www.circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

#109 AUGUST 1999

DEVELOPMENT AND DEBUGGING

Designing with Virtual Components

In-Circuit Debuggers—Shorten the Cycle

Building a USB Host Controller

Embedded GPS

CIRCUIT CELLAR ONLINE

Double your technical pleasure each month. After you read *Circuit Cellar* magazine, get a second shot of engineering adrenaline with *Circuit Cellar Online*, hosted by ChipCenter.

WWW.CHIPCENTER.COM/CIRCUITCELLAR

Table of Contents for July 1999

INSIDE ISSUE 109

EMBEDDED PC

Persistence Pays

It's often said that showing up is half the battle. And if it gets you 50% of the way toward your goal, well, then that's great. In fact, maybe if you can get that far ahead so quickly, then you get to save up all your energy for what is, in my opinion, the hardest part—sticking it out until the end.

Whether it's developing a marketable embedded system, pulling together another magazine's worth of quality editorial, or starting a business venture, the process is pretty much the same. Design it, do it; tweak it, do it again; fiddle with it, do it some more.

The necessary ingredients are similar, too. There are the moments of inspiration, the months of headaches, and of course all those times when you can't figure out what could possibly be the problem with that one part that just won't come together the way you want it.

In this issue, with its focus on development and debugging, our concern is on not only the how-we-do-it (the trademark *Circuit Cellar* article) but on how-we-circumvent-all-the-obstacles-that-keep-us-from-doing-it. Of course, I'm not talking here about how to budget your time more effectively so you can accomplish two tasks at once or how to tell the guy in the next cubicle to turn his radio down so you can even begin to think.

But the willingness to keep plugging away on a project—that's a characteristic we all can benefit from. Think about it: how many people do you know who will start a project but never finish? And how many people do you know who, once they set their mind to a task, will never quit? (Which number is higher? Who do you admire more?)

On a rare occasion, it may be worthwhile to throw in the towel. But it's more likely that you just need a different approach. There's a big difference between quitting and changing direction.

More often than not, regardless of what you're working on, the original ideas need to be changed around a bit. Nobody starts a project with the perfect plan. And nobody executes a task perfectly from beginning to end.

But we are always encouraged to hear stories about beating the odds, overcoming setbacks, or finding the new approach that leads us more effectively to the goal. We have such articles in abundance in this issue.

First off, Thomas Anderson examines how virtual components ease the task of designing for reuse. Next, the debugger comparison presented by John Andrews and John Day helps you choose the appropriate system that enables you to get your latest project to market faster.

In our third feature, Gordon Dick tackles a variable frequency drive project that's been haunting his workbench for years. And Kenneth Ciszewski explains how flash memory can be a cost-efficient solution when you've got tough design requirements in front of you.

Eli

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR®

THE MAGAZINE FOR COMPUTER APPLICATIONS

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

MANAGING EDITOR
Elizabeth Laurençot

TECHNICAL EDITORS
Michael Palumbo
Rob Walker

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Mike Baptiste
Ingo Cyliax
Fred Eady
George Martin
Bob Perrin

NEW PRODUCTS EDITOR
Harv Weiner

ASSOCIATE PUBLISHER
Sue Skolnick

CIRCULATION MANAGER
Rose Mansella

CHIEF FINANCIAL OFFICER
Jeannette Ciarcia

ART DIRECTOR
KC Zienka

GRAPHIC DESIGNER
Jessica Nutt

ENGINEERING STAFF
Jeff Bachiochi
Ken Davidson
John Gorsky

PRODUCTION STAFF
James Soussounis

PROJECT EDITOR
Janice Hughes

EDITORIAL ADVISORY BOARD
Ingo Cyliax Norman Jackson David Prutchi

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

ADVERTISING

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

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

CONTACTING CIRCUIT CELLAR

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, 4 Park St., Vernon, CT 06066

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

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

CIRCUIT CELLAR®, 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 Subscriptions, P.O. Box 698, Holmes, PA 19043-9613 or call (800) 269-6301.**
Postmaster: Send address changes to Circuit Cellar, Circulation Dept., P.O. Box 698, Holmes, PA 19043-9613.

NEW PRODUCT NEWS

Edited by Harv Weiner

MICROCONTROLLER CHIP

The **BasicX** microcontroller was designed for rapid development of control applications using an easy-to-learn BASIC language compiler. Its controller features an on-chip multitasking OS with IEEE single-precision floating point and networking. The chip supports compiled code up to 8 MB and RAM up to 128 KB in external storage. A built-in network allows multiple BasicX processors to communicate with each other over an RS-485 multidrop connection at speeds up to 460 kbps.

BasicX also features on-chip peripherals and functions like real-time clock/calendar, timers and counters, analog comparator, watchdog timer, dual PWMs, and SPI peripheral bus. It offers high-power I/O lines and low-power sleep modes as well. BasicX has up to 32 programmable I/O pins. Routines are provided to communicate with LCD, servos, ND converters, and other typical applications.

The **BasicX development kit** (**$99.95**) includes an RS-485 network, RS-232 COM port, in-circuit emulator/expansion connector, 32-KB code EEPROM, and 256 bytes of on-chip RAM. The development system is connected to a PC parallel port for downloading, and a complete BasicX compiler and editor allow easy development.

BasicX is currently available in three packages: 40-pin DIP, 44-pin PLCC, and TQFP. Quantity pricing is **$9.95** in 1000s.

NetMedia, Inc.
(520) 544-4567 • Fax: (520) 544-0800
www.netmedia.com

POINTING STICK CONTROLLER

The USAR **PixiPoint Z encoder IC** is a strain-gauge miniature joystick with *z*-axis functionality. The device is equipped with an advanced motion algorithm that provides smooth and accurate cursor control in all directions.

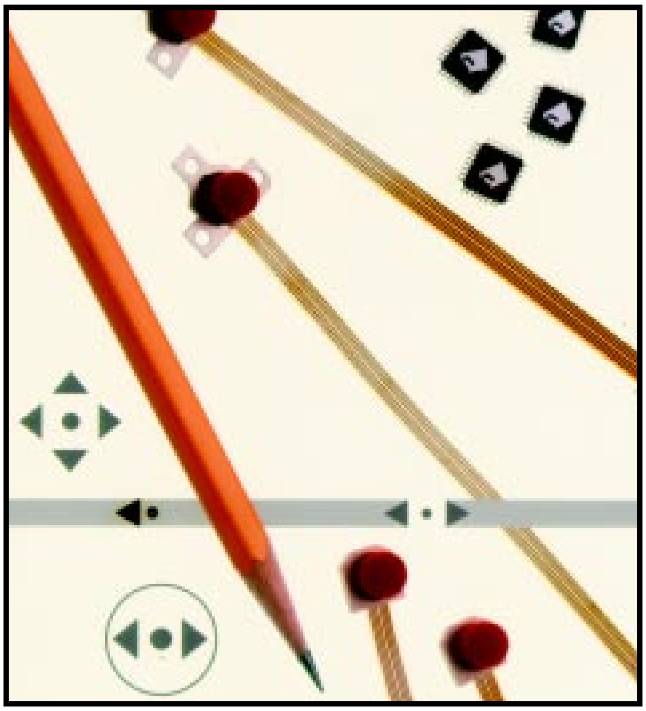
The IC implements the Tap (select), Double Tap (execute), and Press Hold (drag) functions in firmware with no need for special drivers. Additionally, the track stick IC enables Scroll as well as the anchor functions of Autoscroll and Panning. Panning allows for both horizontal and vertical movement.

Other features include a PS/2 port for the hot-plug connection of an additional pointing device and CleanStop and EasyDrag technology. USAR's proprietary CleanStop technology ensures that the cursor stops the instant the user needs it to. The EasyDrag function reduces the cursor's speed when the user drags an object on the desktop, making it simple to accurately handle the object.

The USAR PixiPoint Z IC uses a sophisticated high-precision signal-conditioning circuit. Simple to implement, the complete circuit requires few components, so it is cost effective and saves real estate.

The USAR PixiPoint Z IC is **$3** in OEM quantities.

USAR Systems, Inc.
(212) 226-2042 • Fax: (212) 226-3215
www.usar.com



NEW PRODUCT NEWS

SINGLE-BOARD COMPUTER

The **ipEngine-1** is a miniature SBC and associated software designed to significantly reduce the complexity required to network-enable embedded products. It incorporates a PowerPC processor, 16 MB of DRAM, and 2 MB of flash memory. It features an LCD controller, USB and Ethernet interfaces, 16,000-gate FPGA, and switching power supply onto a board the size of a credit card.

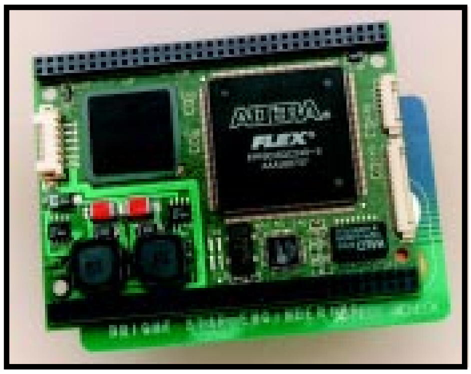
Complementing this highly integrated board is a choice of two preintegrated OS environments. The first is BSE's real-time pKernel POSIX-based network OS. The second option is an embedded version of Linux that contains the full kernel plus network utilities, the Apache Web server, and a Java Virtual Machine. Both provide full TCP/IP networking and web-server support.

Unlike traditional board-level products with fixed interfaces, the ipEngine-1 can adapt itself to the user's hardware requirements. The external connection to the ipEngine hardware is via an FPGA-based "virtual interface" that can be configured on-the-fly to adapt to the user's needs. The FPGA can emulate a variety of bus architectures, and it can also implement peripheral functions like UARTs, PWM control, memory emulation, data capture and synthesis, and interfaces to a variety of input devices.

Developer's toolkits for both the Embedded Linux and pKernel operating-system environments contain all of the hardware and software needed for a developer to start building applications. The toolkits include an ipEngine-1 board, case and power supply, the Linux or pKernel distribution, a GNU-based PowerPC cross-compiler tool suite, sample FPGA code, documentation, and technical support.

Pricing for the ipEngine-1 is **$895** for quantities 1–4 and from **$350 to $795** in OEM volumes. The pKernel and Embedded Linux developer's kits have an introductory price of **$1995**, and they include six months of maintenance support.

Bright Star Engineering, Inc.
(978) 470-8738
Fax: (978) 470-8878
www.brightstareng.com



FUZZY-LOGIC ROBOT

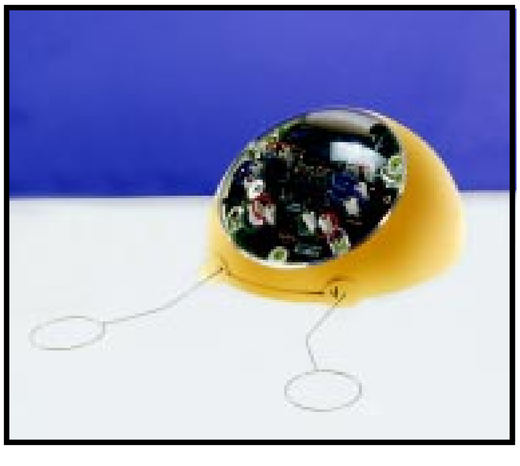
OWI has introduced the **WAO-G** robot. This limited-edition device attempts to dethrone the traditional method of control and problem solving by using fuzzy-logic principles. These principles exploit imprecise decisions, such as "getting close is generally OK!" By building and programming WAO-G, you can set up membership functions and learn basic fuzzy control principles. WAO-G can draw straight lines, circles, and even words by putting a pen in its pen-holder.

WAO-G will help in understanding the new control methods being used in washing machines, vacuum cleaners, video camcorders, cameras, and automobiles. You can learn how to input data using eight kinds of membership functions. Simple operations are commanded in Direct mode, and complicated operations are commanded in Program mode. The robot uses sensor feelers to detect its movement, and an optional interface enables the user to program the device through a PC. Three demonstration programs (Dice, Roulette, and Timer) are included for illustration.

The robot measures 9.5″ × 6.25″ × 3.5″ and is powered by three AA batteries and one 9-V battery. Power consumption is 12.5 mA for the electronics and 600 mA for the mechanics. The unit includes 48 (max.) program steps and 16 (max.) for-next loops.

The WAO-G robot sells for **$89.95** and is available as a kit or fully assembled.

OWI, Inc.
(310) 638-4732
Fax: (310) 638-8347
www.owirobot.com



NEW PRODUCT NEWS

SINGLE-CHIP DATA LOGGER

The **DS1615 Temperature Recorder** is an integrated temperature recorder that combines a Y2K-compatible real-time clock with temperature data-logging and histogram capabilities. It is designed for applications that require temperature profiling over a given period of time.

The DS1615 can operate in a stand-alone system as a complete data logger or in an embedded system to track environmental conditions during system powerdown or minimal power periods. Its digital thermometer measures temperatures from –40°C to +85°C in 0.5° increments with ±2° accuracy. The data can be used in calibration, maintenance, and warranty applications. Also, the DS-1615's real-time clock and temperature sensor are available to the system.

Both standard three-wire synchronous and RS-232 interfaces with a built-in CRC generator are available. Nonvolatile memory can record 2048 consecutive temperatures, which the user can program to start at specified times and be taken at selected rates. Alarms and interrupts can be programmed for user-determined out-of-spec conditions, including expiration dates and extreme temperatures. The separate histogram memory allows for longer term data collection and distribution analysis.

A **DS1615K evaluation kit** enables designers to program and retrieve data from the DS1615. The kit has a self-contained data-logger board, RS-232 cable, Windows-driven software, source code, and documentation.

The DS1615 Temperature Recorder costs **$5.43** in 1000-piece quantities. The DS1615K Temperature Recorder evaluation kit costs **$75** in single quantities.

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

READER I/O

GETTING A MOUTHFUL

Steve's "Servings Per Issue" editorial (*Circuit Cellar* 107) generated plenty of responses with regards to the editorial direction of the magazine. Here are a few of your responses:

Popular Science is one of my favorite publications also and I hadn't been able to put a finger on why it seemed less satisfying now than in the past. As an instructor of electronics technology (mostly DC, AC, transistors, and digital), I use *Circuit Cellar* as a tool to introduce first-year students to the applications part of the technology with hopes that during their second year they will go beyond the textbooks with their projects.

The new-product quickies are important, but it's the complete projects with in-depth coverage that makes this magazine unique. The touchless sensor ("Look Ma, No Hands"), working with accelerometers ("XLR8R"), and the "USB Primer" Microseries are three excellent examples of what's needed in today's classrooms and labs. Don't make too many changes.

George Shaiffer
gshaiffer@uswest.net

As a subscriber since *Circuit Cellar*'s first issue, I enjoy the in-depth articles. Even though I don't always have time to read them all, I know the meat is there in case I get really hungry. Better than the other way around, isn't it?

