 4—Motion-peripheral cards provide access to low-level motion hardware. The PC must perform high-speed trajectory and servo algorithm execution.

engine or program microprocessor on the card at all. There are only motion peripherals such as incremental encoder feedback, analog output, digital I/O, and so forth.

In some ways, motion-peripheral cards aren't motion cards at all. They implement no motion-specific language or instruction set.

What they do offer is low-cost flexibility. As long as the user is willing to write the software required to perform trajectory generation, servo loop closure, and other typical motion tasks, motion-peripheral cards are a viable solution.

Unfortunately, in a typical PC-Windows environment, providing the low latency responses required for servo control and other typical motion tasks is daunting. The programmer must be a wizard at interrupt management. Typical servo-loop rates for



motion language or motion cards are between 1000 and 5000 Hz.

Properly scheduling the servo calculation on the PC at this speed is important. Otherwise, motion jitter may result. PC operating-system software was never designed to work with these low latencies and system-access subroutines (e.g., hard-drive access, screen updates, etc.). Therefore, it must be managed so that it does not affect the low-level motion code.

Another potential drawback of this technique is the safety concern associated with a potential PC-application crash. With a motion-language or motion-engine card, this concern doesn't arise because there

are enough brains on the card to safely handle the system even if the PC stops sending commands.

MOTION-PERIPHERAL APPLICATIONS

Despite these challenges, there has been some recent interest in this type of card, particularly for "canned" motion applications like CNC or contouring. Because the control problem has been worked out by the software vendor, it's worthwhile to check out this interesting, low-cost solution.

For other applications where the user must write the software, this type of card isn't a good match. When you use a motion-peripheral card, a short 10-line program for making a move on a motion engine becomes a complex exercise in algorithm development and timing management on the PC.

The ultimate solution in the direction of reduced card complexity is to use no physical card at all and instead use the PC's parallel port to generate motor command signals. This kind of configuration is commonly performed with step motors, which use digital pulse and direction signals to drive the motor amplifier.

DISTRIBUTED CONTROL

Figures 5a and 5b overview centralized- and distributed-control schemes, respectively, as applied to motion control. The centralized scheme represents the historical method of achieving motion control. In fact, all three board types I just described fall into this category.

The distributed approach organizes the control problem so that a motion-control card is not needed in the PC. In this scheme, the PC talks to each distributed motion module and provides instructions, which the module autonomously carries out. A standard network card is used in the PC instead of a dedicated motion card.

Similar to PC-based motion-control cards, distributed modules come in motion-language and motion-engine varieties. In the context of distributed control, motion-language modules are usually referred to as stand-alone controllers and motion-engine modules are called distributed or network-based controllers.

The primary advantage of distributed control is the reduction in wiring. This helps reduce both the installation cost and the service cost once the product is in the field.

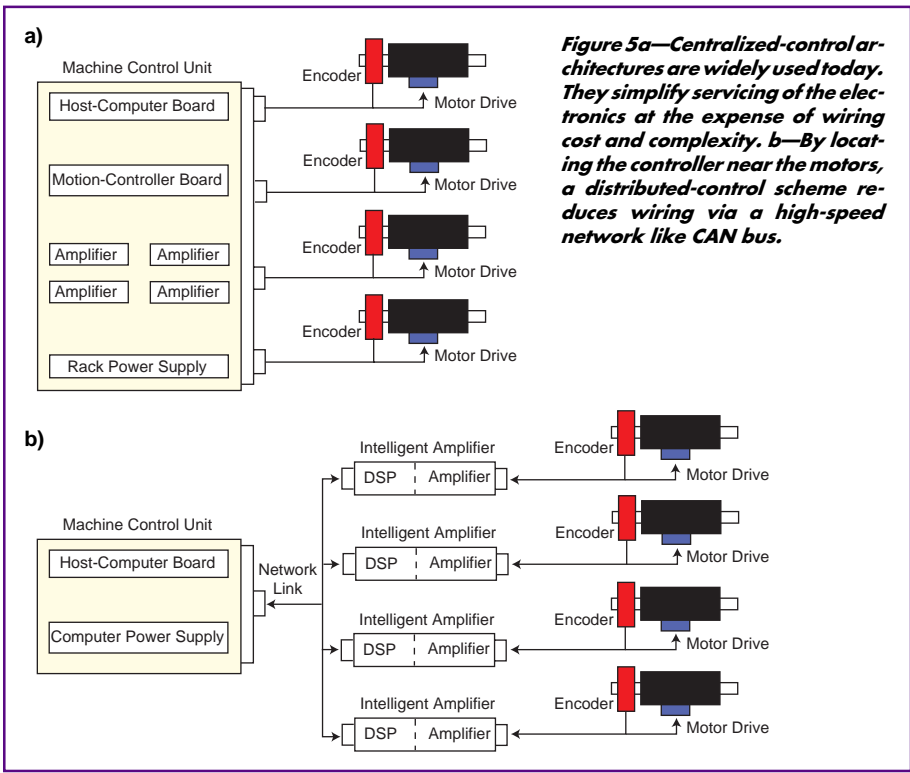


Figure 5a—Centralized-control architectures are widely used today. They simplify servicing of the electronics at the expense of wiring cost and complexity. b—By locating the controller near the motors, a distributed-control scheme reduces wiring via a high-speed network like CAN bus.

As for servo applications, the only canned software packages that work in this mode are for CNC and contouring. If

you're looking for the ultimate in low hardware cost, you may want to consider this approach.

Another big advantage is increased flexibility. Most motion cards implement the same type of motor control (e.g., all step-motor control or all DC servo control). In a distributed scheme, it's easy to mix and match motor types since each module is generally single axis.

In particular, large systems with 8, 12, or more axes benefit from distributed control. Large systems often require a variety of motor types. As for wiring, as long as the distributed motion module is located close to or even physically attached to the motor, cabling is greatly reduced and the MTBF of the system improves.

Distributed control is in its infancy and there are relatively few standards, vendors, or installed systems. One popular distributed motion protocol known as SERCOS implements a motion-engine approach.

However, SERCOS is a very low-level motion-engine approach. In this distributed system, which is popular with CNC vendors, each module can execute only a short motion vector and a central host performs all high-level path planning.

A major problem with SERCOS for general-purpose applications, however, is that in many ways, it is fundamentally a centralized system with a distributed approach to the wiring only. SERCOS cannot be controlled without a motion card because the modules can only perform short vector moves of a few milliseconds' duration.

True distributed control, where modules can share information peer-to-peer and where a substantial amount of intelligence resides in the module, has been implemented here and there by specific vendors, but no standards exist.

Along these lines, several data network protocols are available for the PC. In motion control, CAN bus is leading the pack because of its low cost and high level of acceptance in other industries. CAN bus is fast enough to perform synchronized moves for a modest number of axes and flexible enough to support devices like motion-control modules, digital I/O, and analog input all on the same network bus.

SOFTWARE RULES THE DAY

Motion-card vendors are notorious for claiming the highest servo-loop rate, the fastest update time, and the most complex profiles. For most users of motion technol-

| Card | Description | Advantages | Disadvantages |
|---------------------|--|--|--|
| Motion Language | provides ability to download motion code into card and execute autonomously without PC interaction | popular easy to program stand-alone operation | more expensive vendor-specific language |
| Motion Engine | provides motion instruction set for profiling, servo, etc. PC provides "language" facilities and sends motion commands to card | low cost easy to program standard language easy synchronization | can't run stand alone |
| Motion Peripheral | provides peripherals only such as encoder feedback and analog signal output. PC provides all motion software, including profiling, servo control, etc. | lowest cost | not easy to program safety questionable if PC crashes |
| Distributed Control | control system located in intelligent modules close to motor. Reduces wiring and eliminates motion card in PC | low wiring cost flexible more reliable scalable | requires better standards few motion vendors |

Table 1—The motion card you choose depends on the structure of your software and other design considerations.

ogy, these features have little meaning. Electronics have become so powerful that most popular motion cards provide more features than you really need.

A bigger issue for many designers is ease of use. And, ease of use means good software.

In the context of motion control, software has two meanings. The first is the software that you must write to develop your application. Motion-language cards provide a dedicated vendor-specific language, and motion-engine cards use BASIC or C as the language and provide callable libraries to access the motion card's capabilities (see the sidebar "Programming for PC-based Motion Control").

Sometimes overlooked is the software that most vendors provide for setup, servo tuning, and profile selection. Often called "exercisors," these packages can be run out of the box and enable the user to interact with screens and menus to control the motion card.

Another important software item is libraries. When you buy the card, will you get the source code? Does it cost extra? Does the vendor provide DOS-, Windows 95- and Windows NT-compatible libraries? If not, you may have a hard time transitioning your application to a new platform.

Be careful when selecting a motion-control card vendor, and don't be overimpressed with all those claims of megacounts, kilohertz, and pulse rates. Sometimes what really counts is ease of use!

WEIGH THE OPTIONS

Table 1 summarizes the pros and cons of the different PC-based motion-control systems I've discussed.

The PC has gained widespread acceptance because of its low cost, flexibility, and ease of use. And now, motion-control users can take advantage of its popularity to build their machine controller with a PC as the core processor.

The type of motion card and motion architecture that's best for a given application depends on the nature of your control system and software. In the end, ease of use and cost determine which approach is best for you. [EPC](#)

Chuck Lewin is president of Performance Motion Devices. He has been working in motion control for the past eight years and designing DSP-based motion systems for the past five years. He has written articles for various engineering magazines, providing practical, application-oriented advice on the implementation of motion-control systems. You may reach Chuck at lewin@pmdcorp.com.

SOURCE MC1451A

Performance Motion Devices, Inc.
12 Waltham St.
Lexington, MA 02173
(781) 674-9860
Fax: (781) 674-9861
www.pmdcorp.com

IRS

413 Very Useful
414 Moderately Useful
415 Not Useful

Ingo Cyliax

Software Development for RTOSs

Reusability is critical to software development. So, this month, Ingo shows us how to port programs written in our favorite programming languages into two popular RTOSs. He then debugs the code with target- and host-based debuggers.

Last month, I looked at the issues behind selecting an RTOS. My sample application needed an RTOS to generate precise timing signals for the actuators in a six-legged robot, while at the same time requiring soft real-time processes to run in the system.

This month, I discuss the typical software development environments encountered when building embedded-PC applications. I use some of the sample code presented last month to show how I ported it to two popular RTOSs—QNX OS and Phar Lap's ETS Realtime.

What do we need to develop code for an embedded real-time application? Most RTOSs provide libraries for the API, which are linked with your application. So, you need a compiler, which compiles the code into an object to be linked with your RTOS library, and a linker, which links your compiled code with the RTOS libraries.

Let's take a look at some programming languages used to develop real-time applications for embedded PCs.

REAL-TIME PROGRAMMING LANGUAGES

The most common development language for real-time PC development is C. As you already know, C was developed originally to implement Unix in a portable way and has now been around for quite some time.

C has been a favorite with embedded-systems developers, too. While not as efficient as coding in assembler language, C is portable and enables the programmer to access memory and I/O port resources.

C compilers exist for almost any micro-processor from Cray supercomputers to 64- and 32-bit processors like Pentium and PowerPC, all the way down to PICs. Well, OK, I haven't seen one for a DEC PDP-8.

While it's possible to use assembly in all PC-based RTOSs, it is probably wise to avoid coding in assembly unless absolutely necessary. Most modern C compilers are good at optimizing code for particular architectures (i.e., i486 vs. Pentium).

It is almost never necessary to code in assembly language for speed. Besides, unless your application is very high volume, the increased time and costs associated with developing in assembly are almost always higher than just going with a faster processor.

For example, it may take 80% of the development effort to squeeze that last 20% of speed from an application by coding up critical routines in assembly language and hand-optimizing them. Speed increases achieved via a faster CPU and bigger, faster cache architecture are usually cheaper.

About the only time you want to code in assembly is when you'd like to implement low-latency interrupt service routines, or stubs for ISRs, which let you set up the environment necessary to call C routines. Luckily, many RTOS vendors have done a good job providing these stubs as part of their API.

C++ has been popular in desktop systems for quite some time now and is starting to become more popular in embedded systems. While there is nothing magic about using C++, tool support for C++ in development systems for RTOSs has been slow in coming. However, many RTOS vendors currently have tool and library support for C++.

However, just because you code your applications in C++ doesn't mean you're necessarily using object-oriented programming (OOP) methodology. You can also do OOP in C or assembly language. C++ is just a tool.

Another OOP language getting a lot of press lately is Java, developed by Sun Microsystems as a new network programming technology.

The Java programming language can be used to develop code, which gets compiled into byte code to be executed in an interpreter on a target system. It promises architecture-independent application development.

Several RTOS vendors offer Java support, but there are still some issues to be worked out. In particular, the Java run time relies on garbage collection, which in its current reference implementation from Sun is nondeterministic and thus not suitable for hard real-time applications. Work is being done on this, but more about Java in a later column.

FORTTRAN is still used in embedded systems. In particular, there are numerical libraries which are coded in FORTRAN. These libraries represent much time in development, testing, and tuning.

Since it's harder to find a FORTRAN compiler that can be hosted on current development platforms, some libraries are being converted to C. One solution is to use FORTRAN-to-C converters, which you can find on the Internet.

Unless you code for the government or systems that need to be flight qualified, you probably won't see Ada. Ada is good for implementing large systems where many

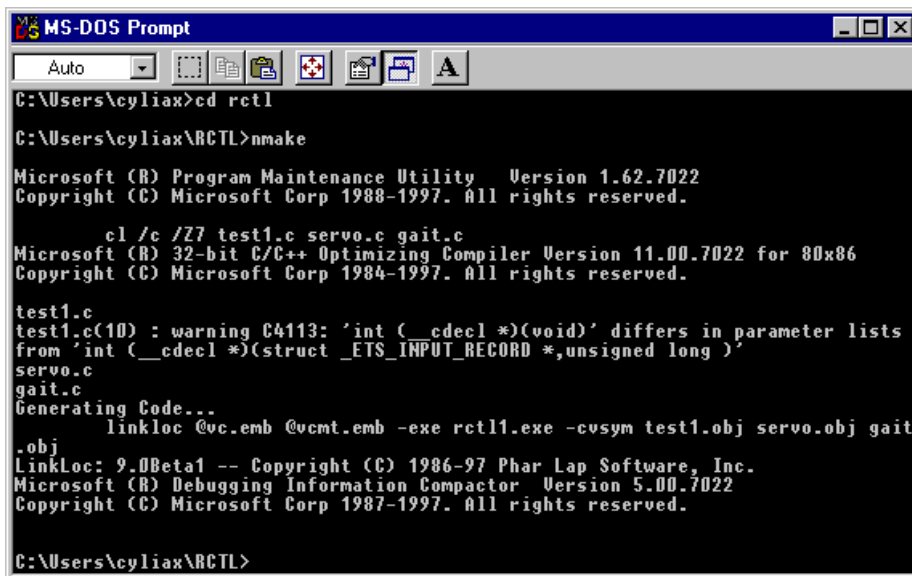


Photo 1—In this screen shot, you see a command-line-driven compilation and linking session under Windows. I'd probably make sure the warning didn't occur in a production version of my program.

developers are working in different organizations, but it's sometimes difficult to use for expressing real-time issues.