Last year, MIT's *Technology Review* magazine made the sort of transition that you described regarding *Popular Science*. I hate the result to the point that I no longer look forward to reading the publication.

Why is everyone trying to copy *Wired* magazine's practice of bombarding the reader with trivia? Engineering is not a trivia quiz. When I need snippets of information, there are plenty of publications and trade journals available.

If anything, I'd like to see *Circuit Cellar* articles offer more theory along with the nuts and bolts. Like many people, I learn best from a happy marriage of theory and practice. Please don't stop selling steak and offer just sizzle. Leave the glitz to the others, and keep helping us become better designers as you've been doing all along.

Phil Doucet
pjdoucet@aol.com

"Servings Per Issue" doesn't have a single numerical answer. I need about one hundred new ideas a week to feel satisfied. At the same time, I also have a need to learn the detail behind those ideas.

Product announcements and sound bytes don't teach much, but full-length articles do. For example, where else would I find out how to create a web-based strip recorder? Sure, I could find out where to buy one. But I derive my satisfaction from learning how to build things, not just where to buy them.

The web is great if you know what you're looking for so the needle in a haystack problem becomes trivial. The problem is that you'll never find the stapler, thread, or any of the thousands of possible substitutes for the needle.

Short paragraphs arranged in a special-interest magazine can show those alternatives, but it's not enough to present other people's products and inventions. A magazine article should have in-depth content that teaches how to build new products and how things work. It should present different approaches to a problem and point out how others have solved similar problems. A vendor has no desire to teach people how to build their products, but a magazine article can teach the underlying technology of new products and concepts, and that's something that appears nowhere else.

Paul Hittel
phittel@microtest.com

I have been an avid reader of Steve's work for many years dating back to the *BYTE* days. I go all the way back to the 4004, 8008, and 6800 micros and have enjoyed watching and applying the changes in technology to the hardware interface world.

In my opinion, don't change anything about *Circuit Cellar*. It's the last great hope to us aging hardware junkies who rely on you and your team to help guide the way through uncharted waters. And don't let whatever happened to *BYTE* happen to *Circuit Cellar*.

James J. Aschberger
jaschbrg@memphis.edu

FEATURE ARTICLE

Thomas Anderson

System-on-Chip Design with Virtual Components

Here in the Recycling Age, designing for reuse may sound like a great idea. But with increasing requirements and chip sizes, it's no easy task. Thomas explains how virtual components help suppliers get more mileage out of their SOC designs.

Design reuse for semiconductor projects has evolved from an interesting concept to a requirement. Today's huge system-on-a-chip (SOC) designs routinely require millions of transistors. Silicon geometry continues to shrink and ever-larger chips are possible.

But, the enormous capacity potential of silicon presents several challenges for designers. Design methodology and EDA tools are being severely stressed by SOC projects at the same time that narrowing time-to-market requirements demand more rapid and frequent introduction of new products.

SOC projects present another problem—how to design enough logic to fill up these devices. Few companies have the expertise to design all the intellectual property (IP) needed for a true SOC, and few have enough engineering resources to complete such a massive project. Even those with the required knowledge and plentiful resources may still be unable to finish a complete chip design in time to meet accelerated market demands.

The net result: SOC projects require design reuse. Only by leveraging off past designs can a huge chip be completed within a reasonable time. This solution usually entails reusing designs from previous generations of products and often leverages design work done by other groups in the same company.

Various forms of intercompany cross licensing and technology sharing can provide access to design technology that may be reused in new ways. Many large companies have established central organizations to promote design reuse and sharing, and to look for external IP sources.

One challenge faced by IP acquisition teams is that many designs aren't well suited for reuse. Designing with reuse in mind requires extra time and effort, and often more logic as well—requirements likely to be at odds with the time-to-market goals of a product design team.

Therefore, a merchant semiconductor IP industry has arisen to provide designs that were developed specifically for reuse in a wide range of applications. These designs are backed by documentation and support similar to that provided by a semiconductor supplier.

The terms "virtual component" and "core" commonly denote reusable semiconductor IP that is offered for license as a product. The latter term is promoted extensively by the Virtual Socket Interface (VSI) Alliance, a joint effort of several hundred companies to set standards for VC design, verification, and use. In this article, I describe the major virtual component (VC) types and discuss their use in SOC designs.

FORMS OF VC

VCs are commonly divided into three categories—hard, soft, and firm. A hard VC or hard macro is a design that is locked to a particular silicon technology. Such macros are fully placed and routed and are available in a fixed size and format.

They can be easily dropped into the floorplan for a chip in the same target technology, because the silicon technology is known, and they usually have predictable timing. However, they can't easily be mapped to another silicon vendor (e.g., a second source) or even to a different technology from the same vendor. The VC user also has little or no choice in terms of feature set modification or customization.

Hard macros are most often provided by ASIC and FPGA vendors as part of their library. Such macros are a natural extension to the basic cell library used to implement the VC user's design. Because the silicon vendor sells chips, there's no incentive to provide a VC in a more portable form that makes it easier for the customer to switch to another supplier.

Some IP vendors, especially those supplying microprocessor and DSP designs, also provide a VC in hard form. This option shows that the key elements of processors, especially data paths for arithmetic computation, are often designed at the transistor level for maximum performance.

Some processors, as well as many other kinds of VC products, are available from IP vendors in soft (or synthesizable) form. A VC described in Verilog RTL or VHDL code gives the user maximum flexibility. It can be mapped to virtually any target ASIC or FPGA technology using commercial logic synthesis tools.

The user may also be able to control the VC feature set, for example, by setting variables in the code or by running a utility that modifies the code under user control. Of course, because the user licenses the actual Verilog or VHDL source code, it can always be modified directly.

One issue with a synthesizable VC is that the precise timing is not known until the VC is mapped to a target technology. Accordingly, soft VC suppliers must synthesize to a range of representative target libraries and ensure that timing is satisfied.

The supplier may also have to supply guidelines to assist the user in laying out the chip containing the VC so that the postroute timing is still correct. Such guidelines may include recommendations for target technology, pin assignments for external I/O, floorplanning for key modules, and routing of critical paths.

The definition of a firm VC varies widely. The term is used most commonly to refer to a soft VC accompanied by an example layout-level implementation, although some people refer to a VC as firm whenever it comes with layout guidelines. The term also refers to a netlist-level VC that has been mapped by synthesis to a target technology but is not yet placed and routed.

It is possible, although difficult, to make customizations to a VC in netlist form. Synthesis tools can also provide some degree of portability to new technologies, but the range of optimizations available when synthesizing from the netlist level is more limited than from Verilog or VHDL.

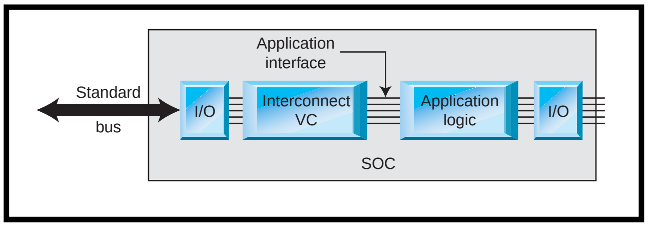


Figure 1—*A soft VC interconnect fits between the application logic and the I/O signals. Implementation instructions generally include guidelines for connecting the interconnect to the external chip pins.*

VC FUNCTIONS

Numerous factors can lead to a decision to license a commercial VC for inclusion in an SOC design. The expertise and resources available and the time-to-market requirements for the end product must be balanced against the expense of the VC license. Even a company with vast, expert resources may be able to produce a better product faster by leveraging external IP.

This is especially true if the VC implements a common function because little is gained by designing and optimizing such a function rather than focusing on product-differentiating features. For example, the VC may duplicate the function of existing stand-alone chips (e.g., a UART or a floppy disk controller) or implement a common arithmetic function such as a multiplier.

Perhaps the highest leverage is provided by a VC that implements a formal or de facto standard. Because many types of chips and end products must meet a standard, a VC that implements this standard is ideal as a commercial IP product. It's rare that an end user can add enough value with an in-house design to offset the time savings and standards expertise embodied in a well-designed VC.

The standards provided by VCs range from formal IEEE, ANSI, and IEC specifications to new technologies. Examples include communications protocols like Ethernet and ATM, computational functions such as MPEG and JPEG, parallel interconnect standards such as PCI and AGP, and serial interconnects like USB and IEEE 1394. These examples have wide applicability to many different types of SOC-based products, and the standards themselves are well enough defined to allow implementation as a VC.

A VC implementing an interconnect technology probably has the widest range of application. For example, PCI is used in diverse types of electronic products. Although it was developed as a personal computer peripheral bus, PCI has now been adopted for workstations, mainframe computers, military applications, and networking/telecommunications systems. USB is following a similar expansion of scope beyond the PC, as it is used to connect peripherals to gaming systems, set-top boxes, and PDAs.

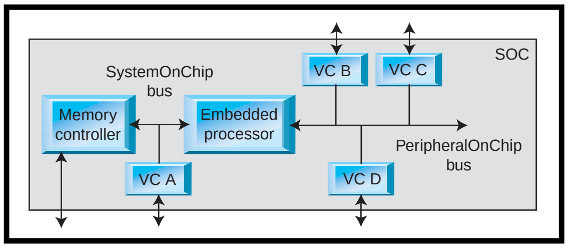


Figure 2—*System-on-chip designs may contain both a system bus connect and a peripheral bus connect. Custom I/O blocks that provide functions not commercially available may also be included.*

The widest penetration of all may occur with 1394, which is designed to interconnect both computers and diverse consumer electronics devices. Products available today with 1394 support include video cameras, digital televisions, digital VCRs and professional audio equipment. Although it is not yet supported in mainstream PC chipsets,

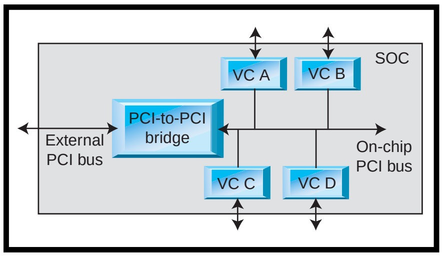


Figure 3—*A multiprotocol I/O controller can be enabled using PCI as an on-chip bus. In this application, a PCI-to-PCI bridge supports multiple I/O technologies using a single PCI load or slot.*

many desktop and laptop computers offer 1394 support and such peripherals as disk drives and videoconferencing cameras are starting to appear.

USING A VIRTUAL COMPONENT

The path for a chip designer to use a VC depends on both form and function. A hard macro can be dropped into a chip layout fairly easily, as long as the macro and chip use the same silicon technology. But, simulation and timing analysis with a hard macro is not as simple.

Generally, the VC supplier must provide separate simulation and timing models. Correlation of these models to the hard-macro implementation may be a difficult problem for the VC provider and a potential issue for the user. A VC at the netlist level has fewer problems, although the inefficiency of gate-level simulation may require the VC supplier to provide a high-level model in addition to the netlist.

A soft VC has several advantages in terms of design flow because it can usually follow the same design process as the rest of the chip. The user runs synthesis to map the RTL design to the target technology, uses static timing analysis to verify timing, lays out the chip following the supplier's guidelines, and reruns static timing analysis with back-annotated postroute delays.

The VC also has the same advantages as any RTL design in that the source code also serves as the simulation and timing model. The lack of perturbation to the user's design methodology is a key attraction for a synthesizable VC.

A VC with no requirements for connection to chip pins, such as a fully embedded processor, is wired into the chip design like any other module. An interface VC, however, generally has some I/O signals that need to connect to external chip pins. The implementation and layout instructions for a soft interconnect VC generally include guidelines on how to connect to the pins.

As shown in Figure 1, such a VC essentially fits in between the chip pins and the user's application logic. The set of VC I/O signals to which the user connects is often referred to as the application interface.

MULTIPLE VC APPLICATIONS

It's becoming common for an SOC design to use more than one VC. Although there may be no direct interaction between one VC and another, in other cases they may be linked on a common bus. The term "on-chip bus" (OCB) describes a formally specified bus that interconnects multiple VC blocks within a single chip.

An OCB is likely to fall into one of two categories—system or peripheral bus. A system bus connects an embedded processor or DSP with the memory controller and higher speed I/O devices. A peripheral bus connects to lower speed I/O technologies.

An interface block generally bridges these two buses, although some embedded processors directly drive both buses. In SOC designs with multiple embedded processors, the processors generally communicate over the system bus.

Figure 2 shows an SOC that has both system and peripheral OCBs. In an actual chip, the system bus might link to a 400-Mbps 1394 interconnect VC and the peripheral bus would support slower I/O technologies such as USB, RS-232, and IrDA (infrared). It's also possible for the SOC designer to create custom I/O blocks that connect to an OCB to support functions not available from commercial VC sources.

It would be nice if widely adopted OCB standards existed, but this is not the case. Nearly every embedded processor has its own proprietary system bus; some have defined proprietary peripheral buses as well. This situation can make it difficult to take a VC designed for an SOC with one embedded processor and move it to a different chip. A few buses (e.g., AMBA buses for ARM processors) are supported widely enough to be considered a de facto standard in some application spaces.

One interesting option for a peripheral OCB is an on-chip version of PCI. Several popular embedded processors are available in versions that provide PCI support, and many interconnect VC families include an option for PCI support on the application interface. Using a PCI OCB also enables existing PCI-based chips to be easily transformed into macros for use in larger SOC designs in newer technologies.

The size of a PCI VC, which usually ranges from 7k to 15k gates, is not a major issue in the context of a million-gate SOC. Other objections to PCI as an OCB (e.g., its use of tristates and multiplexed address/data lines) can be addressed by using a PCI derivative.

In fact, a number of SOC designers use PCI or a derivative bus as an OCB. Figure 3 shows one interesting application, a multiprotocol I/O controller.

This design allows multiple I/O technologies (e.g., USB, 1394, and Ethernet) to be combined by using a VC with PCI for each and then using a PCI OCB to link the VCs together. A PCI-to-PCI bridge permits this wide range of I/O support while using only a single PCI load on the motherboard or a single PCI slot in the system.

VSI is tackling the OCB issue by defining the virtual component interface (VCI), a standard application interface for VC designs done in-house or available from commercial IP suppliers. VCI is not an OCB but a standard VC interface that enables OCB usage.

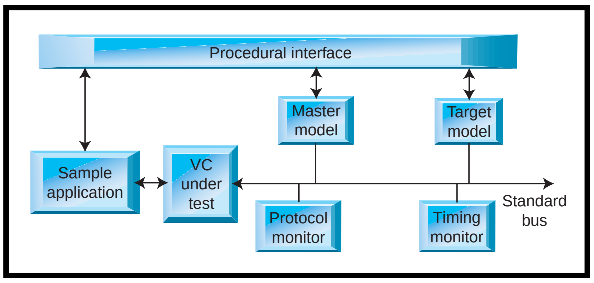


Figure 4—*A verification environment provides behavioral models and test scripts to verify the functionality of the VCs. This approach can be used in full SOC verification as well as stand-alone VC verification.*

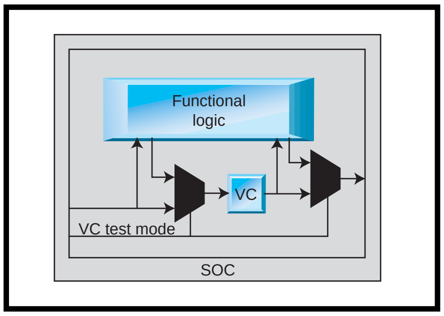


Figure 5—*One test method for legacy VCs is a parallel access test process. This process uses multiplexers to bring all inputs and outputs to the external pins and plays a predefined set of test vectors.*

The idea is that the SOC designer needs to develop only a bus translator from VCI to the chosen OCB. With this translator, any VCI-based block can easily be connected to a given OCB. VCI has been defined for easy translation to popular OCBs (including PCI) and translators for such buses will be available as licensable IP.

VC VERIFICATION ISSUES

One area common to both single-VC and multiple-VC SOC designs is the need to verify and test the complete chip. The SOC design and test engineers want to leverage and build on the verification and test done for each individual VC, and accordingly they expect the VC provider to assist in this process.

As previously noted, each VC is accompanied by some sort of simulation model that enables the SOC designer to run chip-level tests that involve the VC. But, that doesn't provide any support for determining whether the VC is connected properly in the design and is operating correctly in the context of the full SOC. A VC with a miswired application interface will simulate, but the results may not be correct.

Many suppliers address this problem by providing a verification environment along with the VC itself. Such an environment provides behavioral models (usually in Verilog, VHDL, or C) that interact with the VC and a set of test scripts that use these models to verify the functionality of the VC. Figure 4 shows a sample verification environment for an interconnect VC such as PCI or 1394. Key components of this approach include:

- master model to address the VC as a target
- target model to respond to VC as a master
- sample application code to stimulate VC
- tests written as scripts of procedural calls
- monitors for protocol and timing correctness

As a stand-alone test for the VC, a verification environment lets you verify a postsynthesis netlist or a customized version of the VC to ensure that protocol and timing rules are satisfied. A simple set of test vectors performs much the same function for stand-alone VC verification, but debug is harder without monitors and readable test scripts.

One of the main advantages of the verification environment approach is that many of its components can be used in the full SOC verification in addition to the stand-alone VC verification. The master and target models can be connected to the SOC bus interface while the protocol and timing monitors can continue to be used.

It may be possible to continue to run the same test scripts on the SOC, requiring the chip designer to modify the procedural interface to stimulate the VC from the actual SOC logic rather than from the sample application.

A verification environment can be provided with any VC, whether in hard or soft form. Synthesizable-VC suppliers usually provide verification environments and hard-macro suppliers often provide some components such as bus models to aid in SOC verification.

Many of these components are useful to a designer developing a custom implementation of an interface or processor. So, it's quite common for IP providers to license a verification environment even to customers who do not license the VC itself.

VC TEST CHALLENGES

By its nature, a VC is embedded into the SOC design by the VC user. Once a VC is inside the larger chip design, it's no longer accessible as a stand-alone functional block. Whatever test vectors or methods the VC supplier provides can no longer be used without special considerations to design-for-test (DFT) approaches during chip design.

The challenges of embedded VC tests depend on the nature of the VC itself. A synthesizable VC is the easiest case. The VC user runs synthesis to map the Verilog or VHDL code to the target technology, lays out the chip following the supplier's guidelines, and runs timing analysis with back-annotated postroute delays.

The result, as I noted, is that the VC follows the same design process as the rest of the chip. The same is generally true for chip test methodology, since virtually all test insertion tools run on the postsynthesis netlist. Whatever approach the VC user takes for the rest of the chip—full scan, partial scan, built-in self test (BIST), or JTAG—is usually applied to the soft VC also.

To ensure that the user has no problems, a soft-VC supplier should use a clean design style with DFT in mind. Typically, soft VC designers use simple clocking schemes, avoiding latches and gated clocks unless they are required.

For example, the USB protocol has suspend and resume commands that put a peripheral device into a minimal-power state. Some amount of clock gating is unavoidable in a USB device VC because of this requirement.

In contrast, a hard macro user is stuck with whatever test technique (if any) is built into the VC. If the VC includes full scan, partial scan, or BIST technology, it's helpful if the remainder of the chip also uses this approach. Integrating a VC scan chain into the full-chip scan chain is usually a simple matter of running scan insertion and stitching tools.

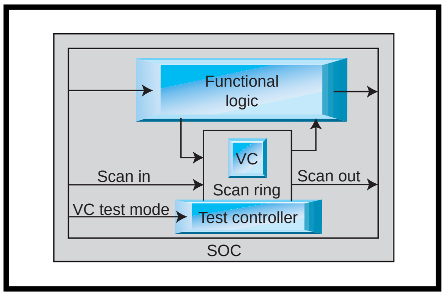


Figure 6—*Another approach for testing legacy VC involves serial access to the virtual-component I/O signals. The legacy VC is surrounded with a JTAG-like register chain that can drive VC inputs and read VC outputs. The disadvantage of this method is that test times can be very long for complex blocks.*

Sometimes the hard macro includes no internal DFT at all (often called legacy VCs because the user needs to treat them as black boxes in terms of testing). Generally, the VC supplier provides a set of test vectors, perhaps guaranteed to provide a certain level of coverage as defined by the single stuck-at fault (SSF) model. Of course, running this exact set of tests on the VC once it's embedded in a chip can be a challenge for the VC user.

When the only test method available for a legacy VC is "playing" a set of predefined test vectors, two approaches are common. The first is simply to bring all VC inputs and outputs out to external chip I/O pins using multiplexers as shown in Figure 5.

The parallel test-access process can be automated by test-insertion tools and requires only a VC test-mode pin setup prior to running functional vectors on the VC. This approach is attractive for interconnect VCs like PCI because some VC inputs and outputs will already be connected to chip I/O pins for functional reasons.

This method breaks down if there are more VC inputs and outputs than chip pins available. Staging registers may be needed to accrue each complete VC vector over several clock cycles. If the test vectors are also intended to check VC timing, inserting multiplexers into the path adds delays and may require changes to the timing vectors.

This approach doesn't work at all for analog VCs unless some sort of analog multiplexer is available. Intervening digital logic makes it impossible to apply or measure continuous analog values.

The second approach, called internal boundary scan or VC isolation, surrounds the legacy VC with a JTAG-like register chain that can drive the VC inputs and read the VC outputs. As shown in Figure 6, this setup requires some form of TAP-like test controller to run the scan chain.

Because this technique relies on serial access to the VC I/O signals, test times can be long for complex blocks like embedded microprocessors. So, IP suppliers are developing methods to test complex VCs using existing functional datapaths, including on-chip buses.

CORES IN FPGA DEVICES

As I mentioned, it's common for FPGA and ASIC vendors to provide hard macros for common core functions. Traditionally, these offerings have been limited, but the increasing speed and size of FPGAs means a broader scope of core offerings.

For example, 33-MHz PCI cores are readily available, and some FPGA vendors even claim fully-compliant 66-MHz cores. Such cores require a great deal of hand-tuning during the design and layout stages so they are optimized for a particular technology.

Like their ASIC counterparts, FPGA designers may desire flexibility and portability and can therefore benefit from synthesizable designs. There's no reason that synthesizable cores can't be targeted to FPGA designs, but mapping a core to an FPGA technology does present challenges.

In general, commercial synthesis tools are less efficient at mapping to complex programmable logic blocks than to relatively simple ASIC cell libraries. The effect of routing length is usually greater for FPGAs, producing unanticipated delays on critical paths.

The design-tool flows for most FPGA vendors do not have a tight loop from layout back to synthesis. So, the synthesis process usually can't take into account useful layout information (e.g., a chip floorplan specifying the location of timing-critical blocks.

The result: less correlation between the preroute timing estimates from the synthesis tool and the accurate post-route timing results. When it comes to final chip timing, surprises are usually negative rather than positive.

Finally, the gate capacity of even the largest FPGA devices is far below that of ASICs and custom chips, which limits opportunities for multicore designs. So, on-chip buses aren't common in FPGAs.

Combinations of a few cores are possible in large programmable devices: for example, including several Ethernet cores to implement a network repeater, or pairing a PCI core and a USB host core for an adapter chip to add USB to a PC without chipset support.

WORKS IN PROGRESS

Design reuse, including licensing of commercial IP, is key to SOC design. And a significant industry has arisen to provide a wide range of VC products. Several industry initiatives are addressing the needs of VC suppliers and users.

The IEEE Test Technology Technical Committee Embedded Core Test Study Group has also been working on VC test issues, and this has led to the proposed IEEE P1500 specification.

It's important not to minimize the issues and concerns involved in VC use. Recognizing this, two industry groups focus on the business and legal aspects of VC license and use.

Many VC suppliers are members of the RAPID trade association, which works on common VC license agreements and catalog methods. RAPID cooperates with the Virtual Component Exchange (VCX), which is developing a structure for simplified VC transactions.

The immaturity of EDA tools for VC integration and SOC design is one challenge to design reuse. Also, new suppliers may underestimate the difficulty of designing for reuse across a diverse customer base. Some VC types (e.g., interconnect cores, embedded processors) need a vertically integrated supplier that supports software and hardware.

Despite these issues, virtually every major system and semiconductor manufacturer has licensed external IP and employs internal reuse. Commercial VC products are incorporated into thousands of chip designs, and the many successful products on the market prove the value of this approach. ◼

Thomas Anderson is director of engineering in the Semiconductor IP Group at Phoenix Technologies. He is also a member of the PCI special interest group steering committee and chairperson of the 1394 Developers' Conference. You may reach him at tom_anderson@phoenix.com.

REFERENCES

T.L. Anderson, "Interconnect Solutions for Embedded Systems," *Microsoft Embedded Review*, 1 Mar. 1999.

T.L. Anderson, "The Reality of Using Cores as Virtual Components," *Electronic Engineering*, July/Aug. 1998.

T.L. Anderson, "Make vs. Buy: A Case for Licensing Cores," *Silicon Strategies* **1**, Feb. 1998.

T.L. Anderson and C. Stahr, "ASIC Design Flow with Synthesizable Cores," Proc. of Design, Automation and Test in Europe: User's Track, Feb. 1998.

T.L. Anderson, "Thoughts on Core Integration and Test," Int'l Test Conference 1997 Proc., Nov. 1997.

T.L. Anderson, "CPLD Cores can Speed Design Turnaround," *EE Times*, 21 Apr. 1997.

T.L. Anderson, "The Challenge of Verifying a Synthesizable Core," *Computer Design*, July 1996.

T.L. Anderson, "RTL Cores Promote Design Flexibility," *EE Times*, 15 May 1996.

S. Brandt, "Building a Multimedia Video Conferencing Chip with a Synthesizable PCI Core," *Integrated System Design*, Sept. 1997.

S. Davidmann, "Using Pre-designed Components in PCI Bus-based Systems," *Electronic Engineering*, May 1996.

IEEE, "A D&T Roundtable: Testing Embedded Cores," *IEEE Design and Test of Computers*, May/June 1997.

IEEE P1500 Standards for Embedded Core Test, grouper.ieee.org/groups/1500.

A. Oldham, M. Knecht, and T.L. Anderson, "IP Cores in AGP Bus-based Systems," *Electronic Product Design*, May 1998.

SOURCES

VSI Alliance
(408) 256-8800
Fax: (408) 356-9018
www.vsi.org

RAPID
(408) 341-8966
www.rapid.org

Virtual Component Exchange
+44 1506 404100
Fax: +44 1506 404104
www.vcx.org

FEATURE ARTICLE

John Andrews & John Day

Debugging, In-Circuit Style

Looking for a tool that will shorten the development and debug cycle? Ask John, or John, and they'll point you to in-circuit debuggers. Listen in as they compare emulators, simulators, and other tools that can reduce your time to market.

Thanks to the amazing Turing machine, embedded-system designers can create an endless array of different end products using the same simple, finite instruction set of their choice.

This freedom means that a generic tool that supports the features of a specific processor can address the needs of developing a flight-control system or a toaster oven. For those of us in the business of creating flight-control systems, toaster ovens, or any other electronic systems, this is a good thing.

Arguably, the strongest advantage of basing a design on programmable digital logic, as opposed to fixed function digital or analog devices, is the ability to bring these generic tools to bear. There's no doubt that, in practice, every system must be brought to life in a test bed as unique as itself. Thanks to the theoretical advances of people like Mr. Turing and the practical experience of generations since, a lot of the work has been done for us.

The standard tools of the trade for embedded-system design range from purely software monitors and simulators to custom silicon for in-circuit emulation. The main objective of all these tools is to provide a window into the operation of the software inside the embedded processor.

A software simulator is independent of the hardware under development, and an in-circuit emulator seamlessly replaces the target processor for maximum hardware control. Other tools range between these approaches in cost and performance. The benefit of a tool in shortening the product development cycle increases with its level of integration with the end-product hardware.

SIMULATION

Many embedded processors are supported with software instruction simulators. Some of these only simulate instruction execution. Most offer breakpoints, which allow fast execution until a specified instruction is executed. Many also offer trace capability, which shows the instruction-execution history.

All simulators provide read and write access to the internal processor registers and control of simulated program execution. Although instruction simulation is useful for algorithm development, embedded systems (by their nature) require access to peripherals, including I/O ports, timers, ADCs, PWMs, and so on.

More advanced simulators (such as MPLAB-SIM) implement many peripheral features including I/O pins, interrupts, as well as status and control registers. These peripheral simulators help you verify timing and basic peripheral operation.

They provide various stimulus inputs, ranging from push buttons connected to I/O pin inputs, to logic vector I/O input stimulus files, to regular clock inputs and internal register value injection for simulating A/D conversion data or serial communication input. Many embedded systems can effectively be debugged using the proper peripheral stimulus.

Simulators also offer you the lowest cost development environment. In many cases, they're available free of charge. Peripheral implementations help debug peripheral interaction and are more effective than instruction-set-only simulators.

Unfortunately, it's rather difficult to simulate all possible external conditions, so many real-time systems are tough to debug with simulation only. Also, simulators typically run at speeds 100–1000× slower than the actual processor, so long timeout delays must be eliminated when simulating.

IN-CIRCUIT EMULATION

In-circuit emulators (ICEs) offer real-time code execution, full peripheral implementation, and breakpoint capability. High-end emulators also offer real-time trace buffers, and some will timestamp instruction execution for code profiling. Emulators plug into your target system in place of the embedded processor.

ICEs are sometimes implemented with a special ASIC or FPGA that imitates the core processor code execution and peripherals. Although this approach can yield an emulator that supports more processor families, behavioral differences between the actual device and emulator can crop up.