That's not to say that there aren't some RTOSs for embedded systems which are written in Ada. For example, RTEMS, which is developed by the US Army and

distributed for free, is available in Ada or C source.

Also, there's trend towards using high-level behavioral modeling language for systems. In these systems, the modeling language describes what the system components do. Typically, hardware

Listing 1a—Here's the main routine for Phar Lap's ETS. You program the timer using `EtsSetTimerperiod()` and install the servo interrupt service routine (ISR) as a timer call back using `EtsRegisterCallback()` from the ETS application programming interface. b—In QNX, setting up the ISR is done much like in ETS. `clock_setres()` sets the timer period, and `qnx_hint_attach()` attaches an ISR to the timer call-back chain. Since QNX is an operating system, which persists after we get done running our program, we want to make sure to remove our ISR when done.

```

a)
#include "global.h"
#include "servo.h"
main(){
    int Servo_Isr();
    EtsSetTimerPeriod(1);
    EtsRegisterCallback((WORD)ETS_CB_TIMER,
        &Servo_Isr, 0, ETS_CB_ADD);
    GaitThread();}

b)
#include "global.h"
#include "servo.h"
main(){
    pid_t far Servo_Isr();
    int id,i;
    struct timespec st;
    st.tv_sec = 0;
    st.tv_nsec = 500000;
    clock_setres(CLOCK_REALTIME, &st);
    if((id = qnx_hint_attach( 0, &Servo_Isr,
        FP_SEG(&SetTime[0]))) == -1 ){
        printf("can't attach interrupt\n");
        exit(2);}
    GaitThread();
    qnx_hint_detach(id);}

```

components modeled in these systems are then synthesized into hardware-description languages like VHDL or Verilog, and software components with languages like Ada or C, which are then compiled using standard compilers.

A REAL-TIME EXAMPLE

So, enough theory. Let's look at a real example. Remember the RC-servo-based robot controller I told you about last month for the six-legged Stiquito II robots? If you recall, I presented some sample code to illustrate the idea of interrupt latency.

To illustrate two kinds software development environments—target and host based—I ported the interrupt-based servo driver and gait generator to two popular RTOSs, Phar Lap ETS and QNX. While the original example also included a network-based command interpreter, I'll save it for another column.

I replaced the main module with a new module, `test1.c` (Listings 1a–b), which does the necessary initialization and starts `GaitThread()`, which is implemented in module `gait.c` (see Listing 2).

`GaitThread()` is a high-level process, which computes the necessary patterns for the legs actuators. The RC-servo-based driver is implemented using an interrupt service routine in `gait.c`. Just like `test1.c`, ISRs are RTOS specific, so I have two versions, which are given in Listings 3a and 3b.

MOVING SOFTWARE

Software development for ETS is done using a Windows NT or 95 host. Since ETS is Win32 compatible, we can actually use the same 32-bit C and C++ compilers to develop code for ETS as we do for our regular Windows software development.

In particular, I used Visual C++ V.5.0 (VC), which is part of Visual Studio for Windows 95 and NT. I used my notebook, which runs Windows 95, as the development host.

Since I'm more comfortable using command-line-oriented tools, I will use the command-line version of VC to build my code. If you're more familiar with Visual Studio, use it instead. In V.9.0 of ETS, Phar Lap added support for building applications using VC under Visual Studio.

To start off, I developed a short `Makefile`, given in Listing 4a. Here I call `cl`, which is the command-line interface to compile and link using VC. By using the `/C`

Listing 2—The gait-thread implementation is the same for both OS environments. Here, I use macros to map generic constructs like `GetMutex()` and `Sleep()` to OS-specific calls. Using C macros is one common technique for making C source code portable between different OS environments.

```
#include "global.h"
#include "servo.h"
#include "gait.h"
int Command; /* current command */
int LastCommand;
int cmdmutex;
int MaxSteps[nCMD] = {4,4,4,4}; /* number of steps in pattern */
/* gait patterns */
int Pattern[nCMD][nSTEP][nCHAN]=
{
  {{2,4,2,4,2,4,2,4},{4,2,4,2,4,2,4,2},{2,4,2,4,2,4,2,4},{4,2,4,2,4,2,4,2}},
  {{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2}},
  {{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2}},
  {{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2},{2,2,2,2,2,2,2,2}}
};
/* generate leg actuations depending on current command */
GaitThread(){
  int CurrStep;
  int i;
  GetMutex(cmdmutex);
  Command = CMD_STOP;
  LastCommand = Command;
  ReleaseMutex(cmdmutex);
  CurrStep = 0;
  while(1){
    Sleep(STEPTIME);
    GetMutex(cmdmutex); /* check whether command mode changed */
    if(LastCommand != Command) CurrStep = 0;
    LastCommand = Command;
    ReleaseMutex(cmdmutex);
    for(i=0;i<nCHAN;i++){
      SetTime[i] = Pattern[LastCommand][CurrStep][i]-1;
      /* set servo channels */
    }
    CurrStep = (CurrStep + 1) % MaxSteps[LastCommand];
    /* next step */
  }
}
```

switch, I instruct `cl` to only compile each source module.

Once the modules are compiled into object modules (`test1.obj`, `gait.obj`, and `servo.obj`), `linkloc` links the modules, using the required ETS libraries into an executable image `rct11.exe`. `Linkloc` also generates the symbol table for the debugger, which is loaded in the executable. A sample compile and link run in a command-line window under Windows 95 produces the output seen in Photo 1.

Once we have the executable, we need to get it to the target system. Phar Lap provides a monitor and kernel, which can be booted from a floppy, called `diskkern.bin`. It can be configured to load the application either from disk or remotely over serial or parallel port.

In my example, I connected the target system to my notebook via a serial connection. The target system was another notebook—a 25-MHz 386SX-based notebook I keep around for just this purpose.

I use the target's floppy drive to boot `diskkern.bin` configured to load the application from the serial port. Once the target has loaded the monitor from floppy, I can use the command `-com 1 rct11.exe` to download and start executing my program.

`GaitThread()` simply cycles eight servo channels, one on each of the parallel ports data pins between full extensions of the actuator. I verified the timing with an oscilloscope, which also gave me a good indication of the jitter, which was minimal.

Now, doing the same thing for QNX, where we compile the program running on the OS, is a bit different. QNX can be configured as a full-featured RTOS.

Once installed on hard disk, QNX boots and presents a login screen. After supplying the login and password needed to get into the system, we get a command-line prompt. (QNX also has a windowing system, Photon. For now, I'm just going to focus on the command-line interface.)

The command-line interface in QNX is Unix-like, which is not surprising, since many software developers (me included) are familiar with Unix and feel mostly at home using arcane commands like `vi` and `ls`.

Also, QNX offers TCP/IP support, including remote login facilities for telnet, rlogin, and file-transfer facilities (e.g., FTP and rcp). In fact, once you have a QNX machine fully configured, it's much like having a Unix machine or server.

There is, however, one major difference between Unix and QNX. QNX is a hard real-time OS. It has deterministic timing behavior for interrupt latency and operating system calls and implements preemptive multipriority scheduling. It also presents us with a rich set of process synchronization and communication mechanisms.

QNX is also a network distributed OS. Nodes can be used for remote execution of processes. It's probably overkill for this example, but it does a good job of illustrating the idea of host-based development.

Since QNX is so Unix like, I had to adapt the makefile I used for ETS (see

Listing 4b). Here I use `cc`, the usual name for a Unix-based C-compiler/linker front end, and instruct it to simply compile and link all modules and produce an output binary `rct11` with the `-o` flag. I also specified the `-g` flag to generate an extensive symbol table, to be used with the source-level debugger available under QNX.

Since QNX supports remote logins via TCP/IP, I can simply use telnet from my notebook to log in to the system and build my application using `make`. I use a wireless LAN PC Card in my notebook, so I can connect to the Ethernet in my house.

The target system is a 66-MHz 486DX2-based systems, with a small 120-MB hard disk that hosts QNX OS and is wired into my Ethernet using an NE2000-compatible Ethernet card. Being able to log in to my target to develop real-time applications from my notebook anywhere in my house or on the Internet is quite nice.

Photo 2 shows what a session looks like. Once the application is built, the executable becomes one of the commands of the system. Simply by executing `./rct11`,

Listing 3a—To make the servo ISR behave well with other ISRs at the same interrupt level, Phar Lap's servo ISR returns `ETS_CB_Continue` to indicate that other ISRs chained to the same interrupt should run as well. b—In the QNX version of the ISR, we need to use a compiler directive `#pragma off (check_stack)` to disable run-time stack checking since the ISR runs on its own interrupt stack and not the normal user stack.

```
a)
#include "global.h"
#include "servo.h"
volatile int SetTime[nCHAN]; /* value for timer */
volatile int Ticks; /* value for current output */
volatile int CurrChan; /* current channel */
volatile int servomutex;
int Servo_Isr(ETS_INPUT_RECORD *r, DWORD dummy){
    /* only do something when we run out of ticks */
    if(!Ticks--){
        setport(SERVO_PORT, (1<<CurrChan));
        Ticks = SetTime[CurrChan++];
        CurrChan %= nCHAN;}
    return(ETS_CB_CONTINUE);}

b)
#include "global.h"
#include "servo.h"
volatile int SetTime[nCHAN]; /* value for timer */
volatile int Ticks; /* value for current output */
volatile int CurrChan; /* current channel */
volatile int servomutex;
#pragma off (check_stack)
pid_t far Servo_Isr(){
    /* only do something when we run out of ticks */
    if(!Ticks--){
        setport(SERVO_PORT, (1<<CurrChan));
        Ticks = SetTime[CurrChan++];
        CurrChan %= nCHAN;}
    return(0);}
#pragma on (check_stack)
```

```

Telnet - tmp2
Connect Edit Terminal Help
Welcome to QNX 4.24
Copyright (c) QNX Software Systems Ltd. 1982,1997
login: root
Last login: Thu Jan 01 18:35:12 1998 on //1/dev/tty0
Thu Jan 01 18:47:21 1998
# cd tmp
# make
cc -g -o rctl1 test1.c servo.c gait.c
/usr/watcom/10.6/bin/wcc386 -zq -d2 -ms -4r -i=/usr/watcom/10.6/usr/include -i=/
usr/include test1.c
/usr/watcom/10.6/bin/wcc386 -zq -d2 -ms -4r -i=/usr/watcom/10.6/usr/include -i=/
usr/include servo.c
/usr/watcom/10.6/bin/wcc386 -zq -d2 -ms -4r -i=/usr/watcom/10.6/usr/include -i=/
usr/include gait.c
/usr/watcom/10.6/bin/wlink op quiet form qnx flat na rctl1 op priv=3 op c libp /
usr/watcom/10.6/usr/lib:/usr/lib: de all f test1.o f servo.o f gait.o op offset=
40k op st=32k
# ./rctl1
█

```

Photo 2—Here's a telnet session to a target-based development system. Don't be fooled into thinking this is a generic Unix machine. The user interface is similar, but when the program, rctl1, executes, it runs in a real-time environment and is able to generate 1-2-ms pulses with $\pm 50\text{-}\mu\text{s}$ jitter.

I can generate waveforms under real-time control.

Since QNX's timing resolution goes down to 500 μs , I can generate the timing pulses with 0.5-ms resolution. And, I can achieve all this while remote users log in and compile applications.

DEBUGGING TECHNIQUES

Development systems for real-time applications usually have debuggers available for application code. These debuggers vary from being simple monitors, which examine or modify memory and control execution of the application, or sophisticated source-level debuggers.

Source-level debuggers let you correlate the instructions and data in the application

with the source code from which it was built. You can set breakpoints by simply locating a specific line in the source code module, and examine the variables.

Debuggers, just like the development system in general, can be target or host based. Target-based debuggers run on the target in conjunction with the RTOS and the application to be debugged. It communicates with the system or, if the RTOS supports it, via a network connection.

Target-based debuggers usually have the best performance since they run on the same machine, which reduces the communication latency for reading and writing memory. Tracing execution flow of an application, where the program is single stepped showing what instructions are being

Listing 4a—Here's the makefile for compiling the program using Visual C++ in command line mode and linking it with Phar Lap's linkloc linker. b—In contrast, the QNX makefile looks like a Unix makefile, calling cc to both compile and link the program.

```

a) # Makefile for Phar Lap ETS
RCTL1=test1.c servo.c gait.c
RCTL1HDR=global.h servo.h gait.h
rctl1.exe: $(RCTL1) $(RCTL1HDR)
    cl /c /Z7 $(RCTL1)
    linkloc @vc.emb -exe rctl1.exe -cvsym \
    test1.obj servo.obj gait.obj -cvsym

b) # Makefile for QNX
RCTL1=test1.c servo.c gait.c
RCTL1HDR=global.h servo.h gait.h
rctl1: $(RCTL1) $(RCTL1HDR)
    cc -o rctl1 $(RCTL1)

```

executed, is also fastest using the target-based debugger, since it has access to the software interrupts necessary to implement this.

Sometimes when running target-based debuggers, we have to deal with a reduced level of functionality for the user interface. For example, when I run a target-based debugger on QNX over telnet, I may only have access to a basic character-based interface.

Host-based (or remote) debuggers have two components. The actual debugger, which runs on the host, manages the user interface and reads and writes the symbol table. It communicates with the target, which runs a monitor, using a communication channel. Usually, this is a serial interface or a parallel interface with the target. Photo 3 shows a screen shot of `codeview`, a remote debugger used with Phar Lap's ETS.

If you remember, in the ETS example, I used `diskkern.bin`, which was booted from the floppy disk. With `diskkern.bin`, the command-line program `runemb` was used to download and run the application to the target.

When we want to use the debugger, we simply use `cvemb`, which also lets us download the image. Once it's downloaded, we can control our application and examine memory/registers using the debugger running on the development host.

In this environment, the debugger is also capable of attaching to an already running application. This makes it possible to debug real-time applications running from a boot ROM. I'll explain more about how to run the application from boot ROM a little later.

One thing to keep in mind—when we debug a real-time application, the application may not perform in real time once the program stops at breakpoint or when we trace or single step through the application. In our servo example, the system simply stops and the scope displays flatlines.

To debug real-time applications running in real time, you need hardware-based debugging facilities like an in-circuit emulator (ICE). With an ICE, you can replace the CPU in the embedded system with a system that emulates the CPU running in real time.

ICE systems are usually attached to a development host through a serial or parallel interface, but some are network based. The user interface on the development host is

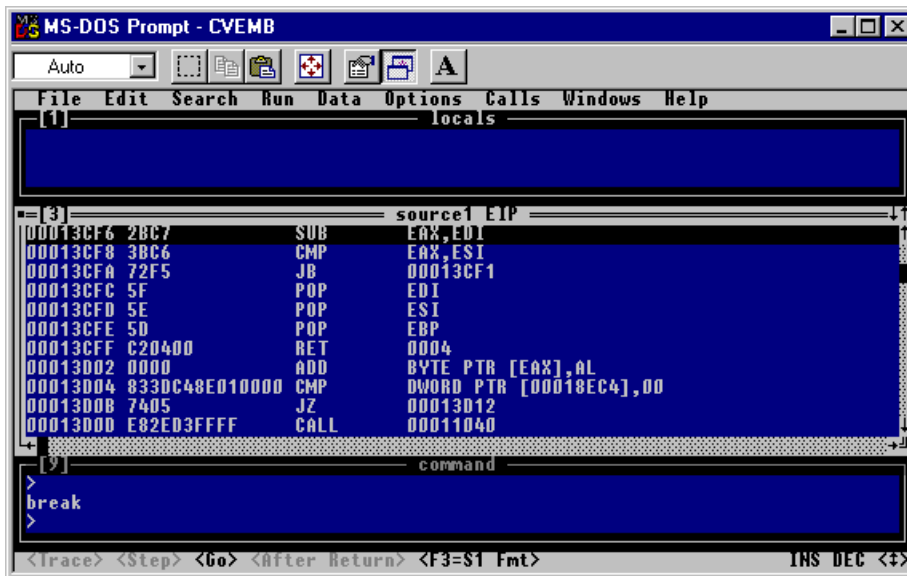


Photo 3—In this typical debugging session to an RTOS target, the debugger has several windows: one to trace execution flow in assembly language, one to trace variables, and a command window. Even though it's not shown here, the debugger can also trace program flow in the C source code.

usually the same as software-based debuggers and has about the same functionality of downloading code and examining memory and registers. However, ICE-based debuggers allow real-time tracing of the target.

WRAP UP

So, software development for RTOSs is not that different from writing code for your favorite desktop system. In some cases, you can even use the same development environment you'd use to develop Windows applications.

Other systems let the developer log in to their RTOS system just like a full-featured operating system and develop there. Either way, it's usually easy to set up a development environment for the RTOS you want to use.

Also, by doing the software development in C and isolating OS-dependent API features in a single module or using macros defined in header files, it's possible to write applications that are relatively portable between RTOS architectures.

To learn more about writing software for a particular RTOS, check out the documentation. It's usually full of example code illustrating how to use the specific APIs. Also many RTOSs provide on-line examples, which you can compile and run to illustrate certain features about an RTOS's API.

Now, you've got the software down. Next month, we'll move on to take a look at some GUIs used in real-time embedded PCs. **RPC/EPC**

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

REFERENCES

- RTOS FAQs, www.realtime-info.be/encyc/market/rtos/rtos.htm
- V. Toth, *Visual C++ 5.0: Unleashed*, SAMS, Carmel, IN, 1997.

SOURCES

QNX OS/Watcom C/C++ Compiler for QNX
 QNX Software Systems Ltd.
 175 Terence Mathews Cres.
 Kanata, ON
 Canada K2M 1W8
 (613) 591-0931
 Fax: (613) 591-3579
 info@qnx.com
 www.qnx.com

ETS Realtime

Phar Lap Software
 60 Aberdeen Ave.
 Cambridge, MA 02138
 (617) 661-1510
 Fax: (617) 876-2972
 www.pharlap.com

IRS

- 416 Very Useful
- 417 Moderately Useful
- 418 Not Useful

Applied PCs

Fred Eady

Embedding PC Card

Part 2: Getting in Touch

You've got a service console with no ports for a printer, display, keyboard, or mouse. How do you talk to this thing? Using diagnostic code, a PC Card, and a touchscreen controller, Fred puts together an interface that's just what you need.

Last time, we examined the innards of PC Card technology. Hopefully, you saw how beneficial PC Cards can be to embedded solutions.

Although some of you may design PC Cards, most of us won't be building PC Cards for our embedded solutions. Instead, we usually procure PC Cards for embedded applications as the need arises.

With that in mind, let's take a look at an embedded touchscreen application that employs a PC Card coupled with an embedded PC.

TOUCHING ON THE PROBLEM

Many times, the applications we write aren't doing the real work. Sometimes, it's necessary to implement code that supports the overall embedded solution but that isn't part of the main program or application. In this case, the supporting code is diagnostic code intended to help the field engineer identify and repair peripheral hardware connected to the core embedded system.

In a standard environment, this diagnostic code can be loaded from existing magnetic media and manipulated via keyboard or mouse. In an embedded environment, the keyboard and mouse may not exist. There may not be any spinning disks in the area, either.

Let's assume that we know these extra goodies don't exist and we can design an embedded system with diagnostic capability that matches the limits or absence of hardware in the final embedded hardware suite.

The first question to be answered is how to interface the technician to the hardware. If external test equipment is required and no special hardware ports exist to connect the test equipment, the usual route is to plug stuff into existing serial or parallel ports on the embedded PC.

But what if the available ports are all being used? If the embedded PC needs to talk to remote devices, at least one of the serial ports is most likely an interface to a modem.

It's also possible that the embedded application may need to provide printed output. A printer could use either the second serial port or tie up the standard parallel port.

If I designed the application, the parallel port would probably be doing unnatural things in the I/O world, and if print was desired, I'd use the remaining serial port. Depending on the location of the embedded devices, there may not be

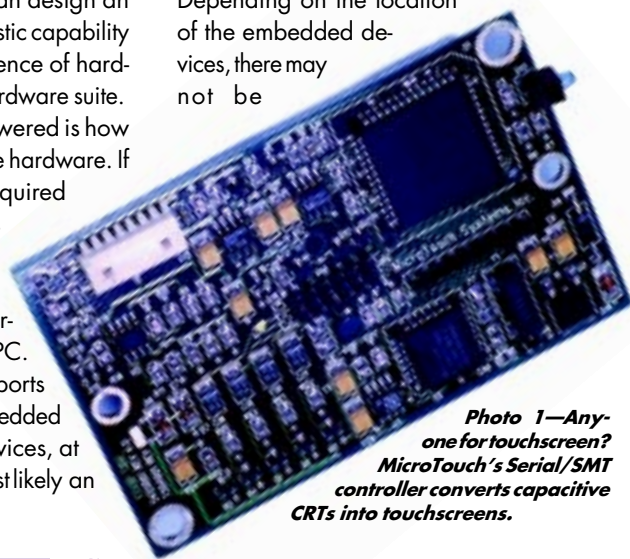


Photo 1—Anyone for touchscreen? MicroTouch's Serial/SMT controller converts capacitive CRTs into touchscreens.

enough service area to attach and use a standard PC keyboard or mouse.

Many embedded applications don't require a display, but in this case, we need a service console.

Assuming this solution isn't one of a kind, it would be impractical to have the field technician carry a display from site to site.

Depending on the application software, you may need a display anyway. Either way, the display ends up being the only user interface, so we must use it to its fullest capacity functionally and diagnostically.

OK. So far, it's clear. The embedded design will most likely use all of the available standard I/O and there is but one user interface port—the display.

Last time, I talked about how PC Cards could be thrown into an embedded system to enhance their usability. To provide suitable diagnostic capability, this embedded solution needs extra I/O, which can be provided by including PC Cards in the initial design. Using a PC Card and the appropriate driver, we can now effect almost any type of I/O interface our diagnostic routines require.

It stands to reason that if we can generate any type of I/O interface we need by simply plugging in a PC Card, we can use any piece of test equipment to troubleshoot problems with the embedded system. That's great if you have an unlimited budget. Good test equipment costs lots of money. And, if your product is in the field, there's probably more than one technician servicing it. More money.

The good news—you probably have all the test equipment you need in your embedded suite. All you have to do is write code to enable it. External test equipment is necessary if the problem lies in the embedded hardware itself, but the testing and exercising of peripheral equipment connected to the embedded I/O ports can be tested with user-written routines that exist within the embedded PC's firmware.

| Controller | Technology | Mounting |
|---------------|------------|-----------------------------|
| Serial/SMT2 | Capacitive | External or internal |
| Serial/SMT3 | Capacitive | External or internal |
| Serial/SMT3V | Capacitive | External or internal |
| Serial/SMT3R | Resistive | External or internal |
| Serial/SMT3RV | Resistive | External or internal |
| Serial/SMT2 | Capacitive | On CPU board |
| Daughterboard | | |
| Serial/SMT3V | Capacitive | On CPU board |
| Daughterboard | | |
| PC Bus SMT2 | Capacitive | In 16-bit PC expansion slot |
| PC Bus SMT3V | Capacitive | In 16-bit PC expansion slot |
| PC Bus SMT3RV | Resistive | In 16-bit PC expansion slot |
| TouchPen 4 | Capacitive | Internal |
| | Digitizer | |
| TouchPen 4+ | Capacitive | Internal |
| | Digitizer | |
| MousePort | Capacitive | External or internal |
| Chipsets | Capacitive | Integrated into the design |
| | Resistive | of your system board |

Table 1—Looks like if you can see it, you can touch it.

Within the boundaries of our embedded system, we know that the only user diagnostic port is the display because no keyboard, mouse, or external test equipment can be attached. Other than talking to the display (like Scotty did when cooking up some transparent aluminum), there's only one other means of manipulating diagnostic routines via CRT—touch.

REACH OUT AND MICROTOUCH

A company called MicroTouch produces the touchscreen technology I'll use in this diagnostic application. MicroTouch specializes in the conversion of standard CRTs and flat-panel displays to touchscreens. MicroTouch offers controllers for both capacitive and resistive touchscreens.

Because this solution is embedded and specific, I'm not going to consider most of the controller configurations you see in Table 1, but I thought you'd like to know all the possibilities. Table 1 is the entire list of MicroTouch touchscreen controllers, including name, technology supported, and mounting options. Let's take a look at each one.

The Serial/SMT controllers are RS-232 serial controllers. The controller can be internally mounted in a standard monitor or enclosed in a molded plastic case, which is typically mounted to the back or side of the monitor. This is the configuration I'll use, and the touchscreen controller is mounted inside the CRT.

The daughterboard controller is a CMOS serial add-on board that can be mounted onto an embedded CPU board. This option is normally taken when you wish to integrate

Listing 1—It really is this simple to initialize the touchscreen controller. Impressive, huh!

| | |
|------------------|----------------|
| Reset | <SOH>R<CR> |
| AutoBaud Disable | <SOH>AD<CR> |
| Parameter Set | <SOH>PN812<CR> |
| Format Decimal | <SOH>FD<CR> |
| Mode Stream | <SOH>MS<CR> |

the daughterboard controller onto a system board you're designing from scratch.

The PC Bus controller is a half-slot bus card that is installed in systems capable of accepting standard PC-bus cards. If the embedded PC for this solution was ISA-slot capable, this option would be a good choice.

The PC Bus controller has its own serial communications port, which enables you to use existing embedded COM ports for external peripherals. The touchscreen cable connects to the port on the controller.

The TouchPen controller offers the same features as the Serial/SMT capacitive controller, with the addition of pen support. The controller can accept touch input from a finger or touch pen.

This RS-232 serial controller is designed to easily fit inside flat-panel displays and CRTs. Unlike the Serial/SMT controller shown in Photo 1, which can be mounted internally or externally, the TouchPen controller is always mounted internally.

The MousePort controller has an attached 8', six-pin mini-DIN PS/2 connector. This option is worth considering because you can connect this controller to a PS/2 mouse port, leaving extra serial communication and bus slots available for peripherals.

Chipsets are available if you want to integrate a MicroTouch touchscreen controller directly into your own circuitry. These chipsets include an optimized controller circuit that can be used with a MicroTouch capacitive or five-wire resistive touchscreen.

That about does it for the hardware I'm writing diagnostic code for. The application calls for PC Card capability and a touchscreen diagnostic interface. And because I am going to use MicroTouch touchscreen controllers, I also know the PC Card interface must be RS-232.

So, I'll select a serial PC Card with the necessary drivers to implement the touchscreen serial interface. As you can see, using PC Card technology has taken much of the complexity out of the design, enabling us to concentrate on the application at hand—building a touch-based diagnostic interface.

KEEP IN TOUCH

The diagnostic application consists of placing targets on the touchscreen that correspond to diagnostic routines. Once a target is touched, the diagnostic software is notified by the touchscreen controller and the touched diagnostic routine is kicked off. To effect this, it's pretty obvious that the software must communicate with the touchscreen controller. Let's see how that's done.

Commands to the touchscreen controller are provided via the diagnostic software on the Receive Data (RXD) signal pin as a serial datastream. This software must also be capable of receiving responses from the touchscreen controller.

Touchscreen-controller responses are data sent from the controller to the embedded host system in response to the commands

received by the touchscreen controller. These responses to the embedded host system are provided on the transmit data (TXD) signal pin.

To successfully send a command to the controller, it's imperative that you use the correct command format. The general format is broken into three parts—header, command, and terminator.

The header is the first character in the command string and is the ASCII start-of-header character <SOH>. The ASCII <SOH> character is equivalent to 01 hexadecimal.

The command, which always follows the header, consists of ASCII uppercase letters and numbers. The terminator, the last character of each command string, is an ASCII carriage return <CR>, which is equivalent to 0D hexadecimal.

Thus, a standard MicroTouch touchscreen controller command looks like <SOH>Command<CR>. After executing a command, the controller returns a response or acknowledgment to the embedded host system.

Similar to the outbound commands, each touchscreen-controller response consists of a header, command response, and termi-

nator in the same format as the outbound touchscreen-controller command that emanates from the embedded host system.

The header and terminator in the response string are identical to their counterparts in the outbound command sequence. The response, which always follows the header, is a little different. It's a range of ASCII characters that depend on the type of command received.

Responses can be in many forms. An example of a standard response is <SOH>0<CR> (ASCII character "0" or 30 hexadecimal), which indicates successful command completion. When this response is returned to the embedded host system, the touchscreen controller received a valid command and executed the command properly.