Some manufacturers have lock-stepped the behavior of emulated and real processors by designing special bond-out emulation devices. These devices use the same circuit technology as the target processor and provide the emulator access to the internal data registers, program memory, and peripherals. This process is accomplished by eliminating the processor's internal program memory and providing this memory through emulation RAM.

The microcontroller firmware is downloaded into the emulation RAM, and the bond-out processor executes these instructions while using the same data registers and peripherals as the target processor. The I/Os of the bond-out silicon are made available on a socket that's plugged into the system under development instead of the target processor being emulated.

Emulator systems provide the most direct connection between the user interface host and the system being developed. Direct I/O and peripheral access is provided by the bond-out chip. The emulation RAM supports fast "single button" code revision downloads.

State-of-the-art emulators provide additional aids, like multilevel conditional breakpoints and instruction trace including date- and timestamp. For example, the MPLAB-ICE 2000 trace analyzer permits system debugging without halting the processor.

With development tools, like anything else, you get what you pay for. The complexity of trace buffers, high-speed emulation RAM, and specialty bond-out chips make emulators more expensive.

Some manufacturers also place restrictions on maximum processor clock speed or operating voltage. Cabling to the system can be clumsy or cause RF interference. Also, a system with close mechanical constraints may have difficulty accommodating the emulator probe instead of the target processor.

DEBUGGING WITHOUT AN EMULATION SYSTEM

The price/performance gap between the emulation's total replacement of the target processor and interpreting hints from a software simulator leaves a lot of room for intermediate solutions. The simulator provides the basic user interface required for embedded hardware cross-development (breakpoints, single-stepping, watching variables, etc.). Its main limitation is its complete isolation from the target hardware.

If you're simulating, you're also probably using the burn-and-learn method of run-time firmware development. First, you burn a chip with a device programmer, then plug it into the target hardware and watch the system crash. After much head scratching and reproducing the symptoms in simulation, you change the source code, rebuild the executable, and burn another chip.

Without access to internal processor RAM, the program counter, and quick program-memory downloads and breakpoints, this debugging method can be inefficient, slow, and tedious. Routines can be added to dump vital debugging information to a serial port for display on a terminal. I/O pins can be toggled to indicate program flow.

Obviously, if more symptoms are provided by the target system as it runs, you can make more logical changes to the source code. Otherwise, it's frustrating to wonder why certain things are happening when you use this method.

Feature	Burn and learn	Software simulation	In-circuit simulation	In-circuit emulator	In-circuit debugger
Real-time execution	Yes	No	No	Yes	Yes
Low cost	Yes/free	Yes/low/free	Yes/low	No/high	Yes/low
Full device peripheral implementation	Yes	No/limited	Yes/limited	Yes	Yes
Automatic download of new program	No	Yes	Yes	Yes	Yes
No loss of target device I/O pins when debugging	Yes	Yes/sometimes	Yes/sometimes	Yes	No
View and modify RAM and peripherals	No	Yes	Yes/limited	Yes	Yes
Resources needed to support debugging	None	None	Serial port, some I/O pins	None	Two or less I/O, minimal program and data RAM
Simple connectivity of debugger to target	None	None	No/complex	No/complex	Yes
Real-time trace buffer	None	No/limited	No/limited	Yes	No
Single-stepping	None	Yes	Yes	Yes	Yes
Hardware breakpoints	None	Yes/unlimited	Yes/limited	Yes/unlimited	Yes/one

Table 1—*Choose the appropriate development tool by comparing the features and tradeoffs of the various options available.*

IN-CIRCUIT SIMULATORS

When simulating code where branches are conditional on the state of an input pin or other hardware condition, a simulator needs to know that state from the target hardware. Usually, you can manually provide this information in the form of simulator stimulus. If the simulator can ask the target hardware for the stimulus directly, it takes one step away from being software-only and enables a certain amount of hardware debug.

This approach is taken by the SIMICE development tool for the PIC16C5*x* 8-bit microcontroller family. SIMICE integrates the capability of the MPLAB-SIM simulator with a communication module that acts as the target processor.

The module stimulates the simulation directly from the target processor's digital input pins and enables the simulator to set binary values on the output pins. This compromise gets the simulator one step closer to the hardware and overcomes the expense of an emulator.

If a well-developed software simulator exists, providing it with a means of gathering stimulus, via communication with target hardware, makes it more valuable. But, there are a few hurdles that SIMICE doesn't overcome.

It runs at the speed of a simulator so it can't set and clear output pins fast enough to implement timing-critical features like a software UART. It also doesn't support the more complex peripheral features (e.g., ADCs, PWMs) of higher integration microcontrollers.

RUN-TIME MONITORS

One important feature present with in-circuit simulators is a communication channel between the host development system and the target hardware.

Once this channel exists, the next logical step takes execution of the target code out of the host system's simulator and cuts it loose on its native target hardware. This multiplexed execution of code under development and communication of debug information with a cross-development host is referred to as a run-time monitor.

Such tools typically consume something like a UART in the target hardware to provide the communication. The host can then issue commands to the target processor so it performs debug functions like setting or reading memory contents.

Some code-execution control is also possible. Software breakpoints are implemented by inserting `GOTO` instructions in locations where you want code execution to vector to the monitor and provide control and data to you. If the features of an existing host-system simulation tool can provide the user interface, then existing technology may be leveraged into greater functionality.

Careful comparison of the wide variety of development tools available today is the best way to figure out how to get your product to market in a timely fashion.

Software breakpoints can be built in at compile time. Other features, such as hooking into a periodic timer interrupt to copy data from the target to the host, can also be included when building the executable. These techniques don't require writing to program memory at runtime, so they can be used with burn-and-learn debug using OTP devices.

It's amazing that the time spent waiting for windowed devices to UV erase didn't get included in "crash, erase, and burn." If changes are being made quickly and only a few devices are available, the erasing can be the most time consuming. So, monitors that can be written quickly and without a UV erase cycle are often found in systems with RAM or flash program memory.

The routines that transfer debug data to the host or download a new executable (i.e., the monitor code) occupy a certain amount of program memory on the target. They also require data memory and consume some of the target processor's bandwidth in addition to the UART or other communication device.

The bandwidth consumption is largely mitigated by the fact that most of this overhead occurs when target code execution is not in progress. Uploading data values to a watch window while servicing a software breakpoint or downloading a new target code revision both occur when the target code isn't running.

A run-time monitor is a great step forward from simulating in isolation from the target hardware. But because it's a purely software tool running on the target processor, there are some limitations on how much control it can get over program execution. Adding a few simple support features in the silicon of the target processor can turn a software monitor into a system with all the basic features of an emulator.

IN-CIRCUIT DEBUGGERS

The widespread advent of reprogrammable flash program memory has made in-circuit debugger tools practical for single-chip embedded microcontrollers. In-circuit debuggers enable the embedded processor to self-emulate.

It's hard to describe a debugger's capabilities and tradeoffs of a debugger without referring to a example. So, let's take a look at MPLAB-ICD, which is based on the PIC16F877 8-bit flash memory microcontrollers and can be used to develop various PIC16C*xx* controllers. It's also a programmer for the flash PIC16F87*x* family.

MPLAB-ICD uses the in-circuit debugging capability built into the PIC-16F87*x*. This feature, along with the in-circuit serial programming (ICSP) protocol, offers in-circuit flash programming and debugging from the GUI of MPLAB's IDE.

You can develop and debug source code by watching variables, single-stepping, and setting breakpoints. Running at full speed enables you to test hardware in real time.

The in-circuit debugger consists of three basic components—the ICD module, ICD header, and ICD demo board. Your serial port connects the MPLAB software environment to the ICD module. When instructed by MPLAB, this module programs and issues debug commands to the target '16F87*x* using the ICSP protocol, which is communicated via a 9″ six-conductor cable using a modular plug and jack.

A modular jack can be designed into a target circuit board to support direct connection to the ICD module or the ICD header can be used to plug into a DIP socket. The ICD header contains a target ’16F877, a modular jack to connect to the ICD module, and provides 40- and 28-pin male DIP headers to plug into a target circuit board.

You can plug the ICD into custom hardware or use the included ICD demo board, which provides 40- and 28-pin DIP sockets that accept a ’16F87*x* device or the ICD header. The board also offers LEDs, DIP switches, an analog potentiometer, and prototyping area. Even if your hardware isn’t available, PICmicro prototype development and evaluation of the MPLAB-ICD are feasible with this board.

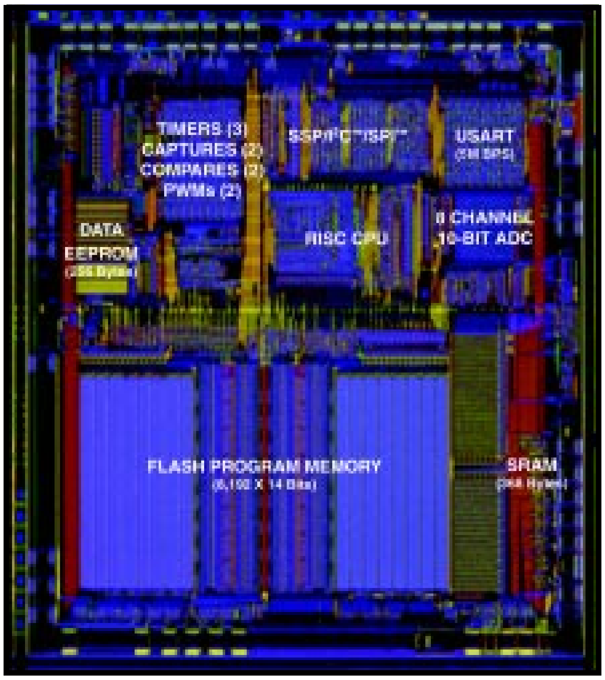


Photo 1—*The Si area that implements the ICD feature is very small compared to the major peripherals labeled in this die photo of the PIC16F877.*

RUN-TIME OPERATION

The debug kernel is downloaded along with the target firmware via the ICSP interface. A nonmaskable interrupt vectors execution to the kernel when the program counter equals a preselected hardware breakpoint address, after a single step, or when a halt command is received from the host.

As with all interrupts, this interrupt pushes the return address onto the stack. On reset, the breakpoint register is set equal to the reset vector, so the kernel is entered immediately when the device comes out of any reset.

The ICD module issues a reset to the target ’16F877 immediately after a download. So, after a download, the kernel is entered and control is passed to MPLAB running on the host.

You can then command the target processor as you choose. All RAM registers including the PC and other special-function registers can be modified or interrogated. You can single-step, set a breakpoint, animate, and start or stop full-speed execution.

Once started, a halt of program execution causes the PC address prior to kernel entry to be stored, which enables MPLAB to display (in source code) where execution halted. When you command the target host to run again, the kernel executes a return from interrupt instruction and execution continues at the address that pops off the hardware stack.

SILICON REQUIREMENTS

The breakpoint address register and comparator, along with some logic to single-step and recognize asynchronous commands from the host, make up most of what’s needed in silicon. The ICSP interface is in place to support programming and doesn’t constitute an additional silicon requirement.

When this channel is used for in-circuit debug, these I/O pins may not be used for other run-time functions. Photo 1 shows the device’s major functional blocks. The ICD uses very little Si area because it connects the existing features of the ICSP interface and ICE support with only a little extra logic.

RUN-TIME REQUIREMENTS

Besides the I/O pins of the ICSP interface, the ICD consumes several other processor resources. The debugger kernel must reside and execute in the target processor, so it consumes a small amount of what’s available:

- ~256 words of program memory
- ~10 bytes of general-purpose RAM
- one level of hardware subroutine stack
- two general-purpose I/O pins

The processor bandwidth considerations are the same as those for a run-time monitor. All debugging processing is done at a time when target code isn’t running anyway (e.g., between single steps or after hitting a breakpoint).

We’re fortunate that the days of hand-assembling source code and keying it into a bootloader are long gone. Make sure the development tool you choose has full source-level debugging and all the other modern creature comforts. Table 1 summarizes the important features of emulators, simulators, and in-circuit debuggers.

With basic run-time features of single step, hardware breakpoint, full-speed execution, and full access to all RAM and peripherals, an in-circuit debugger is a reasonable alternative to in-circuit emulators.

A five-wire cable can give a clear window into the device code execution and peripheral state. Such tools may shorten your development cycle and get your product to market sooner. ■

John Andrews has more than 15 years of experience as an embedded system hardware and software design engineer. He is currently a principal field applications engineer at Microchip. You may reach him at john.andrews@microchip.com.

Also a principal field applications engineer, John Day has worked for Microchip for six years and was previously senior hardware design engineer for Digital Equipment Corporation’s Alpha Workstations Group. You may reach him at john.day@microchip.com.

SOURCE

MPLAB-SIM/-ICE/-ICD
Microchip Technology, Inc.
(888) 628-6247
(480) 786-7200
Fax: (480) 899-9210
www.microchip.com

FEATURE ARTICLE

Gordon Dick

Induction Motors

Part 1: A Different VFD

If at first your variable frequency drive project doesn't succeed…. Fueled with the desire to make a VFD project work and armed with an embedded controller, Gordon charged through the setbacks and reached his objective at last. Here's how.

Several years ago, I started a variable frequency drive (VFD) project. The circuit came from an applications book and it produced a variable frequency squarewave whose amplitude also varied to keep the voltage-to-frequency ratio constant.

To get the variable amplitude, I built a variable supply that used a phase-controlled SCR bridge feeding an LC filter. The high-voltage electrolytic for the filter was quite expensive and the inductor was made from a microwave oven transformer rewound. The squarewave was produced by connecting the motor to a power FET bridge supplied from the variable-amplitude supply.

I have no doubt that the circuit would have eventually worked, but during the prototyping I managed to blow all four devices in the bridge drive section twice. I can't remember my exact mistakes, but the frustration and anger are still fairly vivid in my mind.

After the second blowout, the project sat on my bench for a long time and was eventually dismantled to make way for something else. It always bothered me that I let that project beat me.

The project I'm going to describe here is also a VFD, but it's implemented in quite a different way. An embedded controller is a significant component to this project, which was not the case the first time around.

That makes this project more difficult in some ways (there's all that code to create, debug, and test) but it certainly makes it more powerful. If you want to add a feature, just create the code and reblast the EPROM.

I believe I've vindicated myself. I eventually did get a VFD project to work, albeit many years later. And, in many ways, this VFD is superior to the one I first tried to build.

CURRENT STATE OF VFD

Many semiconductor manufacturers produce complete lines of power modules suitable for driving induction motors, and modules capable of driving motors in the 20- to 50-hp range are available. IGBTs are the usual power devices used in a power-module drive bridge.

A power module only gives you the muscle; there's still quite a bit of circuitry required before you have a motor-drive unit. For example, there's the drive circuits for the IGBTs as well as the circuitry to monitor the motor-

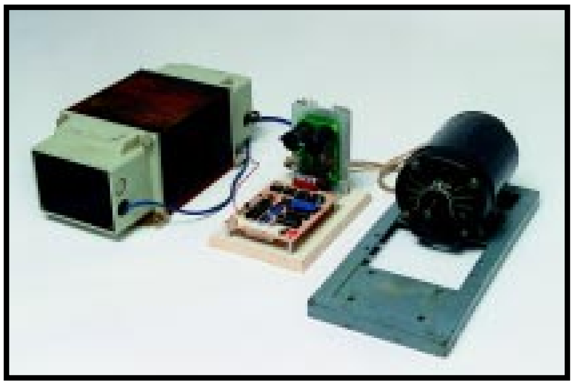


Photo 1—*Here's the complete system. The wirewrap board contains the 'HC11 and the SA828, and the speed knob is the red pot in the corner. The IRPT-1059C is mounted to the vertical heatsink. The motor is a ⅓-hp 1725-rpm unit and the isolation transformer is a 10-kVA unit—big and heavy.*

drive current, voltage, and temperature. International Rectifier produces a reasonably priced motor-drive module for three-phase motors up to 1 hp (they make modules for larger motors, too).

The IRPT1059C PowerIR-Train does everything I mentioned above and also has the rectifier/filter unit to produce the inverter supply voltage. The functions contained in the IRPT1059C are shown in Figure 1. In Photo 1, the green PCB with two large black electrolytics standing on the board mounted to a vertical heatsink is the IRPT1059C.

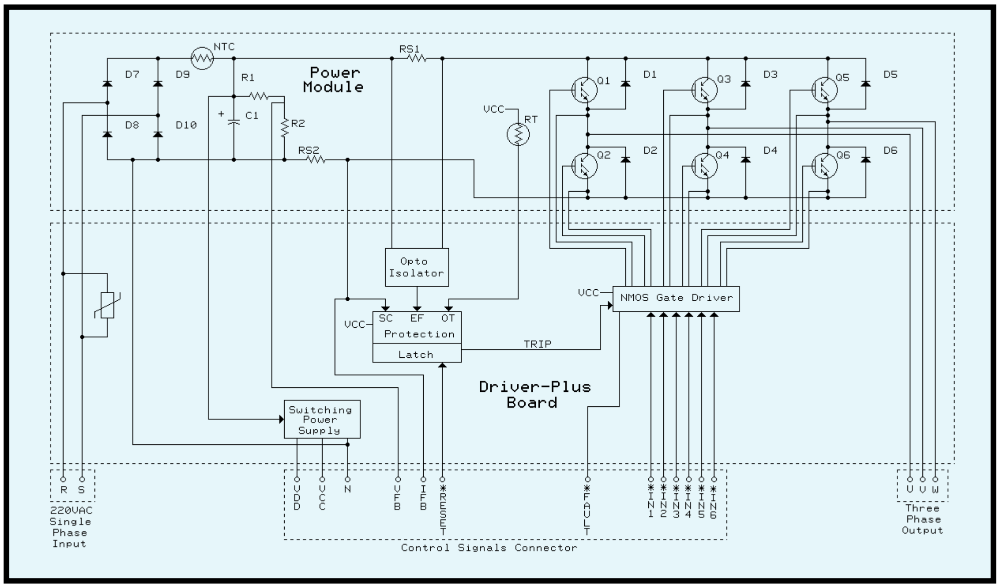


Figure 1—*IR doesn't give us a complete schematic of the IRPT1059C, but this figure shows the functions that the board implements.*

There may be other folks that make PWM waveform generators, but I came across Mitel first, and after they sent me evaluation samples, I had no need to look for others. Certainly, another device may have features that make it more attractive in a particular application, but the Mitel SA828 has an impressive list of features:

- MOTEL interface that enables it to be used with most micros
- wide power frequency range
- 12-bit speed control accuracy
- carrier frequency selectable to 24 kHz
- sinusoidal waveform data stored in internal ROM

The block diagram of the SA828 is shown in Figure 2. From the programmer's perspective, there are five registers to be written to. Unfortunately, like many micro peripherals, this device is write-only.

PROGRAMMING THE SA828 BLOCK

There are two 24-bit registers that need to be written to 8 bits at a time to use the SA828. One of the registers receives initialization data and the other receives operating data.

I won't go through the procedure for establishing the bit pattern for these registers here because it's somewhat tedious and involves lots of new acronyms (for details, see the SA828 datasheet and the code posted on the *Circuit Cellar* web site).

The code for the SA828 was built in blocks (or modules). After wiring a socket for it on my trusty 'HC11 prototyping board, I created some simple code just to verify that data was being received properly. Once I knew data was getting to the SA828 and it was creating PWM waveforms correctly, I gained some confidence.

Because the motor speed in this project is established by setting a speed setpoint potentiometer, some code is created that will read one of the 'HC11 ADC inputs (which has a pot connected to it) and convert that to an appropriate frequency value to send to the SA828. The 'HC11 has nothing else to do here but look after the SA828. I have it average the four ADC result registers before using the setpoint from the pot.

To convert the ADC number to a desired motor frequency, multiply it by:

$$\frac{90}{256}$$

and add 10. This way, you never try for zero speed. The ADC values range from 0 to 255, and the desired motor frequency varies from 10 to 100 Hz. For the 1725-rpm motor I used, this would produce synchronous speeds of 300–3000 rpm.

Before the number from the ADC can be used as desired motor frequency by the SA828, it needs to be converted to a 12-bit power frequency select word (PFS, as it's referred to in the datasheet). This conversion is done by multiplying the number from the ADC by:

$$\frac{4096}{100}$$

Keep in mind that these numbers are for this project only.

At this point, I have a routine that reads the ADC, calculates the PFS, and sends a new desired frequency to the SA828 continuously. I can verify that this routine is working correctly with a scope by checking the PWM outputs of the SA828. There's still no motor connected and won't be for a while.

When the speed of the motor is reduced below 60 Hz, the motor voltage should be reduced proportionately to keep the voltage-to-hertz ratio constant. The amplitude of the PWM voltage is established by an 8-bit amplitude select word (ASW), and maximum voltage corresponds to the ASW = 255. For frequencies below 60 Hz, this implies a relationship between frequency and amplitude of:

$$\text{amplitude} = 4.25 \times \text{frequency}$$

The 4.25 factor is convenient from a programming perspective, in that multiplication by four can be done with shifts and multiplication by 0.25 can also be done with shifts. So, the 4.25 factor can be handled without any floating-point routines or approximations. This completes the code for the SA828.

MOVING OUT

It's time to begin the move to a stand-alone system. Until now, the SA828 has been mounted on the prototyping board and all that happened was the creating and testing of code.

Now the SA828 will be mounted on its own board. In addition to that, it will get output buffers for the PWM outputs so that optoisolators, which will eventually drive the IRPT1059C, can be installed.

However, the 'HC11 on my prototyping board is still controlling the SA828. Once the wiring is complete, I'm able to verify that the SA828 is running fine on its own board. The added buffers and optoisolators are working as well.

The optoisolators I used here aren't just general-purpose devices. Because PWM switching could be happening in the 24-kHz range, I wanted to use optoisolators intended for motor-control applications. I had some HP datasheets for suitable devices and was able to get them easily. The HCPL-4506 devices work fine and are reasonably priced.

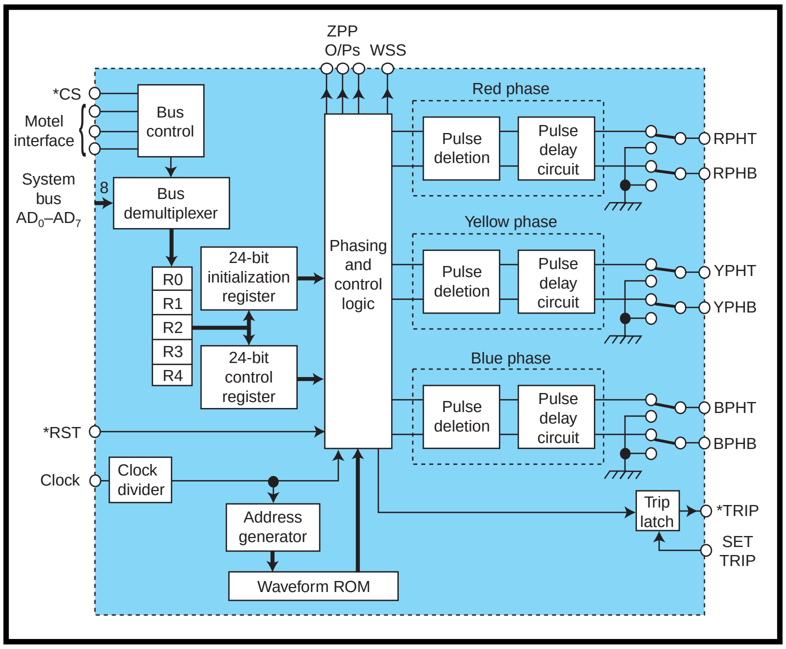
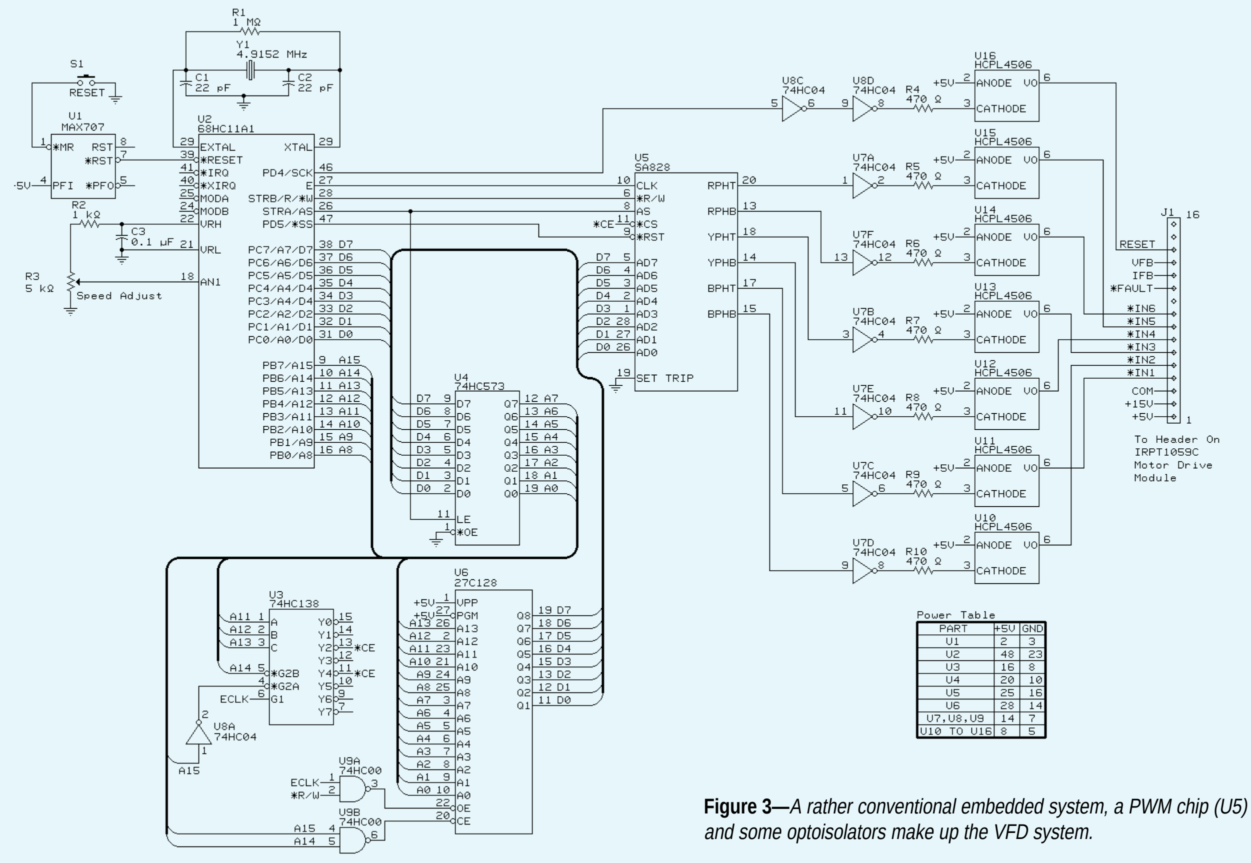


Figure 2—*This simple block diagram hides a lot of circuit complexity. The switches on the right produce the PWM drive signals for the bridge, and the 'HC11 interface pins are on the left.*

Here's where things can get interesting in several ways. It's time to connect some of the building blocks together and make a system. At best, there'll be a motor running whose speed is controlled by a pot setting. At worst, a problem will appear and when you check the motor voltage with the scope, the scope ground becomes an arc welder when connected.



PART	+5V	GND
U1	2	3
U2	48	23
U3	16	8
U4	20	10
U5	25	16
U6	28	14
U7,U8,U9	14	7
U10 TO U16	8	5

Figure 3—*A rather conventional embedded system, a PWM chip (U5) and some optoisolators make up the VFD system.*

I've seen my students do this many times when working on unisolated equipment connected to the AC power line. It's important that some care and thought be applied here to avoid equipment damage and shock hazard.

Isolation is important. You either isolate the circuit under test or the test equipment (when the prototyping phase is complete, you can remove the isolation).

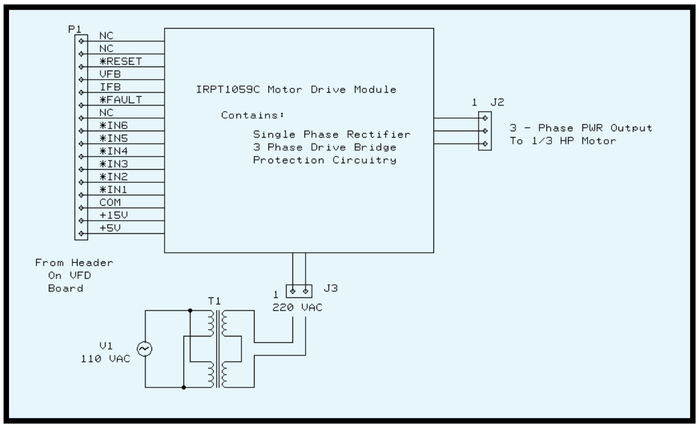


Figure 4—*The complete system wiring is shown here. The step-up transformer provides isolation, which means safety.*

In this case, it was convenient to isolate the circuit. I didn't have 220 VAC where I was working on this project so I used a heavy-duty step-up transformer to get from 110 to 220 VAC. With this increase, I got isolation. More than once when I connected the scope to check something, I was pleased that I had isolation. No shocks and no arc welding!

Some initial testing on the IRPT-1059C indicated that a pullup was required on the *RESET input. The logic high-level didn't seem to be high enough. This turned out to be a mistake.

After extensive troubleshooting and hairpulling, I found where it said in the datasheet not to add any pullup on the *RESET input. When I took out the pullup, it appeared to power up correctly. Shot myself in the foot again.