On the dark side of that, <SOH>1<CR> (ASCII character "1" or 31 hexadecimal) indicates that the command failed. When this response is returned, the controller received an invalid command and did not execute the command. If this happens, usually the command wasn't formatted correctly, system parameters weren't set up for

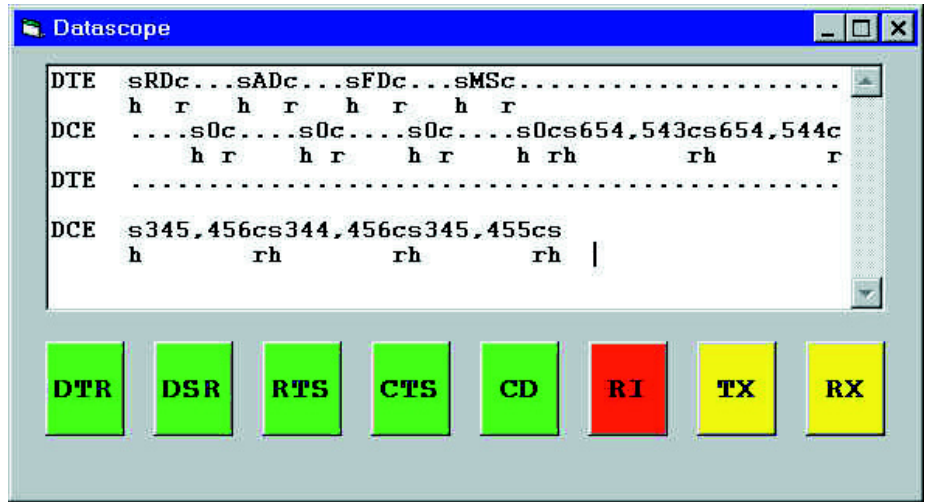


Photo 2—Here's a good example of Mode Stream. Notice the datastream is continuous when I hold my finger in one place.

command execution, or the touchscreen controller doesn't support the command.

You now have enough background on the embedded hardware and the MicroTouch touchscreen controller to visualize the diagnostic application software routines. All we have to do is send the appropriate commands to the touchscreen controller and interpret the responses. Depending on

the received response, we initiate particular diagnostic routines. With that, let's examine the MicroTouch command set shown in Table 2.

Take a look at a command sequence for initializing a touchscreen controller (see Listing 1). The first command—Reset—initializes the touchscreen-controller hardware and firmware. On receiving this command, the controller stops sending data and recalculates environmental conditions.

Reset also cancels the Format Raw and Calibrate Raw commands and returns the controller to normal operation. Reset should be issued whenever the embedded host system is powered on and is attempting to establish communication with the touchscreen controller.

Depending on the controller, the amount of time needed to execute Reset ranges from 225 to 800 ms. Therefore, we must code the diagnostic application program to wait and then be sure it receives a positive command response before issuing another command to the touchscreen controller.

The AutoBaud Disable command follows Reset. AutoBaud Disable turns off the automatic data-rate-detection feature. When AutoBaud is disabled, the touchscreen controller maintains the communication rate currently set in nonvolatile RAM (NVRAM). The touchscreen controller continues to use this communication rate until it is changed by the Parameter Set or AutoBaud Enable command.

Once AutoBaud is disabled, the logical thing to do is set the desired communications parameters. Parameter Set lets you adjust the communication parameters

(parity, data bits, and stop bits) of the touchscreen controller.

You can also change the communication rate by appending a character to the command string. On execution of `Parameter Set`, the controller automatically stores the new settings, current operating mode, and current data format in NVRAM.

Just one gotcha. The communication parameters of the embedded host system must match the present settings of the touchscreen controller when the command is given for it to be accepted and the changes implemented. Otherwise, we're spittin' into the wind.

Thus, the process of changing the parameters implies that our embedded host system must first communicate with the touchscreen controller using a matched set of parameters. Once `Parameter Set` is issued with new parameters to the touchscreen controller, the new settings immediately take effect.

At this point, our embedded host must be changed to the new parameters to talk with the touchscreen controller again. If we don't follow these rules, we could wind up

in the ditch with no way to communicate with the touchscreen controller.

The good news—there's a way out of the ditch. MicroTouch provides a diagnostic tool called `Microcal` that could be used to winch us out. You can download `Microcal` from the MicroTouch Web site.

`Parameter Set` is pretty straightforward. Its command syntax is:

```
<SOH>Ppds[b]<CR>
```

where p is parity type (N is no parity, O is odd, and E is even), d equals the number of data bits (7 or 8), s is the number of stop bits (either 1 or 2), and b stands for the communication rate. The communication rate is expressed by 1 equaling 19,200 bps, 2 as 9600 bps, 3 as 4800 bps, 4 as 2400 bps, and 5 as 1200 bps.

The next command causes the touchscreen controller to output the x,y touch coordinate data as a nine-byte packet in a decimal format. The packet consists of nine bytes arranged as:

| Command Name | ASCII Code | SMT3, SMT3RV | | | |
|------------------------|-------------|-------------------------|------------------------------|-----------------------|-------------------------|
| | | SMT2 PC Bus SMT2 | PC Bus SMT3RV PC Bus SMT3V | SMT3, SMT3R MousePort | TouchPen 4 TouchPen 4+ |
| Default Settings | | N72, 9600 AE, FD, MS | N72, 9600 (AD/AE), FC, MS | N81, 9600 FT, MS | N81, 9600 FT, MS, PF |
| AutoBaud Disable | AD | X | | | |
| AutoBaud Enable | AE | X | | | |
| Calibrate Extended | CX | X | X | X | X |
| Calibrate Interactive | CI | X | X | | |
| Calibrate New | CN | X | X | | |
| Filter Number | FNnn | X | X | | |
| Finger Only | FO | | | | X |
| Format Binary (Stream) | FB(S) | X | X | | |
| Format Decimal | FD | X | X | | |
| Format Hexadecimal | FH | X | X | | |
| Format Raw | FR | X | X | X | X |
| Format Tablet | FT | X | X | X | X |
| Format Zone | FZ | X | X | | |
| Frequency Adjust | <Ctrl C>Fnn | X | | | |
| Get Parameter Block | GPn | X | X | X | X |
| Mode Down/Up | MDU | X | X | | |
| Mode Inactive | MI | X | X | | |
| Mode Point | MP | X | X | | |
| Mode Polled | MQ | X | X | | |
| Mode Status | MT | X | X | | |
| Mode Stream | MS | X | X | X | X |
| Null Command | Z | X | X | X | X |
| Output Identity | OI | X | X | X | X |
| Output Status | OS | X | X | | |
| Parameter Lock | PL | X | X | | |
| Parameter Set | Ppds(b) | X | X | | |
| Pen Only | PO | | | | X |
| Pen or Finger | PF | | | | X |
| Reset | R | X | X | X | X |
| Restore Defaults | RD | X | X | X | X |
| Sensitivity Set | SEn | X | X | | |
| Set Parameter Block | SPn | X | X | X | X |
| Unit Type | UT | | | X | X |
| Unit Type Verify | UV | | X | | |

Table 2—Wow! This command set is powerful and easy to remember.

- one header byte
- three bytes of *x*-coordinate data
- an ASCII comma
- three bytes of *y*-coordinate data
- a terminator byte

Data is sent as a string of decimal ASCII characters (0–9). The output range for the *x* and *y* data is 000–999.

When activated, `Format Decimal` resets the `Mode Status` to report the standard `<SOH>` header. Resetting to the standard header implies that `Format Decimal` does not contain touchdown and liftoff information like `Format Tablet`.

To obtain this type of information, the `Mode Status` command is issued:

```
<HDR>Xxx, Yyy<CR>
```

Let me go through each command. `<HDR>` is the start-of-header marker (hex 01). If you send a `Mode Status` command after a `Format Decimal` command, this first byte becomes a status byte. The status byte defines whether the *x,y* coordinates are generated from a touchdown, a touch continuation (when the finger is resting on the screen), or a touch liftoff.

Xxx stands for the *x* (horizontal) coordinate data. It has a total of three bytes. Then, there's an ASCII comma separating the *x* data from the *y* (vertical) coordinate data, which also has a total of three bytes. Finally, `<CR>` is the terminator (hex 0D).

The final command in our init string is `Mode Stream`, which instructs the touchscreen controller to send a continuous stream of *x,y* coordinate data on touch. The controller continues to send data as long as the user touches the screen, even if the touch is stationary and unchanging.

I've pretty much touched on every aspect of enabling the diagnostic software, so let's put it together and assemble the application.

TOUCHING UP

You know me. The first thing I connected to the touchscreen was an RS-232 datascop. I wanted to see the bits flow.

After playing, I decided I'd rather perform some commands and capture the results on the datascop—basically, emulate the diagnostic program's outbound command sequence—than describe bit flow. To effect this emulation, I put the datascop in RS-232 monitor mode and tapped into the datastream flowing be-

tween the MicroTouch touchscreen and the PC Card-enabled embedded PC.

Photo 2 is the result. The DTE side of the trace is outbound data from the PC Card serial port, and DCE data is from the touchscreen controller.

I said earlier that a reset sequence should be done on powerup. Well, look closely. At the beginning of the trace, the DTE issues `<sh>RD<cr>`, which translates to `<start of header>Restore Defaults<carriage return>`. `Restore Defaults` copies the MicroTouch factory-default parameters from ROM to the NVRAM and then executes `Reset`.

Notice the right side of the DCE trace. The touchscreen controller gives a positive acknowledgement with `<sh>0<cr>`. As you scroll right, you see three more DTE commands, followed by controller ACKs.

The fun begins just after the `Mode Stream` command. That's when I touched the screen. Just like the MicroTouch tech manual says, I got nine bytes of position info starting with `<sh>` and ending with `<cr>`. As you move through the trace, you can see that I moved my finger and then held it there.

TOUCH AND GO

Now you have it all with a means of getting position data from a MicroTouch touchscreen via a PC Card serial port. All that's left to do is to place some targets on the touchscreen and poll for coordinates tied to particular diagnostic routines.

Once again, we've proven that it doesn't have to be complicated to be embedded.

APC.EPC

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

SOURCE

Touchscreen controllers
 MicroTouch Systems, Inc.
 300 Griffin Brook Park Dr.
 Methuen, MA 01844
 (978) 694-9900
 Fax: (978) 659-9100
 www.microtouch.com

IRS

419 Very Useful
 420 Moderately Useful
 421 Not Useful

DEPARTMENTS

68

MicroSeries

74

From the Bench

82

Silicon Update

EMI Gone Technical

MICRO SERIES

Joe DiBartolomeo

Protection Components

Part
3
of
4

Last month, Joe gave us the low-

down on common suppression devices.

This time, he preps us on zener diodes, TVS thyristors, TVS diodes, and a whole lot more. Come catch the performance differences.



So far in this MicroSeries, I've given you a look at the most common causes of electrical transients affecting electronic equipment and systems. These threats include lightning, electrical fast transients/bursts, and electrostatic discharge.

I've shown you their waveforms and specifications, which provided a starting point for designing protection for circuits and systems. After all, the first step in protection is to understand what you're protecting against.

Last month, I discussed two classes of protection components—crowbar and clamp. Arcing devices are of the crowbar class, but MOVs are of the clamping class. And of course, both classes have advantages and disadvantages.

In this installment, I continue by looking at more components used to protect against transient threats—the ubiquitous zener diode, TVS thyristors, TVS diodes, and positive temperature coefficient resistors. I also compare the performance of zener diodes to TVS diodes and MOVs. And, I present a specification table of the devices I've talked about.

Of course, my list of transient protection devices is by no means complete. There are many more transient-suppression components. I'm simply attempting to present the most commonly used and easily obtainable devices.

ZENER DIODES

The inherent characteristics of zener diodes make them popular as transient suppressors. Used within their limits, zener diodes provide good protection against some transient threats.

Zener diodes have a high impedance in the off state. When conducting (i.e., in the on state), they clamp the shunt voltage to a maximum zener voltage, V_z , by dramatically reducing their impedance. Although zener diodes are directional, they can be placed back to back as shown in Figure 1 to give a bidirectional VI characteristic curve.

Zener diodes are available in low clamping voltages, well below 5 V. Their transition from the off state to the clamping or on state is relatively fast and hard. This is in contrast to MOVs, which have a soft transition to the on state. Zener diodes turn on in only a few nanoseconds. Unlike MOVs, zener diodes do not degrade when used within their ratings.

Because of these characteristics, zener diodes are one of the few components suitable for protecting individual ICs and I/O lines, particularly for low-voltage applications. However, like all clamping devices, zener diodes exhibit capacitance and draw leakage current in the off state. Depending on your application, this could be a problem.

Since zener diodes are clamping devices, they must absorb the transient pulse energy and dissipate it as heat. Recall from last month, I mentioned that a MOV dissipates the transient throughout its total area, whereas a zener diode must dissipate the transient energy at its junction.

A special class of zener diodes designed to absorb short-duration high-energy pulses is also available. To increase their power-handling capability, these zener diodes have large junction areas.

Like any suppression device, zener diodes exhibit overshoot. This is normally due to a transient with a dv/dt too fast for the zener diode to follow or due to lead inductance that delays the turn on of the zener diode.

Since zener diodes have such fast turn-on times, they can follow most transients. However, zener diodes have problems with electrostatic discharge

(ESD). The rising edge of an ESD waveform as set out in the test specifications is less than 1 ns.

When subjected to an ESD, zener diodes exhibit enough overshoot to cause concern. I'll take a more careful look at this when I compare zener diodes and TVS diodes.

When a zener diode dissipates a transient, its junction temperature increases. There is a finite time required for the junction temperature to return to its steady-state value.

If a zener diode were subjected to repetitive surges, there would be a surge-repetition frequency at which the zener diode could not dissipate the heat from the previous surge before the next surge arrived. This raises the zener diode's temperature above the steady-state value, thereby reducing its maximum power-handling capability.

This is not unique to zener diodes. MOVs and in fact all other clamping devices exhibit this behavior, although it's somewhat more pronounced in zener diodes. I discuss this phenomenon further in the TVS-diode section of this article.

TVS TERMINOLOGY

The term "transient voltage suppressor" can be applied to any of the protection devices we have seen so far.

However, TVS normally refers to a specialized class of zener diodes or thyristors. TVS diodes and thyristors are devices whose characteristics are tailored to the suppression of voltage transients.