Before the IRPT1059C will operate properly, it requires a short initialization sequence. With this code executed after the previously created code and with the motor and 220 V connected, I get motor operation. Excellent! Rotating the pot changes the motor speed, and it looks like I've got functionality. The project is almost complete.

A SELF-CONTAINED VFD

Although I have a system that works, it's not a stand-alone unit. It's time to add an 'HC11 to this board and stop using the micro on my prototyping board. There are some choices to make here. Do I use the 'HC11 in single-chip mode and make use of its internal 512-byte EEPROM or do I use expanded mode and attach additional memory?

Having done an earlier project in single-chip mode (*Circuit Cellar* 92), I remembered that code development was somewhat difficult, due mostly to not having 48-pin ZIF sockets. And although you can do an amazing amount in 512 bytes, I concluded that may not be enough for later enhancements. So, I decided to use expanded mode with a 32-KB EPROM.

The 256 bytes of internal RAM on the 'HC11 will be adequate so I don't need a RAM chip on the VFD board. The memory map is also a duplicate of the prototyping board to make things as convenient as possible. After quite a bit more wirewrapping, I have an 'HC11, a 27C128, and glue logic wired onto the VFD board.

Now, how do I test that it works? There's no monitor program and serial link to a PC to help. I decided to make a simple test program that exercises the micro and memory. An interrupt-driven clock program done for another project will do just fine for the test program.

First, I got the test code to work correctly on the prototyping board, which took a couple of tries since I hadn't made any ROMable code for so long that I forgot some of the details. But, once I got it working on the prototyping board, it was time to try it on the VFD board.

I transferred the EPROM and you guessed it, it doesn't work. I was prepared for this so it wasn't a downer. There were several small snags to correct on the VFD board before it would run the test code.

The Eclk was running too slow because I had used crystal capacitors that were a bit too large, thinking it wouldn't matter much. Seems it does, so I replaced them with the proper value and the Eclk signal was fine.

The MODA and MODB lines were still open and they had to be tied high for expanded mode. Only after ringing out all the data and address bus wires did I find the error. Pin 11 of the 573 had to be grounded, not tied high. I fixed my schematic and changed it on the VFD board, and the test code works fine (see Figure 3).

All I had to do was reblast the EPROM with the code that was already working and I'd be done. But no, it didn't work. I can't explain how those couple of lines of code in the RAM version didn't come along to the ROMable version. But when that discrepancy was fixed, I had the working system (see Figure 4).

The system works much as I expected. Keep in mind that I don't have the test equipment to do detailed measurements of torque or speed on the motor, but the motor speed certainly tracked the fundamental frequency of the PWM drive signal minus the necessary slip.

The routine to vary the motor voltage does its job (demonstrated nicely as the speed is reduced). At the minimum setpoint of 10 Hz (300 rpm), the motor runs but with very little torque because the motor voltage is so low. You can stop the motor by hand and the motor friction is sufficient to inhibit the motor from getting itself going again.

IMPROVEMENT POSSIBILITIES

After the motor runs for several minutes, the body gets quite warm (without a load connected to it). As well, the noise from the switching is quite irritating. Both of these problems exist because the PWM carrier frequency is too low (about 2.4 kHz).

In Figure 3, the CLK input of U5 is fed from the Eclk signal from U2, which is 1.23 MHz when a 4.9152-MHz crystal is used with U2. Using the 1.23-MHz signal, the highest-frequency PWM carrier is 2.4 kHz.

If the signal fed to CLK of U5 is increased to the maximum of 12.5 MHz, the PWM carrier can be boosted to 24.4 kHz. That puts it well out of the audio range and should also be high enough that the switching shouldn't produce any heat in the motor.

This project begs for an LCD to provide information about what's happening with the VFD. I'd first add a display of what frequency the system is driving the motor with, which should probably be displayed in revolutions per minute rather than hertz.

This is actually a fair bit of code to create. There's the LCD initialization code. Then there's all that massaging from hex to decimal and converting to ASCII for the LCD. Maybe that 32-KB EPROM isn't too large after all.

Information about motor current and voltage would also be useful. The motor current and voltage are available as signals from the IRPT1059C. However, they are unisolated.

HP has some chips that are made just for this job. If 8 bits of current and voltage are sufficient, you can use an HPCL-7840 analog isolation amplifier followed by a stage of differential to single-ended conversion. You'd get a signal that could be fed directly to one of the ADC inputs on the 'HC11.

The code to implement this version would be fairly straightforward. But, if like the "tool man" you need more power, HP has an isolated 15-bit ADC (a two-chip solution) that would fit here nicely. Along with more bits, it interfaces to the micro via a three-wire link.

The coding here is potentially more difficult, and you still have two chips to connect. But, if you want 15 bits of resolution, you'll do it.

Another nice feature would be a soft start/stop where motor voltage is increased from zero rather than starting at maximum. A stop button would need to be added to implement a soft stop because the micro has no other way to know when a stop will occur. You can also ensure that rapid setpoint changes are translated into slow speed changes.

There's one last improvement to consider—making use of the *FAULT signal from the IRPT1059C and the TRIP input on the SA828. The *FAULT signal is activated on various current fault conditions as well as overtemperature. If the micro used this as an interrupt and shut off the PWM signals using the TRIP input on the SA828, it would be a nice protection feature. It could also check to see if the fault was removed and subsequently enable the PMW signals.

I imagine some of you are more interested in controlling the speed of single-phase induction motors rather than three-phase motors. You can build your own motor-drive module or buy one.

Mitel has that base covered, but IR doesn't. Mitel offers the SA838 single-phase PWM waveform generator, but IR doesn't have motor-drive modules for single-phase applications.

Next month, I'll explain some of the improvements and additions I've made to this VFD project and the lessons I learned while implementing them. ■

Gordon Dick is an instructor in electronic technology at the Northern Alberta Institute of Technology, Edmonton, Alberta, Canada. He consults occasionally in the area of intelligent motion control and is an avid woodworker. You may reach Gordon at gordond@nait.ab.ca.

SOFTWARE

The complete source code for this article is available via the Circuit Cellar web site.

SOURCES

SA828
Mitel Semiconductors
(619) 675-3400
Fax: (619) 675-3450
www.mitelsemi.com

IRPT1059C
International Rectifier
(310) 322-3331
Fax: (310) 322-3332
www.irf.com

HCPL-4506
Hewlett Packard
(800) 235-0312
(408) 654-8675
Fax: (408) 654-8575
www.hp.com

FEATURE ARTICLE

Kenneth Ciszewski

Flash Memories Do Double Duty

Before your micro leaves home, you want to make sure its bags are all packed. But what if the software doesn't fit into the PROM, EPROM, or EEPROM? Flash-memory technology offers a cost-efficient solution for those tough design requirements.

In the beginning, there was PROM, then EPROM, and then EEPROM—various kinds of storage for microcontroller software. EEPROM was designed to be modifiable on-the-fly but was, and is, expensive. Now there's flash-memory technology, which is an improvement over these technologies because it costs less and provides in-circuit programmability.

I recently worked on a project using a 68HC11ED0 microcontroller (a ROM-less part). The first design requirement for this project was that the microcontroller software had to be reloadable from an external PC via serial port. Second, front-panel control settings needed to be stored and recalled from time to time by the microcontroller using onboard memory to change the system functionality.

The third requirement was that the front-panel control settings had to be saved via serial port to a PC. And lastly, different sets of front-panel settings needed to be loaded into the system via serial port from a PC.

The first requirement immediately suggested the use of flash memory because EPROMs aren't in-circuit reprogrammable. For the other requirements, I could have used any of the serial EEPROMs available (I^2C or SPI interface) because the microcontroller has an SPI port and plenty of I/O pins that could be used to bit-bang a synchronous serial interface. Rather than add an extra part, I decided to store the front-panel control settings in another part of the flash memory.

Most flash memories have fairly large partitions or sectors (4+ KB), and because you must program a complete sector at one time, they require a large SRAM or DRAM buffer to hold the data to be programmed.

Although these flash memories work well with PCs and workstations, they aren't readily adaptable for use with the many 8- and 16-bit microcontrollers that have small quantities of SRAM. Fortunately, some flash memories have very small sectors.

The Atmel AT29C256 is a 32K × 8 flash memory that has 64-byte sectors. The Atmel AT29C010A is a 128K × 8 flash memory with 128-byte sectors. Both are 5-V-only parts (no extra 12-V regulators required) and have become reasonable in price. I estimated the amount of software and data that the system would require and decided to go with the AT29C010A.

SYSTEM CONFIGURATION

Placing the flash memory in the microcontroller address space takes planning (see Figure 1). The interrupt vectors of the 68HC11 (which has a 64-KB address space) start at $FFD6, and the internal registers and SRAM may be placed on any 4-KB boundary in the memory map.

In this design, the registers are left at $0000 and the SRAM is placed at $1000 at startup by writing to the Config register. I placed the flash memory at $8000. At this location, it extends beyond the normal 64-KB address space.

To use the entire memory, I used two Port A output pins on the 'HC11 connected to address lines A15 and A16 as page selects (see Figure 2). This gave four 32K × 8 pages in which to store software and front-panel settings.

The requirement for reloading the microcontroller software meant that I

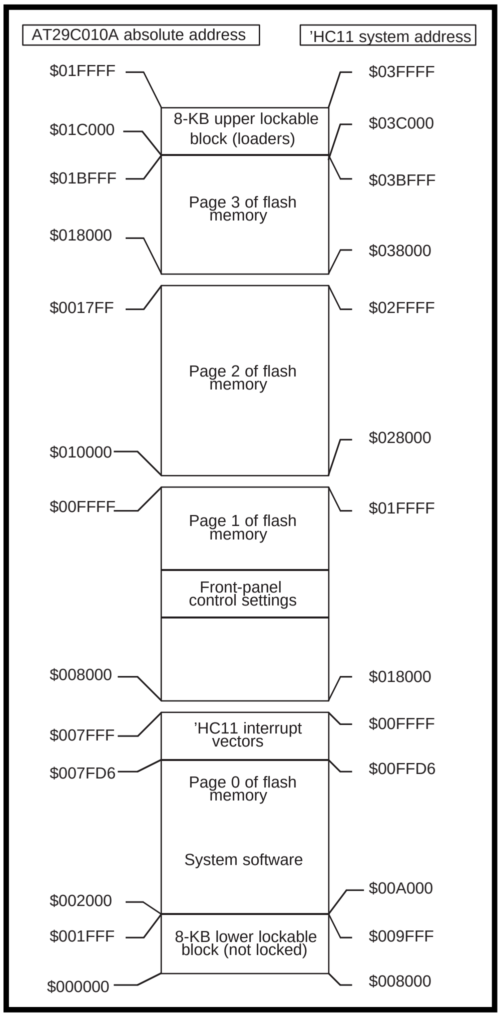


Figure 1—*This memory map shows the absolute addresses of the flash memory that are used by the device programmer and, also, the addresses seen by the microcontroller.*

needed a place to store loader programs. The AT29C010A has two memory areas, the lowest 8 KB and the highest 8 KB, that can be individually block-locked after programming. Once block-locked, these areas cannot be changed ever again.

I selected the top 8 KB to hold the loader software because it will always be the same, once developed and debugged. And once locked, it can never overwrite itself. This ensures the ability to always reload the system software.

I also included initial front-panel default settings to permit the unit to be set to a baseline control functionality, should the saved settings become trashed or otherwise unsuitable.

The initial software and front-panel settings can be programmed into the flash memory using a standard flash-memory programmer. But, a word of caution is in order.

Like most programmable-memory manufacturers, Atmel recommends certain device programmers whose programming algorithms have been validated by them. Data I/O, BP Microsystems, and System General were three device-programmer brands that were recommended by my friendly Atmel FAE. Because of the complexity of the flash-memory algorithms, problems can occur when using equipment whose algorithms are not validated.

I first used a Brand X programmer. There were many system failures and lockups, and the system didn't want to start consistently when powering up. I finally discovered that the software data (sector) protection (discussed later) wasn't being properly implemented by the device programmer and the flash memories were overwriting themselves in random places when the system power was turned on and off.

Worse yet, this condition was intermittent. Sometimes, the memories would be fine. Other times, they would die as soon as I turned the system on or off. Switching to a validated programmer solved this problem.

Another word of caution: on many device programmers, the software data protection must be specifically enabled in a configuration menu before you program the device, despite the apparent fact that the device will not operate reliably in circuit unless this is done.

It seems strange that enabling the software data protection should be a device-programmer option. Perhaps this is because the Atmel datasheet doesn't plainly state that enabling the protection is required.

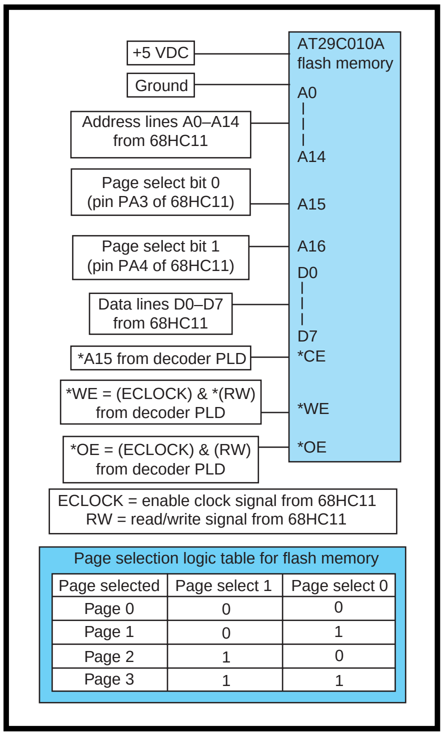
Another issue that arose was that the 68HC11 assembler didn't recognize addresses above $FFFF, although the address space of the flash memory goes beyond that value. So, I had to write a simple program to take separate S-record (`.S19`) files (one for each of the four pages of the flash memory) and combine them into a single S-record (`.S19`) file with proper addresses and line checksums that can be recognized by the device programmer.

This translation program was written in Visual Basic for DOS 1.0 and enabled me to place the contents of pages one, two, and three at arbitrary points in the flash memory. Page zero was assumed to start at $8000. Because the microcontroller was also bootloading software instructions to a DSP that was part of the system, the translation program also allowed placement of the DSP software at a specified location in page zero.

Using Visual Basic for the translation program permitted the creation of a stand-alone executable file for the PC that could be used in both batch and `make` files processed by the Borland Maker utility and used to control assembly and linking of the software.

JUMPING BETWEEN PAGES

One challenge was figuring out how to jump between flash-memory pages without crashing the system. When the software writes the change-pages instruction to Port A, the microcontroller suddenly finds its program counter pointing to whatever the next location would be, but on the new page.


+5 VDC
Ground
Address lines A0–A14 from 68HC11
Page select bit 0 (pin PA3 of 68HC11)
Page select bit 1 (pin PA4 of 68HC11)
Data lines D0–D7 from 68HC11
*A15 from decoder PLD
*WE = (ECLOCK) & *(RW) from decoder PLD
*OE = (ECLOCK) & (RW) from decoder PLD
AT29C010A flash memory
A0
A14
A15
A16
D0
D7
*CE
*WE
*OE
ECLOCK = enable clock signal from 68HC11
RW = read/write signal from 68HC11
Page selection logic table for flash memory

Page selected	Page select 1	Page select 0
Page 0	0	0
Page 1	0	1
Page 2	1	0
Page 3	1	1

Figure 2—*The flash memory connects directly to the microcontroller's address and data buses. The control lines *CE, *OE, and *WE are created by a programmable logic device.*

I needed a way to tell the microcontroller where it should go on the new page, which usually doesn't have any particular relation to where it was on the old page. This was accomplished by loading into the 'HC11 SRAM the code that actually changes the page and then jumping into the SRAM to execute the change.

At this point, the software is at a known address in SRAM and can jump back out of SRAM into the new page of flash memory to any address chosen. The page-change code can be loaded into SRAM at boot-up or on-the-fly as required. Loading on-the-fly allows different entry points to different routines on the different pages as required (see Figure 3).

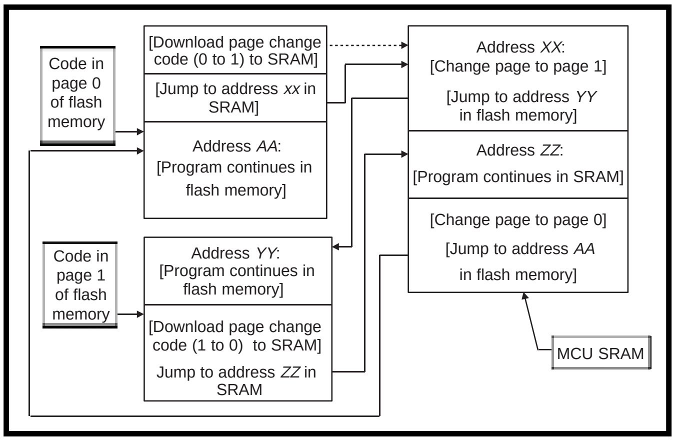

Code in page 0 of flash memory
[Download page change code (0 to 1) to SRAM]
[Jump to address *xx* in SRAM]
Address *AA*: [Program continues in flash memory]
Code in page 1 of flash memory
Address *YY*: [Program continues in flash memory]
[Download page change code (1 to 0) to SRAM]
Jump to address *ZZ* in SRAM
Address *XX*: [Change page to page 1]
[Jump to address *YY* in flash memory]
Address *ZZ*: [Program continues in SRAM]
[Change page to page 0]
[Jump to address *AA* in flash memory]
MCU SRAM

Figure 3—*Changing pages in flash memory requires loading the change-page software into microcontroller SRAM so the change can be executed without crashing the system.*

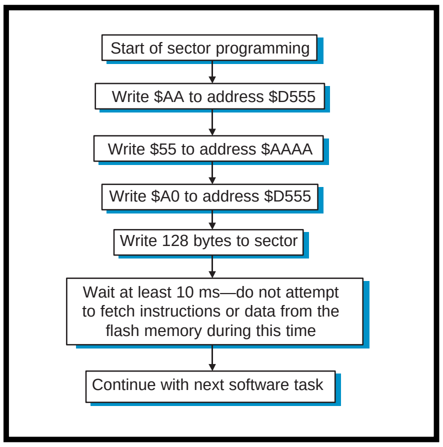

Start of sector programming
Write $AA to address $D555
Write $55 to address $AAAA
Write $A0 to address $D555
Write 128 bytes to sector
Wait at least 10 ms—do not attempt to fetch instructions or data from the flash memory during this time
Continue with next software task

Figure 4—*Programming a flash-memory sector is done by writing software protection codes and 128 bytes of data to the memory, and then waiting while the memory moves the data into its flash cells.*

STORING SETTINGS

Writing to the AT29C010A is wonderfully simple. The memory has 1024 128-byte sectors. Address lines A0–A7 point to the byte within the sector, and A8–A16 point to the sector within the device. The memory has chip-enable (CE) and write-enable (WE) pins similar to those found on SRAM devices.

The micro writes three special codes to three special addresses for software data (sector) protection. It then writes 128 bytes (starting at byte 0 of the sector) into a buffer in the flash memory.

The micro waits about 10 ms for the data to be internally programmed into the flash-memory cells. During those 10 ms, the contents of the memory cannot be read or written, so it's advisable for the software to be running in SRAM so the system doesn't crash!

The datasheet describes how to monitor certain data lines to find out when the programming is complete. Instead, I waited about 14 ms to be safe and then continued on with the next sector write or next software task. This worked well.

To invoke the software protection, the three special codes must first be written in order to the specified addresses. The entire sector of 128 bytes must also be written; otherwise the software protection won't work (see Figure 4). Failure to follow these requirements may prevent the software data (sector) protection from working and cause the flash memory to overwrite itself randomly during system power-on transitions.

Also, note that the addresses used for the data protection given in Figure 4 are for a flash-memory base address of $8000. The flash-memory datasheet gives different addresses for a flash memory having a base address of $0000. The special addresses change with the base address of the flash memory.

I assigned one sector to each group of settings being saved. Because I only needed 12 bytes for the setting values, the remaining bytes of the sector were filled with $01, which is the `NOP` opcode for the microcontroller. Of course, there are ways to pack the data more tightly into a sector, but with all the memory space available, it was simpler to do it this way (see Figure 5).

The process of saving settings starts when the software writes the "save" routine into the microcontroller SRAM and jumps into the SRAM to run it. The software changes pages by setting the Port A page select lines, writes the software data protection codes to the special addresses, transfers the bytes to be saved from an SRAM buffer into the flash memory, and loads the remaining 112 unused bytes with $01. It then waits 14 ms before changing back to the memory page it was previously operating in.

Next, the software exits the SRAM and continues to run out of the flash memory. Listing 1 shows `SECLD`, which writes a sector to the flash memory, and also `LDSECLD`, which loads `LDSEC` into MCU SRAM before it runs (see Figure 6). The process of recalling settings is similar, except there's no need for the 14-ms delay because flash-memory reads incur no long delays.

RELOAD FROM EXTERNAL PC

One of the goals of this design was to use a standard PC program like Procomm or HyperTerminal as the loader software on the PC for design and testing and later for updating software in finished units. My previ-

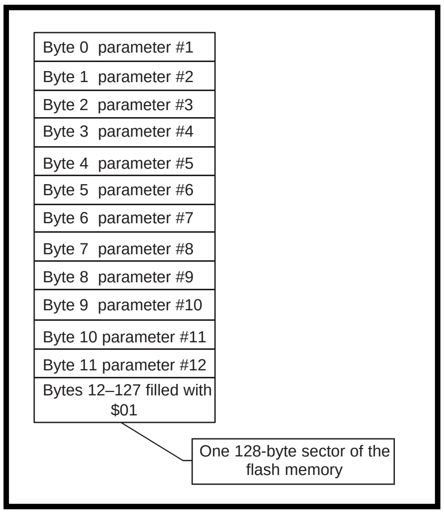


Figure 5—*Saving front-panel settings is simplified by using a complete 128-byte sector, even though the usable data is only 12 bytes.*

Listing 1—*The subroutine `SECLD` saves a sector of data to flash memory. `SECLD` is loaded into SRAM by the subroutine `LDSECLD` before being called.*

```
* SECLD subroutine to save a sector of data to flash memory
* ENTRY: values to be saved to flash memory in buffer "FPLDBUF"
*   (128 byte buffer in SRAM)
*   page value (0,1,2,3) in variable  "MPAGE"
*   address of flash memory sector to be loaded in memory location
*   "CSECTOR"
* EXIT: contents of  "FPLDBUF" stored in flash memory at address
*   "CSECTOR"

SECLD LDX #FPLDBUF  ;point to SRAM buffer containing code/data values
  LDY CSECTOR       ;point to base address of sector in flash memory
                    ;to be loaded

* Go into MCU SRAM to write to flash memory
  JMP SRAMSP        ;JMP to code in MCU SRAM
RAMJSR              ;beginning of code in MCU SRAM

* Code for security data protection enable for AT29C010A
; send codes to temporarily unlock memory sector protection
; then put it back in force
; addresses are based on $8000 base address of flash memory

  LDAA  #$AA
  STAA  $D555
  LDAA  #$55
  STAA  $AAAA
  LDAA  #$A0
  STAA  $D555
RPPAGEM LDAA  #MPAGE;change to the page of flash memory where
                    ;buffer is to be written
        STAA  PORTA ;is to be written
LDAB  #$7F          ;set ACCB (AS COUNTER) for 128 bytes
NBFL  LDAA  0,X     ;get byte in MCU SRAM
STAA  0,Y           ;store in flash memory IC (buffer)
INX                 ;increment pointers
INY
DECB                ;have we moved all the data?
BGT NBFL            ;no, do it again
                    ;yes, we're done

  LDY #$0C00      ;minimum 10-ms delay to let flash internally save
DCT DEY             ;the buffer of data just written to it (~14 ms)
  CPY #$0000        ;at 1.5-MHz ECLOCK rate
  BGT DCT           ;end of timeout
  LDAA  #PAGE3
  STAA  PORTA       ;set memory back to page 3, where the loader is
  JMP EPRJSR        ;jump back into EPROM
EPRJSR NOP          ;this code is  back in EPROM
RTS                 ;end of routine SECLD

* LDSECLD routine to load SRAM portion of SAPRHM
* ENTRY:  N/A
* EXIT: contents of subroutine SECLD from RAMJSR to EPRJSR stored
*   in MCU SRAM starting at location SRAMSP

SRAMSP  EQU $1040   ;start of SRAM program space for part of SECLD

LDSECLD
LDX #RAMJSR;point to beginning of code (in EPROM) to be transferred
  LDY #SRAMSP       ;point to beginning of SRAM space

RTRANS  LDAA  0,X
  STAA  0,Y         ;transfer bytes
  INX
  INY
  CPX #EPRJSR       ;are we finished transferring?
  BLT RTRANS        ;no, continue transferring
RTS                 ;yes, we're finished
```

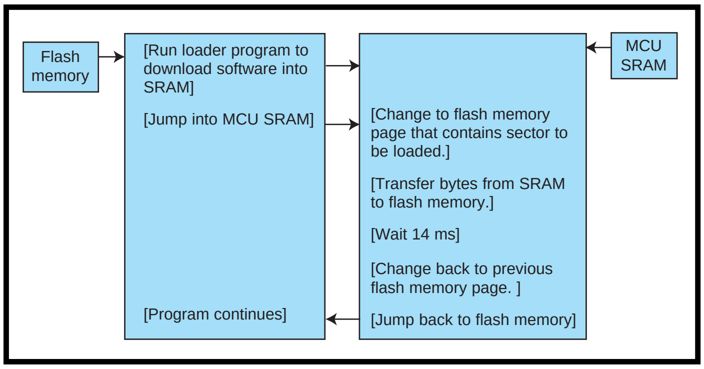

Flash memory
[Run loader program to download software into SRAM]
[Jump into MCU SRAM]
[Program continues]
MCU SRAM
[Change to flash memory page that contains sector to be loaded.]
[Transfer bytes from SRAM to flash memory.]
[Wait 14 ms]
[Change back to previous flash memory page.]
[Jump back to flash memory]

Figure 6—*Storing front-panel settings requires putting the storage software in microcontroller SRAM, and then reading the front-panel control values and putting them into flash memory.*

ous experience with 68HC11 evaluation boards suggested that this was possible at rates up to 9600 bps.

Because of the 10-ms time period required to program the flash-memory sector, it was necessary to set a delay time between the transmission of each line of the S-record file. Fortunately, both Procomm and HyperTerminal have a way to do this.

The loader software receives one line of an S-record file and calculates the checksum of the line contents. If it matches the checksum sent as the last two characters of the S-record, the S-record's data portion is programmed into the appropriate sector. Otherwise, an error message is sent back to the PC, and loading stops.

A couple tricky issues had to be resolved to make this software loader work smoothly. For one, the flash memory had to be programmed on 128-byte sector boundaries, but S-record addresses aren't predictably aligned on any particular address boundaries. Because of the limited SRAM buffer, I decided the sector alignment needed to be done before the data was sent to the micro.

Neither Procomm nor HyperTerminal knows how to align the data, so I wrote a preprocessor in Visual Basic 1.0. This solution also solved the second issue—the fact that S-record files often have variable-length record (line) lengths with no defined length limit. Preformatting the S-record file into a convenient record length answers the need for a limited-size S-record.

DUMPING SYSTEM SOFTWARE

Dumping the complete system software to a file on a PC is the reverse of loading it. Here, the microcontroller must create the S1, S2, or S9 string that starts each line of an S-record.

It arbitrarily sets the length of the S-record, which consists of the number of character pairs comprising the address, code/data, and checksum fields. It must also calculate the checksum, which is the one's complement of the sum of the values of the length, address and code/data fields (see Figure 7).

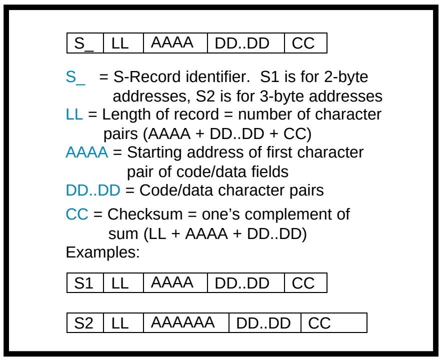
The resulting S-record file has the same record lengths and format as the file that loads the system software. This common format allows for dumping software, examining and modifying it, and loading it back into the system.

SAVING TO AN EXTERNAL PC

Saving the front-panel settings is similar to dumping the system software, except there's less information to transfer. The design originally called for 36 front-panel settings to be saved, although this was later increased to 100.

I decided to use the S-record format for the front-panel settings so they could be included as part of future software releases. This arrangement lets users change the system software but keep their front-panel settings.

Restoring the front-panel settings to the system is similar to reloading the system software, but again, there's less data to transfer. My goal was to enable users to create multiple files of front-panel settings that they can restore when desired. It also lets them share these settings with other system users.



S_	LL	AAAA	DD..DD	CC

S_ = S-Record identifier. S1 is for 2-byte addresses, S2 is for 3-byte addresses
LL = Length of record = number of character pairs (AAAA + DD..DD + CC)
AAAA = Starting address of first character pair of code/data fields
DD..DD = Code/data character pairs
CC = Checksum = one's complement of sum (LL + AAAA + DD..DD)
Examples:

S1	LL	AAAA	DD..DD	CC

S2	LL	AAAAAA	DD..DD	CC

Figure 7—*S1 records are for two-byte addresses, whereas S2 records are for three-byte addresses.*

DOUBLE YOUR PLEASURE

Many 8-bit microcontroller systems can benefit from the ability to save acquired data, user configuration parameters, front-panel settings, and other information that changes as these systems work in the real world. They can also benefit from the ability to have their operating software changed without being taken apart and having their stored program memory replaced.