TVS THYRISTORS

A TVS thyristor is a monolithic device consisting of an SCR-type thyristor whose gate region contains a specially diffused region that behaves like a zener diode. The combination of the zener diode's fast turn on and the

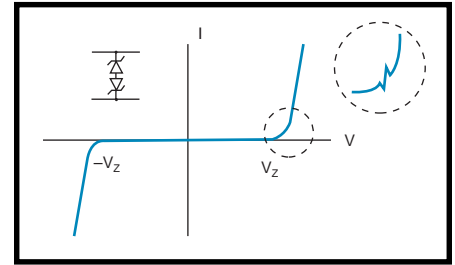


Figure 1—By placing two zener diodes back to back, we get a bidirectional VI curve. The transition region has some jagged edges, which delays the turn-on time. But since zeners are relatively fast devices, this characteristic normally isn't a problem.

thyristor's current-handling capability makes the TVS thyristor a unique and useful protection device.

The construction of the TVS thyristor is shown in Figure 2. Note the N-P-N-P construction and the zener-diode gate region. The VI curve of the TVS thyristor is shown in Figure 3. The thyristor's VI curve is similar to other crowbar devices except that the TVS thyristors has a zener voltage, V_{zener} .

The TVS thyristor is normally used across the line to be protected and ground. As long as the voltage across the device does not exceed V_{zener} , the device is in the off or high impedance state. When the voltage across the TVS thyristor exceeds the avalanche breakdown voltage, the zener diode clamps the voltage at V_{zener} . This accounts for the TVS thyristor's fast turn on, (e.g., low-nanosecond range).

Once the zener diode starts conducting, current flows into the gate of the thyristor and causes the thyristor to begin conducting. When the thyristor is fully conducting, the device is in the low-voltage region of its VI curve.

In effect, the thyristor once turned on by the zener diode diverts current away from the zener diode, thereby giving the device higher current capabilities than a lone zener diode would have. Once the surge has passed, the current in the TVS thyristor must fall below the holding current, I_{Hold} , before the device returns to its high impedance state.

TVS thyristors can be unidirectional or bidirectional with voltage ratings of 25–270 V. TVS thyristors do not degrade with applied transients as MOVs do.

Their fast turn on means that they have low overshoot, unlike the large overshoot associated with gas tubes. When using TVS thyristors or any crow-

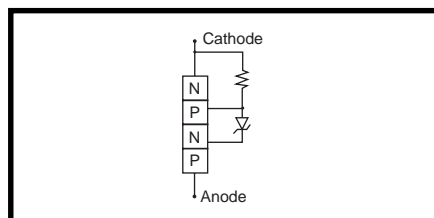


Figure 2—Here's how a TVS thyristor surge suppressor is constructed.

bar devices in DC circuits, ensure that the short circuit current is less than I_{Hold} . Otherwise, the device won't turn off. The holding current of TVS thyristors is fairly high (e.g., 130-mA range).

Since TVS thyristors are crowbar devices, they have better current-handling capabilities than MOVs but not as good as gas tubes. When TVS thyristors fail, they usually fail as shorts. Consider fusing them in some manner.

TVS thyristors have relatively low shunt capacitance and low leakage currents when in the off state. They are used mainly for telecommunications applications.

TVS DIODES

TVS diodes are often referred to as silicon avalanche suppressors, and they are produced by several manufacturers. Basically, they're very large junction zener diodes specifically designed for transient suppression. The most attractive characteristic of these devices is their extremely fast turn-on time (e.g., subnanosecond range).

Avalanche breakdown occurs in picoseconds, but due to lead inductance and test equipment limitations, it's difficult to quote actual picosecond turn-on time. The extremely fast response of TVS diodes means there's very little voltage overshoot in comparison to other suppression devices.

The clamping voltage, V_c , for TVS diodes ranges from 3 to 400 V, making them ideal for protecting individual ICs. Since they are clamping devices, they do not have high-current handling capabilities, but at the individual IC level, this is not an issue.

Unlike MOVs, TVS diodes do not degrade when subjected to transients. TVS diodes come in unidirectional or bidirection configurations. The IV curve for a typical TVS diode is quite similar to that of the zener diode.

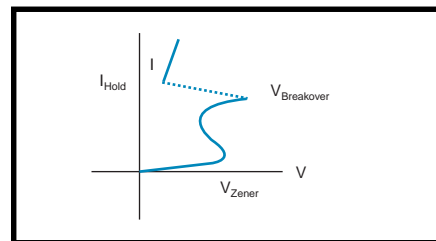


Figure 3—The VI curve of a TVS thyristor is similar to other crowbar devices. However, the TVS thyristor turns on faster due to the zener-diode action.

Like most other protection devices, TVS diodes are normally employed across the line to be protected and ground. The TVS diode, like the TVS thyristor, has low shunt capacitance and low leakage currents when in the nonconducting state.

When a surge across the devices causes the TVS diode voltage to reach its breakdown voltage, the TVS diode clamps to V_c almost instantaneously (i.e., in less than a nanosecond). Any overvoltage is normally due to lead inductance.

Power ratings for TVS diodes range up to 5 kW on a 10/1000 surge and up to 400 W on a 8/20 surge. These power ratings are derived from the product of the peak voltage across the device and the peak current conducted through the device.

If TVS diodes are subject to pulses other than 8/20 or 10/1000, their power ratings must be adjusted. Figure 4 shows a typical device pulse power rating versus pulse width graph for a TVS diode rated for 1000 W on a 10/1000 waveform.

As would be expected when the TVS diode is subjected to a 10/1000- μ s pulse, it can dissipate 1000 W of transient pulse energy. If the pulse width is greater than 1000 μ s, the power rating of the TVS diode is decreased. Likewise, if the pulse width is less than 1000 μ s, the power rating of the TVS diode is increased.

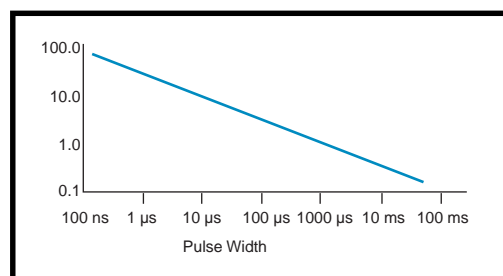


Figure 4—When TVS diodes are subject to pulse widths other than the 10/1000 μ s, their power ratings must be adjusted. This graph shows a typical derating for a 1000-W TVS diode. At 1000 μ s, the TVS handles a peak pulse power of 1000 W (its rating). As the pulse width increases, the power rating decreases, and as the pulse width decreases, the power rating increases.

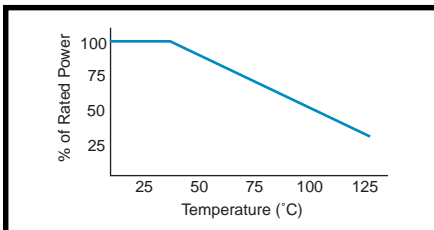


Figure 5—This graph shows a typical percent of power versus operating temperature curve for a TVS diode. Most TVS diode power ratings are specified at 25°C. If the TVS diode is operated at temperatures above 25°C, its power rating must be derated.

This behavior can be seen in all other clamping devices. Since peak pulse power is peak current multiplied by the clamping voltage (which is constant after overshoot), the maximum peak current the device can handle varies directly with the applied transient pulse width.

TVS diodes, like clamping devices, dissipate the transient as heat and transfer that heat to the ambient environment. Therefore, ambient temperature determines the amount of power a TVS diode can dissipate. The higher the ambient temperature, the lower the power rating, as illustrated in Figure 5.

The rating for peak pulse power is given for a single pulse, but what happens if the TVS diode is subjected to repetitive pulses such as electrical fast transients/bursts (EFT/B)?

The key is whether or not the repetitive pulses raise the device's temperature, thereby requiring derating as shown in Figure 5. For example, consider a TVS diode with a clamping voltage (V_c) of 10 V subjected to a pulse train with a pulse width of 10 μ s, pulse amplitude of 100 A (I_p), and pulse period of 10 ms.

Peak pulse power (P_{pp}) is the clamping voltage multiplied by the peak current:

$$\begin{aligned} P_{pp} &= V_c \times I_p \\ &= 100 \text{ A} \times 10 \text{ V} \\ &= 1000 \text{ W} \end{aligned}$$

The average power (P_{avg}) would be the peak pulse power times the ratio of the on and off times of the pulse train:

$$\begin{aligned} P_{avg} &= P_{pp} \left(\frac{T_{on}}{T_{off}} \right) \\ &= 1000 \left(\frac{10 \mu\text{s}}{10 \text{ms}} \right) \\ &= 1 \text{ W} \end{aligned}$$

A TVS diode with a steady-state power rating of 1 W or higher is able to handle the repetitive surges without having its power specification derated.

TVS diodes are the fastest suppression components and therefore exhibit the lowest overshoot of any suppression components. TVS diodes also have low leakage current and low capacitance in the off state. The combination of these factors makes TVS diodes the device of choice when protecting individual IC and low-level I/O lines.

ZENER VS. TVS DIODES

As we have seen, the transient suppression devices that have a hard and fast transition from the off to the on state rely on the zener/avalanche phenomena. TVS diodes are a special class of zener diodes.

However, the significant difference between zener diodes and TVS diodes is in their junctions' ability to dissipate heat. TVS diodes are specifically designed for very fast heat dissipation at their PN junctions.

On the other hand, zener diodes are designed for voltage regulation, which does not require fast heat dissipation at their junctions.

Recall that both devices are clamping devices that must dissipate transient energy as heat. The faster the heat transfers out of the PN junction, the larger the surge current the device can handle and still maintain a low junction impedance, thereby reducing overshoot. This is extremely important when protecting circuits from ESD.

When we are dealing with transients, the peak pulse power is important—not the average power-handling ability. For this reason, it is possible for a 1-W TVS diode to provide better protection than a large 50-W zener. By the time the heat even begins to travel through

the package of the ordinary zener diode, the transient may be long gone.

An experiment to test this was performed using a 6-V zener diode and a 6.8-V TVS diode. Both devices were subjected to a simulated 5.568-kV ESD waveform.

The voltage across the zener diode reached 144 V before it clamped. As Lee points out, the TVS diode clamped when the voltage across it was only 33.5 V.

Since most ICs can withstand a momentary ESD of ~200 V, both devices provided protection. However, a person walking on carpet can easily build up a charge greater than 20 kV, which is much more than the 5.586 kV used in the test and much too large for the zener to provide protection.

POSITIVE TEMPERATURE COEFFICIENT RESISTORS

I would like to briefly mention one last component—the positive temperature coefficient (PTC) resistor. A PTC resistor has low resistance at room temperature.

As the current through the PTC resistor increases, its temperature increases. When the device's temperature reaches the Curie temperature (between 50°F and 120°F), the resistance of the PTC resistor increases dramatically.

When the current through the PTC resistor is reduced, the temperature returns to normal and the resistance of the PTC resistor returns to its low level. PTC resistors are essentially self-resetting fuses. These devices are intended for applications where a fuse would be a nuisance.

Be careful, however, because the PTC resistors may not return to their original resistance after they cool down. This could be a problem in sensitive analog circuits.

| | Turn-On Time | Leakage Current Off | Capacitance Off | Voltage Clamping | Current On |
|---------------|--------------|-----------------------|-----------------|------------------|------------|
| MOV | 50 ns | 5–250 μ A | 10–60,000 pF | 14–1200 V | 4 A–60 kA |
| TVS diode | <1 ns | 0.5–10 μ A | 10–10,000 pF | 3–440 V | up to 50A |
| TVS thyristor | Few ns | 50 nA | 50 pF | 25–270 V | >3 kA |
| Zener diode | Few ns | 1–1000 μ A | 100 pF | 3–275 V | up to 200A |
| Gas tube | 100 ns | 1×10^{-20} A | 1–5pF | 0.1–<10kV | >20 kA |

Table 1—In order to select transient-suppression components, you need to compare their critical specifications. Note there is always a tradeoff to be made when selecting components.

Using a PTC resistor as a transient suppressor is quite simple. An incoming transient increases the voltage across the PTC resistor, which leads to an increase in both the current through and the temperature of the PTC resistor.

When the PTC resistor's temperature reaches the Curie temperature, the dramatic increase in the PTC resistor's resistance protects against the incoming transient.

The problem with using PTC resistors for transient suppression is that they are far too slow to protect against transients. By the time the device's temperature increases to the Curie temperature, the transient is long gone. In fact, with ESD, the transient is very fast, but the energy content—its ability to heat—is relatively small.

Table 1 gives a summary of the protection components are most commonly used to protect equipment and circuits from transient induced EMI.

There is always a tradeoff made when you select a component. Your protection scheme therefore normally requires several of these components placed at different points in the circuit and/or equipment. You wouldn't use a coarse protection device such as a gas tube to protect an IC. Nor would you expect that a TVS diode could handle a lightning strike.

Also important are the off-state parameters of the transient-suppression devices. Some parameters are more important depending on the type of circuit or system you need to protect.

For example, when you need to protect digital lines, the leakage current of the device isn't a major concern. However, you do need to think about the device capacitance and lead inductance. They tend to round the edges of the digital signals.

If you're protecting an A/D input, the leakage current of the device in the off state may cause an expensive 16-bit ADC to be an expensive 10-bit ADC.

Table 1 offers a general guide. The values given are only typical. Also, review datasheets carefully to determine the test conditions.

For example, the device junction capacitance varies with the applied reverse bias. The greater the reverse bias, the less the capacitance. The

device capacitance also varies with the test frequency, which ranges from 1 kHz to 1 MHz. And given that:

$$X_c = \frac{1}{2\pi fc}$$

then there is a 1000× difference.

For current and power ratings, be sure you understand the test waveforms used to determine the ratings. Was an 8/20, 100/1000, or another pulse used?

PUTTING THEM TOGETHER

When designing for transient protection, it's important to consider each device's characteristics. Next month, I'll look at protection scheme that takes advantage of the strengths of each suppression component while accounting for their weaknesses. 📧

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

- T. Armstrong, "Surge protection for mobile communications," *Electronic Design*, p. 79, January/February 1997.
- G. Dash and I. Straus, "Designing for power line surge immunity," *Compliance Eng.*, p. 235, 1993.
- General Instruments, Power semiconductor division, Product catalog, 1994.
- GE Solid State, *Transient voltage suppressors Selector Guide*, TSS-426, 7, 1987.
- B. Lee, "Can Zener diodes provide ESD protection?" *Compliance Eng.*, p.85, July/August 1997.
- Motorola, TVS/Zener device data-book, 1991.
- Motorola, TVS/Zener device data-sheet DL150 Q3/91, 1991.

I R S

- 422 Very Useful
- 423 Moderately Useful
- 424 Not Useful

Rebirth of the Z8

FROM THE BENCH

Jeff Bachiochi

Part 1: An Old Friend Comes To Visit



It's been a while since Jeff got up close

and personal with Zilog's Z8 micros. He almost forgot how comfortable this old friend can be. Part 1 brings us up to date with the Z8 family developments.



lost my jacket. Well, that's not exactly true. I still know where it is. It's just

that the outside temperature while I'm writing this has once again reached the mid-40s and I don't need it. Even though it's now midwinter, the snow has melted, the lake has thawed, and it feels like spring.

I tried to continue running indoors on a treadmill this winter because the weather outside made our roads unsafe, but I haven't been able to keep up the enthusiasm. It's just not the same as running outdoors.

Getting dressed this morning, however, I noticed my running shoes quietly resting right where I left them after my last run, but this morning was different.

Like the trees who are thinking about budding and the geese who keep circling wondering whether it's now time to fly north, I've got my mind on warmer weather. (Of course, it has nothing to do with the fact that I weighed myself twice this morning thinking the scale was in error.)

Slipping on those running shoes felt good. You know, like old friends. Like that old shirt hanging in the closet—you wonder why you haven't tossed out until you put it on and remember just how comfortable it is.

We all keep things from the past just because of the comfort they bring

us. It doesn't have to be clothing. It might be a knickknack, baseball cap, motorcycle phone, or some old photo of you back home. Point is, if you've had a pleasant experience in the past, the feeling is easily rekindled by a simple object or thought.

YOU NEVER FORGET YOUR FIRST

My first experience with computers was so distracting that I became its slave to unusually late hours. Only stopping for a quick catnap, I must have perfected a dozen routines that first night. I was hooked.

You couldn't separate me from my first computer. It took quite a while before I got the urge to open it up and get inside. What's a Z80? It meant nothing at the time.

Zilog was started by some Intel employees with an alternate vision. I thank them for the years of pleasure they gave me with the gray box. The open architecture provided many adventures. And, as you know, I chose to let microcontrollers take up much of my life.

Steve had begun designing with microcontrollers (called microcomputers back then) in the early '80s. Again, the name Zilog appeared and I immediately had this warm fuzzy feeling. This time it was a Z8. Over that year, I cut a few new teeth on the Z8.

Some of you might remember an article I did back in *INK* 36 on the Z8 ("Breathing New Life Into an Old Friend—Revisiting the Z8"). Having come across some early 2- and 4-KB Z8 circuitry, I explored how it still fit into the larger applications appearing at the time.

It's been only in the last few years that designers have been willing to look hard at the benefits of smaller solutions—other than the obvious fact that smaller pieces of silicon cost less. Now, manufacturers have begun to sit up and take notice of a steadily increasing movement of designers toward the use of smaller "picocontrollers" (i.e., controllers that don't require the typical external code and data space).

Recently, Digi-Key started distributing Zilog parts. What's even more interesting is that these new OTP parts have affordable development tools.

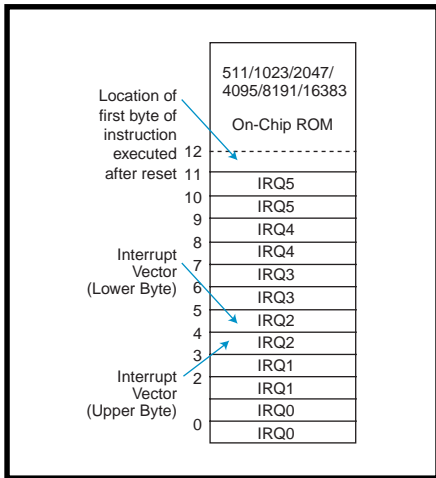


Figure 1—Zilog's OTP cores begin at ½ KB and presently extend to 16 KB of internal code space.

Are these Z8 parts new? Truth is, my preliminary datasheets of the Z86C08 (an 18-pin part), for example, are dated April 1988.

Although OTP is new to Zilog, the Z8 architecture has been around a long time. The problem with the original 18-pin parts is that they were available as ROM parts only. You had to have a rather big project before you would consider working with the part.

Now, however, OTP parts let us code a project using a small Z8 without incurring the masking costs and volume buys necessary, making it financially feasible to use masked parts.

BOTTOM UP VS. TOP DOWN

When chip designers today have no product base, they are free to design from the bottom up. Because they use the latest technologies, innovative designs naturally capture every new user's attention.

On the other hand, chip designers with an already well-established product line can easily manipulate existing winners into new forms to take advantage of a market's most recent direction. This top-down approach is comfortable for all involved. There's little education necessary for that core audience.

When I consider new parts to work with, I want to know two things—one, that there are (small quantity) parts available, and two, that development tools are priced reasonably.

Well, someone at Zilog has seen the light. They're hitting hard via product exposure in new markets as

well as the one closest to home—the Internet.

NOT NEW, JUST DIFFERENT

For those of you already familiar with the Z8, you can pass Go and wait by your mailbox for next month's column while I spend some time this month laying down a Z8 foundation.

Figure 1 shows the memory map of the Z8 microcontroller. Since the program counter is 16 bit, a full 64 KB of code space is available, although only 0.5–16-KB OTP parts are available now and can be programmed with the low-cost emulator.

My point here is the family uses only one core and path with all parts using the same 46 instructions. The program or code space begins at 0000H with six predetermined two-byte interrupt vectors.

The location of the first byte of the instruction to be executed after reset is 000CH. (Larger parts like the Zilog 8671 with BASIC/Debug masked into the first 2 KB use external code and data spaces requiring external address latch and memory devices. This setup requires two 8-bit ports for external addressing. The smaller OTP parts use only internal code space and therefore don't need any of the I/O ports for a multiplexed address/data path.)

Internal register space consists of 256 consecutive bytes, which include up to 239 general-purpose RAM registers, up to four 8-bit I/O port registers, and up to 16 control and status registers.

Unlike other processors where you have a single accumulator or data pointer, any of the Z8 general-purpose registers can function as accumulators or address or index pointers. These registers can work as 8 bit or in pairs as 16-bit registers.

Another option is to treat groups of 16 registers as banks allowing shorter instructions that work within a 4-bit address. Figure 2 shows the Z8's internal register organization.

Figure 2—The Z8 general-purpose registers contain I/O ports, RAM, and control registers.

Stack operation uses as little or as much of the general-purpose RAM as the user wishes to set aside. Although registers 254 and 255 are used as the 16-bit stack pointer, only 255 is needed when you use the internal general-purpose registers for the stack (it's an 8-bit address).

Stack operations can be two-byte addresses, as with a CALL, one-byte values, as with a PUSH/POP operation, or three bytes, as with an interrupt (which saves both the return address and the FLAGS register). The stack grows downward.

Most bits within the port I/O registers can be configured as both inputs and outputs. Many have alternate functions such as analog comparator, interrupt or counter/timer inputs, or handshaking I/O.

Z8 counter/timers are eight-bit programmable. They can be driven externally or by the internal clock and its six-bit prescaler. The counters can start, stop, and be set to autoreload the initial value. Each counter/timer can run in single pass or continuous mode.

The internal clock source for T1 can be externally retriggerable, nonretriggerable, or gated. Reading the counter/timer registers does not disturb their value or count.

| Dec | Hex | Identifier |
|-----------------|---------------------------|------------|
| 255 | FF | SPL |
| 254 | FE | SPH |
| 253 | FD | RP |
| 252 | FC | FLAGS |
| 251 | FB | IMR |
| 250 | FA | IRQ |
| 249 | F9 | IPR |
| 248 | F8 | P01M |
| 247 | F7 | P3M |
| 246 | F6 | P2M |
| 245 | F5 | PRE0 |
| 244 | F4 | T0 |
| 243 | F3 | PRE1 |
| 242 | F2 | T1 |
| 241 | F1 | TMR |
| 240 | F0 | SIO |
| Not Implemented | | |
| 127 | General-Purpose Registers | |
| 4 | | |
| 3 | Port 3 | |
| 2 | Port 2 | |
| 1 | Port 1 | |
| 0 | Port 0 | |

INSTRUCTION SET

The Z8 instruction set has 43 instructions which can be divided into six groupings: load (8), arithmetic (10), logical (6), program control (6), rotate and shift (6), and CPU control (7).

Each instruction has up to two operands associated with it. Instructions with no operands perform a function one specific way. Single-operand instructions perform a function using a source or destination operand in one or more of the addressing modes.

Double-operand instructions can perform a function applying the source operand to the destination operand using a combination of one or more of the addressing modes. Some double-operand instructions (for program control) are based on condition codes (as you'll see later on). These instructions are listed in Table 1.

ADDRESSING MODES

Instruction-wise, this set is not so overwhelming. Its flexibility becomes apparent when you understand the various ways in which many of these

| Mnemonic | Operands | Action |
|--------------------------------|----------|--|
| Load Instructions | | |
| CLR | dst | load dst with zero |
| LD | dst,src | load dst with src |
| LDC | dst,src | load dst with constant |
| LDCI | dst,src | load dst with constant and increment dst and src |
| LDE | dst,src | load dst with external data src |
| LDEI | dst,src | load dst with external data src and increment dst and src |
| POP | dst | load dst with stack value |
| PUSH | src | load stack with src |
| Arithmetic Instructions | | |
| ADC | dst,src | add (using carry) src to dst |
| ADD | dst,src | add src to dst |
| CP | dst,src | compare src with dst |
| DA | dst | decimal adjust on dst (BCD data) |
| DEC | dst | decrement dst (byte) |
| DECW | dst | decrement dst (word) |
| INC | dst | increment dst (byte) |
| INCW | dst | increment dst (word) |
| SBC | dst,src | subtract (using carry) src from dst |
| SUB | dst,src | subtract src from dst |
| Logical Instructions | | |
| AND | dst,src | logically AND src and dst |
| COM | dst | complement src |
| OR | dst,src | logically OR src and dst |
| XOR | dst,src | logically XOR src and dst |
| TCM | dst,src | logically AND the src and the complement of dst altering dst |
| TM | dst,src | logically AND the src and the dst without altering dst |

Table 1—Many of these simple instructions have multiple addressing modes, yielding a multitude of permutations.

(Table 1—continued)

| Mnemonic | Operands | Action |
|--------------------------------------|----------|---|
| Program-Control Instructions | | |
| CALL | dst | push PC onto stack and load PC with dst |
| DJNZ | r,dst | decrement r and jump to dst if r NOT zero |
| IRET | | pop flags and PC from stack |
| JP | cc,dst | conditional load PC with dst |
| JR | cc,dst | conditional add dst to PC |
| RET | | pop PC from stack |
| Rotate and Shift Instructions | | |
| RL | dst | rotate dst left |
| RLC | dst | rotate dst left through carry |
| RR | dst | rotate dst right |
| RRC | dst | rotate dst right through carry |
| SRA | dst | shift dst right into carry most significant bit remains unchanged |
| SWAP | dst | swap dst nibbles |
| CPU Control Instructions | | |
| CCF | | complement carry flag |
| DI | | disable interrupts |
| EI | | enable interrupts |
| NOP | | no operation |
| RCF | | reset carry flag |
| SCF | | set carry flag |
| SRP | src | set register pointer with src |

instructions can be used with various addressing modes.

The Z8 has six possible addressing modes—direct, relative, indirect, register, indexed, and immediate.

Although most addressing is done through the eight-bit register file, multiple registers can be used to contain larger 16-bit words. Eight-bit registers can occur at both even and

odd locations, whereas 16-bit register pairs always begin on an even address.

When you're working within a register group (a group of 16 sequential registers), addressing can be simplified by using just four bits to point to any one of those 16 registers within that working-register group.

The Z8 knows which group you are working with by the value stored in the RP register (R253), which is the working (group) register. Using the working registers is generally an option for optimizing your code.

Direct addressing is used by the conditional JUMP and CALL instructions. This destination is a 16-bit address loaded into the PC (program counter) to redirect the program flow. Figure 3a shows an example of direct addressing.

Relative addressing is similar to direct addressing in that it redirects program flow. Instead of using a direct 16-bit destination address to replace the 16-bit value in the program counter, a direct 8-bit destination value is added to the PC.

This value is actually a two's complement signed displacement in the range of -128 to +127. Therefore, the program flow only moves a short distance relative to where it was prior to the instruction. Figure 3b gives you a relative addressing example.

The next three addressing modes can be used for either source or destination data. Indirect addressing uses an 8-bit pointer to find the 8-bit address of the register of interest or a 4-bit working register pointer to find a 16-bit address of the program or data memory location of interest. See Figure 3c for an indirect addressing example.

Register addressing, illustrated in Figure 3d, is the simplest form of addressing. An 8-bit address points to the 8-bit register of interest.

Indexed addressing is similar to register addressing and used solely with the load instruction. However, an offset is first added to an 8-bit address, which then points to the 8-bit register of interest. The offset (or index) is found at a location pointed to by a 4-bit working register pointer.

Figure 3e gives you a picture of indexed addressing.

The last mode, immediate addressing, can only be used as source data because it is essentially a constant predefined in the code memory area. An example of this mode is shown in Figure 3f.

SALUTING THE FLAG

If it weren't for the status flags in every microcomputer, little processing would actually be done. Z8 flags reside in the flag register (R252) and consist of six process flags and two user flags. From most to least significant bits, they are carry, zero, sign, overflow, decimal adjust, half carry, user2, and user1.

In addition to these flag bits, there are 16 condition codes, which can be used for conditional jumps. Although all the flags are covered within the conditional codes, the extra codes make life easier by giving conditions for both signed and unsigned values greater than, greater than or equal to, less than, and less than or equal to.

INSTRUCTION TIMING

The Z8 processors accept RC, LC, crystal, ceramic resonators, or external clock drive and run at speeds up to 16 MHz (depending on the part). The on-chip oscillator is divided by two to produce one instruction cycle.

Instruction completion time requires a minimum of three instruction cycles for each fetch and three for each execution cycle (even for single-byte instructions). The first instruction requires a fetch and an execution cycle, whereas the following instructions are executed one after the next, thanks to pipelining. Pipelining prefetches the next instruction while the last one is executing.

Three-byte instructions require an extra fetch cycle to be added. When the PC is changed by an instruction, the pipe is automatically flushed and the next instruction takes an additional fetch cycle to refill the pipe. Most instructions take only six instruction cycles, but CALL (the longest instruction) takes 20 instruction cycles due to extra stack manipulations.

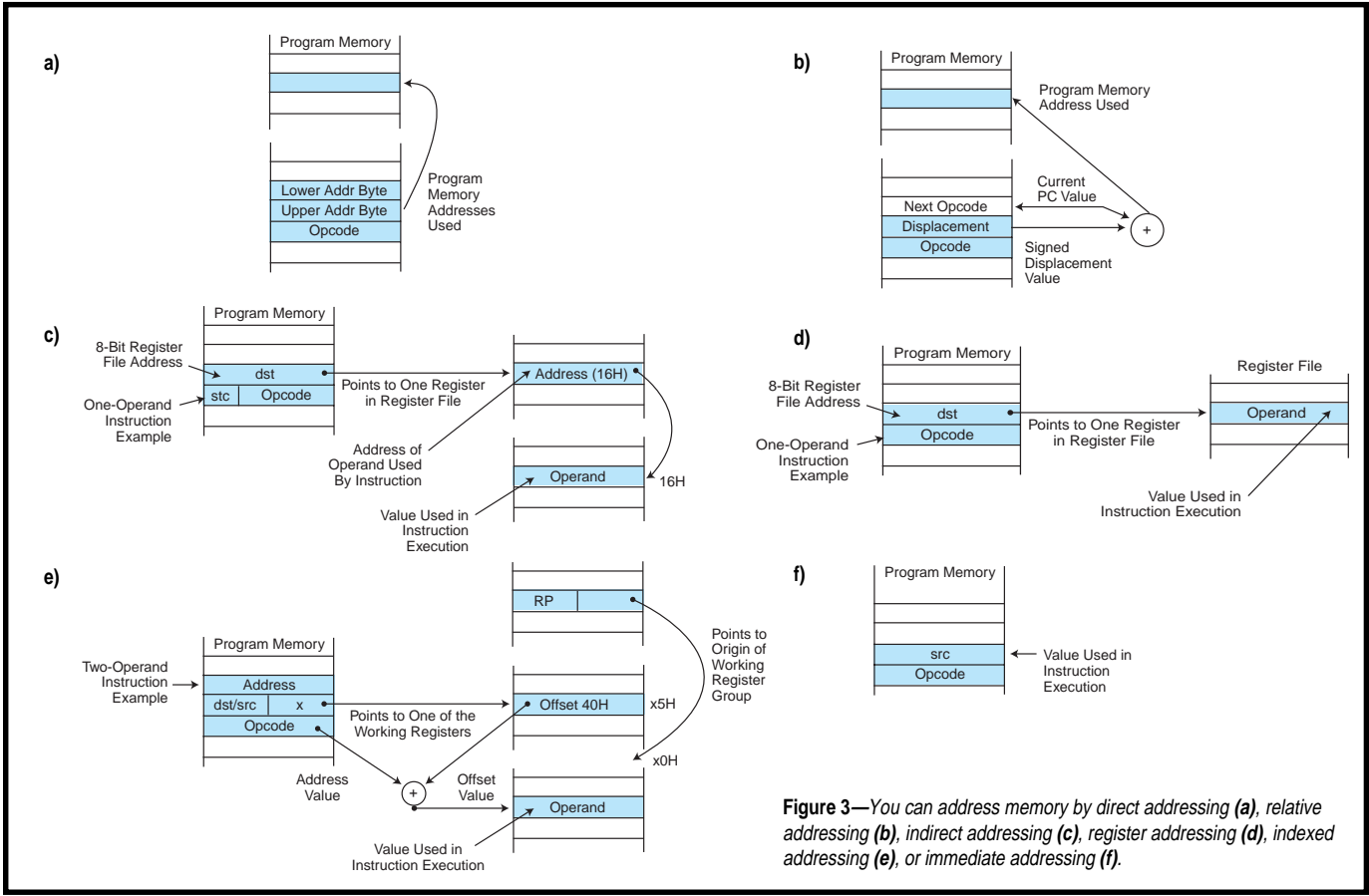


Figure 3—You can address memory by direct addressing (a), relative addressing (b), indirect addressing (c), register addressing (d), indexed addressing (e), or immediate addressing (f).

ADDITIONAL FEATURES

The CMOS I/O can source 2 mA and sink 12 mA while safely staying within logic output specification, even though the maximum current can be much higher. Internal brownout protection places the Z8 in reset if V_{CC} drops below ~ 3 V.

No external reset is necessary because power-on reset delays execution ~ 70 ms. An independent internal watchdog timer is available to restart operations should execution stray from its normal course.

Halt and Stop modes allow the Z8 to reach standby currents of milliamps and microamps, respectively.

The Z8 has a flexible interrupt structure. You may choose to use vectored (and prioritized) interrupts or poll the interrupt request register directly. Nested interrupts are also allowed.

UP TO SPEED

The Z8 comes from a company that was founded back in '74 and that within a year introduced the Z80 to the world. The Z80 rapidly became

the most popular and best-selling microprocessor in the world.

Based on its big brother, the Z8 was first produced as an NMOS device. Later, in the '80s, the CMOS process was introduced and now OTP parts are available.

The flexible addressing modes of the Z8 enable coding shortcuts like context switching via working register groups. Load and increment instructions provide for block moves with an absolute minimum of coding.

The open use of the stack for passing parameters to and from subroutines is a welcome sight. And, the ability to prioritize interrupts reduces coding within the interrupt routines.

The Z8's instruction set lends itself well to multiply, divide, conversion, and BCD arithmetic algorithms. Now with OTP parts and low-cost tools, Zilog has become a force to be reckoned with.

We're now through with introductions. Next month, I'll start a simple project using one of the OTP members of the Z8 family of processors. If you want more info now, visit the Zilog

Web site and download some of the available Z8 tools.

In the meantime, my week of mid-winter Indian summer must be over. It's now more like the winter I know—it's starting to snow. Guess I'll need my jacket after all. ☞

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.

SOURCE

Z8

Zilog, Inc.
210 E. Hacienda Ave.
Campbell, CA 95008-6600
(408) 370-8000
Fax: (408) 370-8056
www.zilog.com

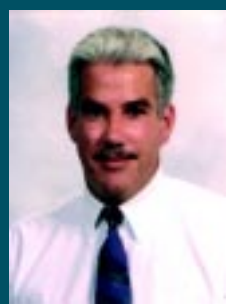
I R S

425 Very Useful
426 Moderately Useful
427 Not Useful

SILICON UPDATE

Tom Cantrell

VolksArray



Xilinx's Spartan reminds Tom of a VW Bug—

simple, good value. However, it also offers the style and good design of a high-end Porsche. It even provides user RAM and a flexible interconnect system.



ave you heard that Volkswagen's introducing a new Beetle?

I still remember my first car, Dad's hand-me-down '63 VW Bug. Yeah, with something on the order of 60 hp, it wasn't a speed demon (probably just as well). It had other notable foibles as well, including a rubber-band-like shifter and a heater that didn't.

Yet, the Beetle's simplicity and purity of purpose overcame all. Lack of amenities like power anything, air conditioning, a cooling system, and so forth simply meant less to break. And if a problem did occur, it was usually cheap and easy to fix or even ignore. I once went a couple of weeks relying on solo push starts. Don't try that with your SUV!

Though perhaps remembered for its cuteness, in fact, the Beetle's legacy is much more profound. It

Photo 1—High pin count yet small form-factor plastic packages are key parts of the Spartan cost-cutting equation.

was the first economy car (both in terms of price and operating cost) that could reach deep into the middle class, with used ones trickling down to poor students, surfers, hippies, and the like.

What's all this reminiscing got to do with silicon? Let's check out the recently introduced line of economy FPGAs from Xilinx, and I think you'll see the connection.

GO SELL THE SPARTANS

Referring to ancient Greek tough guys (males were drafted at age 7), my encyclopedia observes that "the word Spartan has since become a byword for endurance and rugged simplicity," words aptly describing that '63 bug—and the new chips from Xilinx shown in Photo 1.

Until now, reflecting the inexorable onward and upward march of silicon, most FPGA activity has been focused at the high end. Indeed, the same Xilinx press kit includes the announcement of their Virtex line, touting 0.25- μ m five-layer metal chips that deliver millions of gates and 100+ MHz.

Of course, such capabilities don't come cheap, with rarefied prices into the hundreds and thousands of dollars. Targeting bleeding-edge signal-processing and reconfigurable-computing apps, performance-at-any-price FPGAs are the equivalent of a 911 Turbo.

By contrast, pricing for the Spartan line busts the single-digit barrier with volume projections, making FPGAs an option for the huge middle class of cost-sensitive high-volume designs.

Just as the VW shared more than a bit of heritage with the pricey Porsche,



| Device | Logic Cells | Max. System Gates | Typical Gate Range (Logic and RAM)* | CLB Matrix | Total CLB | Flip-flops | Max. User I/O | Packages | Price Projection 'XL 100k '99 |
|------------|-------------|-------------------|-------------------------------------|------------|-----------|------------|---------------|-------------------------|-------------------------------|
| XCS05 (XL) | 238 | 5000 | 2000–5000 | 10 × 10 | 100 | 360 | 80 | 84, 100 | \$2.95 |
| XCS10 (XL) | 466 | 10,000 | 3000–10,000 | 14 × 14 | 196 | 616 | 112 | 84, 100, 144 | \$4.45 |
| XCS20 (XL) | 950 | 20,000 | 7000–20,000 | 20 × 20 | 400 | 1120 | 160 | 100, 144, 208 | \$5.45 |
| XCS30 (XL) | 1368 | 30,000 | 10,000–30,000 | 24 × 24 | 576 | 1536 | 192 | 100, 144, 208, 240, 256 | \$6.95 |
| XCS40 (XL) | 1862 | 40,000 | 13,000–40,000 | 28 × 28 | 784 | 2016 | 224 | 208, 240, 256 | \$9.90 |

*Note: Max. values of Typical Gate Range include 20–30% of CLBs used as RAM.

Table 1—The new Spartan line from Xilinx, featuring 5- and 3.3-V (XL) versions from 5k to 40k gates, positions FPGA technology squarely in the high-volume application arena.

you'll find that the Spartan repackages the essence of the FPGA concept, while cutting sticker price to the bone.

AIR COOLED

Table 1 summarizes the lineup, which consists of various permutations of logic and I/O density, duplicated in 5- and 3.3-V (XL) versions. Cost savings start with plastic packages from 84 to 256 pins.

To ease migration, packages are footprint compatible across logic density. For instance, a single 100-pin layout can accommodate an XCS05, '10, '20, or '30.

Although it may not have the chrome and tail fins of a high-end FPGA like the XC40125XV (560 pin, \$1500!), a look under the hood reveals a lot of similarity (see Figure 1). Both chips rely on a matrix of configurable logic blocks (CLBs) surrounded by a ring of I/O blocks (IOBs, one for each pin), all lashed together with a three-tier (single-length, double-length, and long lines) programmable interconnect.

Additional circuits handle start-up initialization of the SRAM switches that define the logic and interconnect. The internal state, including SRAM and other key nodes, can be read back unobtrusively (i.e., during normal chip operation) for in-system debug. The chip also supports JTAG (IEEE 1149.1) boundary scan as an alternative debug and configuration mechanism.

Though fewer in number, the Spartan CLBs (see Figure 2) match the sophistication

of those found in upscale FPGAs. Each CLB comprises a pair of four-input function generators feeding a third three-input function generator.

The function generators are implemented as simple SRAM look-up tables (LUTs). For example, the four-input function generators are essentially 16 × 1 SRAMs, with the inputs serving as address lines and the output reflecting the logic function. For instance, a four-input AND is achieved by storing a 1 at address 1111 and a 0 at 0000–1110, and a four-input OR by storing a 0 at address 0000 and a 1 at 0001–1111.

One benefit of the scheme: the delay (i.e., SRAM access time) is the same regardless of the function programmed. For example, turning AND and OR into NAND and NOR is simply a matter of complementing the SRAM data, rather than adding an inverter.

The output of the CLB is via a pair of signals (x and y) made available in both combinatorial and registered form. The latter uses a pair of flip-flops which offer a degree of flexibility even though they share common clock (CK), clock enable (EC), and set/reset (SR) lines.

For instance, although the clock is common, each flip-flop can be independently configured to trigger on the rising or falling edge. Similarly, the active-high SR input can be configured to either set or reset each flip-flop independently. The clock enable can be left disconnected at either or both flip-flops, in which case it defaults to enabled.

In between the input and output, a number of multiplexers do their best to get the signal you want where you want it. One mux selects between direct (i.e., SR and DIN) or function generator (F-LUT, G-LUT) inputs to the H-LUT. Combinatorial output choices are limited to F-LUT or H-LUT for the x output and G-LUT or H-LUT for the y.

However, for registered outputs, 4:1 muxes steer any of the three LUT outputs or DIN to either flip-flop. Furthermore, though not shown in Figure 2, each CLB is fronted by four general-control inputs (C1–C4), any of which can drive any or all of the direct inputs SR, H1, DIN, or EC.

Put it all together, and you find the CLB is rather agile in the turns. Obviously, it can handle two functions of four inputs and a third of three inputs (corresponding directly with the G-, F-, and

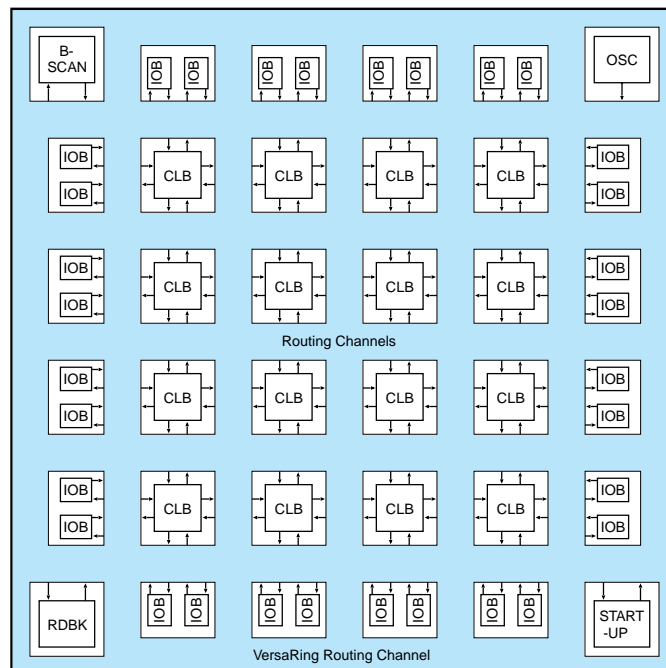


Figure 1—Despite entry-level pricing, Spartan's architecture, comprising configurable logic blocks (CLBs), I/O blocks (IOBs), and programmable interconnect, isn't stripped by any means.

H-LUTs). However, clever wiring lets a CLB alternatively deliver any function of five inputs, any function of four inputs, plus some functions of six inputs, and even some of up to nine inputs.

RAM CHARGER

One big advantage Spartans have over competing econo-chips (including Xilinx's own XC5200) is the specific provision for user RAM.

It's ironic that although they were built on the foundation of SRAM, the earliest FPGAs could barely deliver a few bytes when needed for an on-chip FIFO, register file, or constant (e.g., filter coefficient table). The designer might have to blow a whole CLB for just a few inelegantly brute-forced bits or, worse, take a costly excursion off-chip.

By contrast, Spartan incorporates the Select-RAM concept pioneered on the upscale XC4000, in which a CLB is decomposed to expose its SRAM underpinnings. Relying on the function generator LUTs for storage, a CLB can be configured as one or two 16 × 1 RAMs, a 32 × 1 RAM, or a 16 × 1 dual-port RAM. In fact, the Spartan Select-RAM betters that on the original XC4000 with synchronous timing that eases design and speeds access.

Although it's not dense or cheap by memory IC standards, the Spartan's ability to pack at least a weekend's worth of RAM will prove popular when designers come in to kick the tires.

WIRING HARNESS

Copious and flexible interconnect is critical for efficient utilization. It's

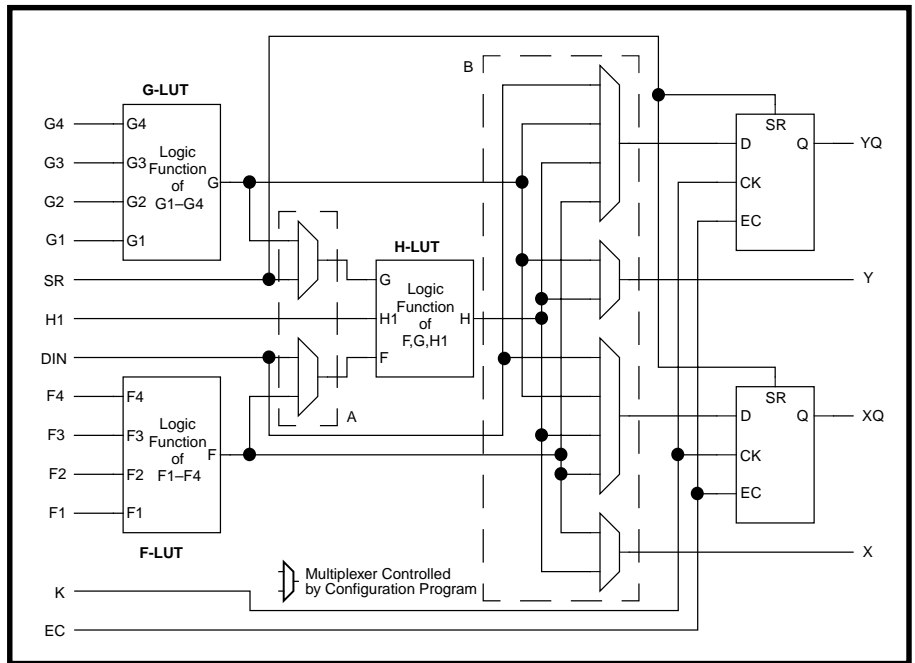


Figure 2—The CLB is the heart and soul of FPGAs. It's composed of three SRAM-based look-up tables (LUTs) and two flip-flops connectable in a myriad of ways. Not shown in this simplified view are additional selectable input sources and dedicated carry chains.

frustrating to have the CLBs you need but no way to get at them. Fortunately, Spartan doesn't skimp on the wiring.

There are eight horizontal and eight vertical single-length lines for each CLB. They provide fast and flexible connection between adjacent CLBs, but accumulating switch delays rule them out for longer distances. Similarly, each CLB has four horizontal and four vertical double-length lines that run twice as far between switches.

Long lines form a grid of interconnect segments running the entire length and width of the CLB array. Each line has a programmable splitter halfway that can turn it into two independent

routing channels, each traversing half the array. Also, every CLB includes a pair of tristate buffers that can drive onto adjacent horizontal long lines, which facilitates the creation of bidirectional or multiplexed buses.

Additional lines (eight double-length and four long) ring the chip to connect CLBs and IOBs. They help decouple pin assignment from logic so the pinout can remain locked across design changes or be tweaked to facilitate PCB layout.

For clocks and other high fan-out control signals, global buses (four vertical lines in each CLB column) are driven by primary and secondary buffers. Primary buffers connect to dedicated pins and offer the least skew, whereas secondary buffers connect to either dedicated pins or internal logic.

Finally, each CLB has carry-in and carry-out lines that weave their way from one corner of the chip to the other. This logic greatly improves performance for arithmetic functions like adders, comparators, and counters.

WHERE THE PIN MEETS THE PAD

Once the logic and interconnect are worked out, the IOBs (see Figure 3), one for each pin, come into play.

With two wires (I1 and I2) available, inputs can be routed directly to interconnect, via an input register, or both.

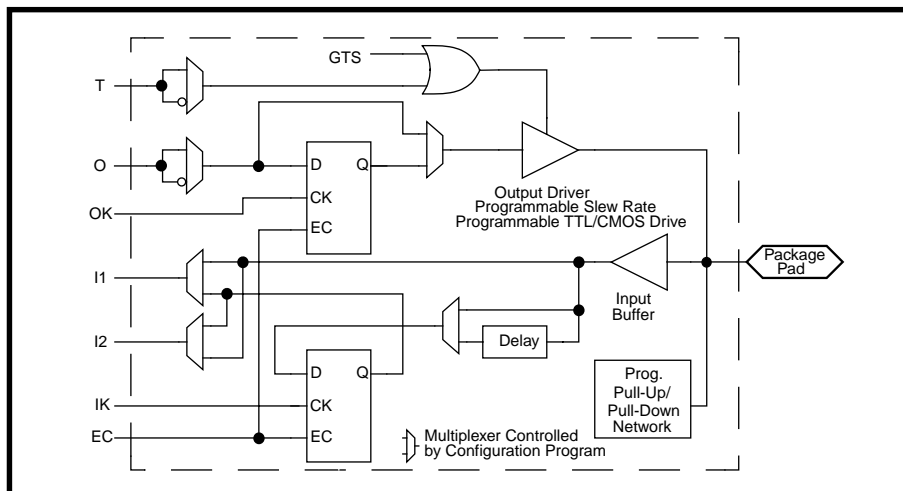


Figure 3—I/O blocks feature programmable registered, combinatorial, TTL/CMOS level, slew rate, and pull-up options.

The input register can be configured as either a level-sensitive latch or edge-triggered flip-flop. An optional delay can be inserted to allow zero hold time for the input relative to the global clock.

Similarly, either a direct or registered output (0) can be delivered to the pin and optionally inverted. The individual output can be tristated with an active-high or -low control signal (T), while a dedicated global tristate signal (GTS) turns off all outputs.

Each flip-flop can be individually configured to set or clear after powerup and in response to the GSR (global set/reset) signal. Much like the CLB, the clocks for each flip-flop are separate and invertible, whereas the clock enable, though common, can be disabled at either or both flip-flops.

Input and output levels are separately programmable as either TTL or CMOS (i.e., rail to rail), with the resulting compatibility matrix shown in Table 2. There's also an output slew-rate control to trade off switching speed for reduced power and noise spikes.

The slew rate is automatically limited after powerup, when all outputs are simultaneously driven to their initial state. Also, unused pins are automatically pulled up to reduce noise sensitivity and minimize power consumption.

YOUR MILEAGE MAY VARY

Part of the Xilinx marketing premise is that FPGAs can encroach on the low end of the gate-array market. According to Dataquest, nearly half of gate-array design starts are 50k gates or less, and thus subject to attack by the Spartans.

However, gate count is the ASIC and FPGA equivalent of MPG (or MIPS for CPU chips). Drive with a lead foot (i.e., sloppy design) and you get one number. Drive downhill with a tailwind and tires inflated to 60 PSI (i.e., artificial example) and you get another.

To help make sense of it all, I put in a call to a local expert on the subject.

| | Spartan | | Spartan-XL |
|--|-----------------|-----------------|-----------------|
| | TTL (5.0 V) | CMOS (5.0 V) | CMOS (3.3 V) |
| Source | Inputs | Inputs | |
| Any device, VCC = 3.3 V CMOS outputs | Y | Unreliable data | Y |
| Spartan series, VCC = 5 V TTL outputs | Y | Unreliable data | Y |
| Any device, VCC = 5 V TTL outputs (Voh <= 3.7 V) | Y | Unreliable data | Y |
| Any device, VCC = 5 V CMOS outputs | Y | Y | Y |
| Destination | Outputs | Outputs | |
| Any device, VCC = 3.3 V CMOS-threshold inputs | Y | Some* | Y |
| Any device, VCC = 5.0 V TTL-threshold inputs | Y | Y | Y |
| Any device, VDD = 5 V CMOS-threshold inputs | Unreliable data | Y | Unreliable data |

*Note: Only if destination device has 5-V tolerant inputs.

Table 2—Most applications interface requirements can be met, thanks to the availability of 5- and 3.3-V (XL) versions and programmable TTL or CMOS I/O levels.

Thanks to Phil Freidin (fliptron@netcom.com) for sharing his considerable FPGA experience and insight.

The most optimistic (i.e., snake oil) interpretation of gate count is to imagine how many gates it takes to duplicate the FPGA. For instance, the four-function generator could be built with banks of AND, OR, and XOR gates plus an inverter.

However, the count would also include all the configuration SRAM (about 500 bits per CLB), not to mention the multiplexers and other support logic that isn't accessible to the designer. Fortunately, even the most aggressive marketer knows that overstating reality by about 10x won't fly.

Even ignoring support logic, the Select-RAM feature introduces its own bias. As mentioned, it's easy to blow a lot of gates brute-forcing RAM onto a chip with no specific provision for it. That's the reason for Table 1's note. Compared to a no-RAM ASIC, the Spartan equivalent gate count increases to the degree you use the Select-RAM.

Comparison within the Xilinx product line is easy enough using CLB count. However, competitors have introduced their own flavors of FPGAs that offer logic blocks of differing complexity. Enter the concept of logic cells, roughly equivalent to a four-function generator plus flip-flop.

According to Phil, in ASIC terms a logic cell is worth about 16 gates (6-8

for four-input LUT plus 8 for a flip-flop with CE and S/R). RAM takes about four gates per bit. And, don't overlook the fact that JTAG is worth about 70-100 gates per pin.

Plugging in the numbers for the Spartan chips, I come up with something close to the middle of the claimed gate-count range. To Phil, optimistic gate-count specs are often best interpreted as "guaranteed not to exceed."

One thing that's for sure is you'll get more gates for less bucks today than yesterday (and tomorrow than today). When I

wrote my first article about FPGAs way back in '89 ("Beyond ASICs," *INK* 10), prices were about 10 cents per gate. Today, it's about 10 gates per cent! The new Beetle may well qualify as an economy car, but I guarantee it won't cost less than the old one.

Ultimately, gate mileage and even the prospects for acceptance of FPGAs in mainstream designs depend as much on tools as the chips themselves. Next month, we'll take a look at FPGA tool trends and see what it takes to jump start your design. ☐

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.

SOURCE

Spartan series FPGAs
Xilinx, Inc.
2100 Logic Dr.
San Jose, CA 95124
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com

I R S

428 Very Useful
429 Moderately Useful
430 Not Useful

PRIORITY INTERRUPT

What You Get with a Handshake



had a very interesting dinner last night with a distributor salesman and a couple manufacturers' representatives. For those of you who aren't familiar with the relationships, the simplest description is that a distributor sells and a rep facilitates. Distributors stock many competing components and, except for very high-volume purchases, they are the place where you physically spend the money when you order parts. Like cross-brand car dealerships, if you walk in the door to look at a BMW and choke at the price, they have little hesitation to lead you over to the Buicks they also sell. Their sales approach is oriented toward building dealership loyalty rather than strict brand loyalty.

Manufacturers' reps facilitate the sale of a specific brand. When you check a bingo card, fill out a literature request, or otherwise ask for specific product information, your name and vital statistics are sent to the manufacturer's rep for that product in your geographical area. Even though the datasheets may come to you directly, generally you can expect a call from the manufacturer's rep. His job is to help you find brand loyalty. If he comes to you because you asked about Teccor triacs, Teccor expects that he's not going to talk to you about the Motorola triacs even though he might also represent Motorola.

For an engineer, getting product information and samples are important. Years ago, unless your literature requests had a major company name on it, they would be ignored. You might get a call from a rep, but the first question had to do with your intended volume rather than your intended application. For many of us, it was tough to get the parts we needed.

This obvious discrimination was the result of thinking that only \$100-million companies ever design something with a product volume of interest. Fortunately, the advent of personal computers changed that assumption. Traditionally, only large companies could afford to design products that might be manufactured in volume. The advent of low-cost personal-computer-based design and development tools ultimately made physical location and company name less relevant.

When I mentioned this at the table, everyone agreed that what I described was a historical fact but they also believed the situation was quite different today. What I found interesting was that the catalyst for change was basically the same for all of them. The typical story always seemed to involve some seedy-looking guy with a parts list. He has this little widget he's putting together but can't get a distributor to sell him a few pieces or a manufacturer's rep to take him seriously. Finally, the guy finds a bunch of parts from nontraditional sources and makes his product. Later on, when requests for pricing this widget in 20-million quantities are floating around the industry, everyone comes to find out that he was designing it for Milton-Bradley in his basement. Of course, the design is locked in, and most of them are locked out.

They went around the table laughing as they described similar experiences where a little guy turned out to be something unexpected. Today, they're very careful not to prejudge a customer's qualifications simply by appearances.

Today, information and product support is abundant. Call distributors like Hamilton Hallmark or Future Electronics, and they have ways to satisfy small orders. Additionally, outside salespeople have become more knowledgeable. The good ones aren't just order takers. They complement traditional reps without as much brand prejudice. So, now that it all seems to be working well, what happens in the future?

Much to my surprise, they were concerned. Both reps and salespeople applauded the instant availability of manufacturer datasheets via the Internet. It certainly reduced the workload of satisfying requests. However, the anonymity of most of these requests was a major concern. While a distributor probably still gets to sell parts, the rep justifies his existence by appearing to enhance brand loyalty among the contacts he makes. If the majority of sales appears to be straight from anonymous Internet download to production order, who needs reps?

The picture isn't all that clear for the distributors, either. Without them specifically admitting it, I think their nemesis is high-volume catalog outfits. Distributors feel they add personal service and support to sales. Catalog outfits just take orders and ship. Distributors hate to compete with catalog pricing.

For the most part, I agree with them. I believe very much in distributor loyalty if not always in brand loyalty. Whatever the prognosis, history has taught us to expect the unexpected. I can't wait for the next episode.



steve.ciarcia@circuitcellar.com