Small-sectored flash memories enable system designers to use a single device for storing changing parameters and updating system software. The additional software required to accomplish these functions is straightforward. ■

Kenneth Ciszewski has been designing electronic equipment for communications systems for more than 20 years. He has designed digital audio systems for networked military flight trainers and currently designs digital audio products for musical applications. You may reach him at ciszewsk@slme.com.

REFERENCES

Atmel, *AT29 Flash Memories*, App note AN-1, 1997.
Atmel, *AT29C010A Flash Memory*, Datasheet, 1997.
Atmel, *Programming AT29 Flash Memory*, App note AN-3, 1997.
Motorola, *M68HC11 E Series Technical Data*, V.1, Datasheet, 1995.
Motorola, *M68HC11 Reference Manual*, V.3, 1991.
Motorola, *M68HC11ED0 Technical Summary*, Datasheet, 1993.

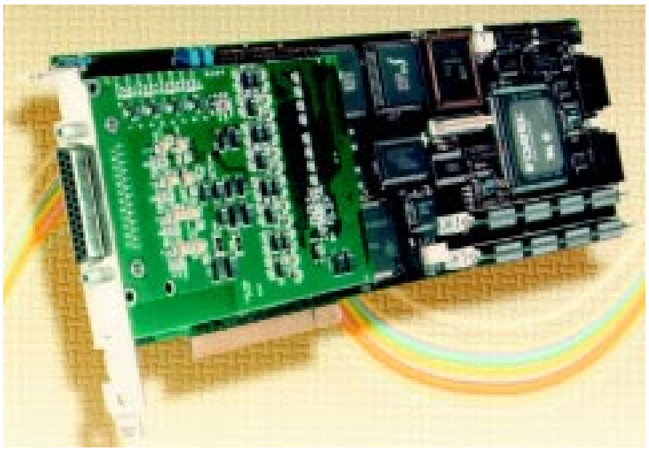
SOURCES

AT29C010A, AT29C256
Atmel Corp.
(408) 441-0311
Fax: (408) 487-2600
www.atmel.com

68HC11ED0
Motorola Semiconductor Products
(512) 502-2130
Fax: (512) 502-2123
www.mot-sps.com

DSP COPROCESSOR BOARD

The **PCI-431E** is a high-performance, 16-bit, four-channel analog I/O DSP coprocessor board for the PCI bus that offers true multilevel concurrent coprocessing. Applications include fast Fourier transform (FFT) applications, spectral analysis, simulation, digital filtering, communications systems, receivers, analytical instruments, vibration testers, robotics, and modeling/simulation.



The PCI-431E includes several subcontrollers so that the DSP is not constantly burdened servicing each A/D sample. Instead, the DSP is free to process blocks of math while the A/D section continues automatic sampling and storage. A fully programmable frequency synthesizer provides high-resolution A/D clocking or the user may supply external clocking.

Frame triggering uses either an internal programmable timebase or external signals. The board's architecture offers modes such as ring-buffered pretriggering for transient analysis applications or analog-level triggering from an onboard 12-bit, 200-kHz DAC. The DAC may also be used as a general-purpose analog output.

The PCI-431 family of boards is fully integratable with the Hyperception RIDE advanced signal processing software for Windows 95 and NT. RIDE offers graphical (nontext) "connect the blocks" programming and lists several hundred functions including FFTs, digital filters, matrix processing, and such.

Datel also offers a low-level Windows C code library for programmers (model **PCI-431 WINS**, comes with full source code, **$1295**). The binary executable version of this library (no sources) is included free with the board. The product includes a one-year warranty.

The PCI-431E is priced from **$3995**.

Datel, Inc.
(508) 339-3000
Fax: (508) 339-6356
www.datel.com

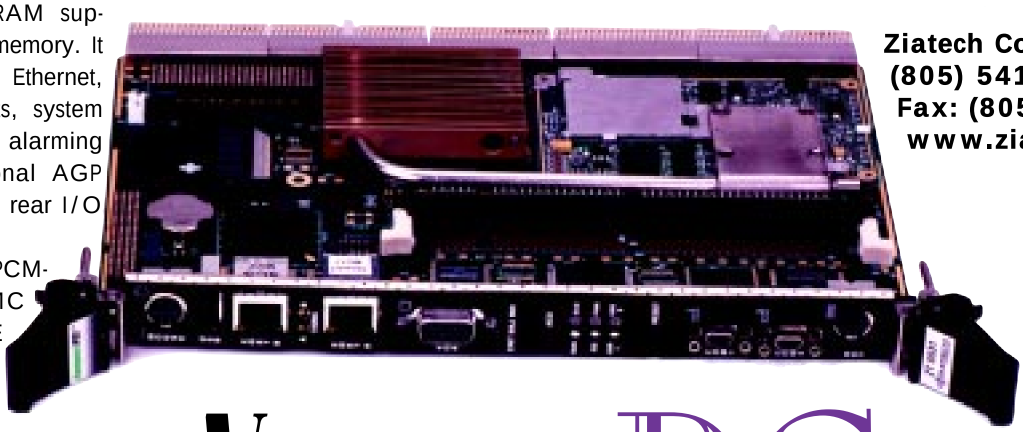
CPU BOARD WITH PENTIUM PROCESSOR

The Ziatech **ZT 5531** is a 6U CompactPCI processor board designed for telecom, datacom, and industrial control applications. The board features a Pentium II Processor Mobile Module at speeds up to 300 MHz.

It supports the hot-swap standard adopted by the PCI Industrial Computer Manufacturers Group (PICMG) and drives up to 14 CompactPCI peripherals.

Its single-slot, onboard feature set includes L2 cache, ECC synchronous DRAM support, and flash memory. It also offers dual Ethernet, dual serial ports, system monitoring and alarming functions, optional AGP video, USB, and rear I/O connections.

Optionally, PCMCIA, dual PMC modules, IDE hard drive, and floppy drive support are available in various combinations.



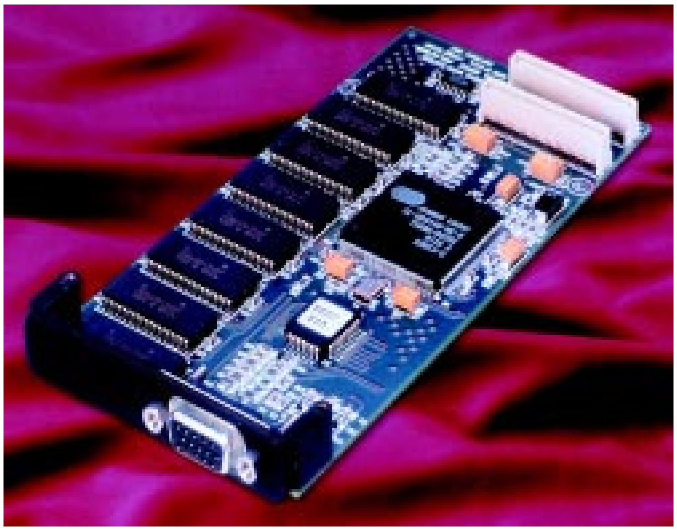
This single-board computer includes Ziatech's industrial Embedded BIOS and flash disk. Additionally, it can be configured with the user's choice of MS-DOS, Windows NT, or VxWorks. Optional development toolkits are available to support the implementation of these operating systems on CompactPCI.

Pricing for the ZT 5531 starts at **$2625**.

Ziatech Corp.
(805) 541-0488
Fax: (805) 541-5088
www.ziatech.com

Nouveau PC

edited by Harv Weiner



HIGH-RESOLUTION VIDEO MODULE

DaVinci is a rugged, high-resolution video display and graphics module designed for embedded systems equipped with PMC (PCI mezzanine card) expansion slots. It supports display resolutions of up to 1280 × 1024 with 16M colors and features a 64-bit BitBLT graphics engine equipped with either 2 or 4 MB of RAM. The module is ideal for embedded applications such as factory automation, imaging, and telecommunications.

DaVinci can add high-resolution video to baseboards or add a second video port to baseboards already equipped with video. This option is ideal for applications that require two monitors. For example, in a CAD application, designers typically use one monitor to work on their drawing and one to display the CAD tool menus. Similarly, in a video-editing application, one screen can be used to display the editing toolbox and the other used to display the source footage and output.

DaVinci features a 32-bit PCI bus interface with bus master capability and meets all IEEE-P1386 specifications for single-slot operation. It also provides a standard 15-pin DIN video connector, which eliminates the need for custom cable assemblies. The module comes ready to run with an onboard video BIOS and drivers for the Windows NT, VxWorks, QNX, and the Real I/X OSs. It complies fully with the VESA standard for monitor timing and operates from a single +5-V supply.

The module costs **$295** in single-piece quantities.

General Micro Systems, Inc.
(909) 980-4863
Fax: (909) 987-4863

PROGRAMMABLE CONTROLLER/TOUCHSCREEN

Z-World has introduced a C-programmable controller with a built-in touchscreen display. The **PK2600** includes a 320 × 240 (¼ VGA) graphics LCD with adjustable contrast and CCFL backlighting. It's ideal for control systems that require an interactive graphic interface.

The PK2600 features 16 protected digital inputs and 16 high-current sinking outputs, eight 12-bit analog input channels, and three serial ports for RS-232/-485. The unit can support up to 512 KB of flash memory or 1 MB of SRAM and includes 32 KB of VRAM. A PLCBus expansion port enables the addition of extra I/O such as relays or DAC channels. The hardware- and software-controlled contrast permits easy viewing from all angles.

Both the controller and display are C-programmable using Dynamic C. A DIP switch on the enclosure allows selection of the component to program.

The PK2600 developer's kit contains all the hardware tools necessary for rapid development: manual, schematics, programming cables, AC adapter, sourcing high-current driver, and mounting hardware.

The PK2600 sells for **$696** in quantities 100.

Z-World, Inc.
(888) 362-3387
(530) 757-3737
Fax: (530) 753-5141
www.zworld.com

Nouveau PC

Stewart Christie

Beyond the 'x86 1-MB Memory Address Limit

Pushing the code envelope is part of the job, and usually you can cram it all in. Still, if you really need that extra room (but not badly enough to justify using the latest Pentium), grab Stewart's keys to fly past the 8086/186 limits.

The 'x86 architecture is prevalent throughout the electronics industry. Although you may have a Pentium III processor in your home or office desktop PC, the latest and greatest chipset is often not economical for an embedded system. Many designers have cost or size limitations that preclude the use of even the cooling fan, never mind the PCI chipset. Others just don't need the bandwidth.

This article shows how to get that little bit more out of an 8086/186 design. I'll discuss a novel method of breaking the 1-MB address limitation of these processors. Originally developed to work with CAD-UL's real-mode compiler, it has been enhanced to use Microsoft's real-mode C compiler, which ships with Visual Studio V.1.52. This flexible memory-management method uses multiple memory banks to add extra code space for your application.

MEMORY BANKING BASICS

Traditionally, a user of banked memory systems needed to know the location of functions in a bank before it was time to compile. Moving a function from one bank to another forced a major redesign of the source code to enable the new bank. The memory banking method presented here removes this concern from the programmer.

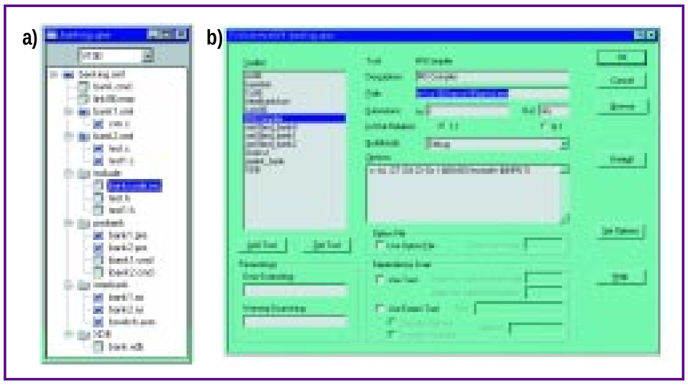


Photo 1a—The Workbench Project seen here is fully expanded. Note the different source directories for `bank1.omf` and `bank2.omf`. b—The Workbench Project file here shows the path to the Microsoft Compiler Executable and the compile command-line settings in the option window.

No changes to the source code are needed to implement this procedure.

I also present a banking-aware debugger that has been enhanced to give transparent source-level debugging of these banking applications. If you read Fred Eady's series on the SuperTAP emulator (*Circuit Cellar* 104–105), you'll recognize it. This is a ROM monitor version of the same debugger used by Applied Microsystems on their 'x86 emulators.

Because there are many ways to implement the hardware bank-switching, the code to switch the banks must be supplied by you, the hardware designer. But that's not needed here. Although my examples simulate the bankswitching with a write to the PC diagnostic Port 80H, you can use any method (e.g., setting a peripheral port bit).

GETTING STARTED

To run this example, you need an 8086 or '186 as a target system.

If you find yourself saying, "Where do I find one of these?", take heart; there are options.

You could dig out your old IBM XT, but it might not work because it probably only has a 5¼″ floppy drive. The good news is that masquerading underneath that Pentium III, with a full 128-MB of RAM, is an 8086. OK, so it's a 450-MHz 8086, but it'll do the job. Any PC will do fine as long as it has a floppy drive for the monitor boot disk.

You also need a null modem serial cable to connect the host PC to your target system. I used an Elan SC400 eval board from AMD with an external floppy drive and an old VGA card to run all of my examples.

You also need the programs, which you can download from the *Circuit Cellar* ftp site. There you'll find a complete evaluation toolset with the examples, not only for the PC, but also preconfigured for the AM186ES evaluation board. If you have the tools, just download the examples.

Because the tools are supplied as shareware, the compiled modules are limited to 1000 lines. The linker will only link 10 modules and the debugger has a symbol limit of 1000 symbols, which is sufficient to run all of these examples. If you need to evaluate a larger program, you can obtain a time-limited license via CAD-UL.

PSEUDO-BANKING

Many developers use the '186 for embedded applications, and constant "upgrading" of features will bring you closer to the 1-MB memory limit. Your options are then to select a new 32-bit processor and go back and streamline your code and hope that it fits.

Typically, 32-bit processors cost a lot more than their 16-bit cousins. A quick price check at my local semiconductor distributor's web site showed a 2:1 ratio between an Am186ER and the lowest grade Elan SC310. Obviously, the Elan device has a lot more on-chip peripherals than the '186, but if all you need is just one extra address line, then it might be cost-prohibitive to redesign a complete system.

This article addresses that need by showing how to transparently address this extra code space. You can insulate the programmers from the need to manually determine which bank their code is in.

Previous implementations required you to manually address a routine from one bank to the other and then figure out how and where to return. This method puts all these smarts into the linker and a set of

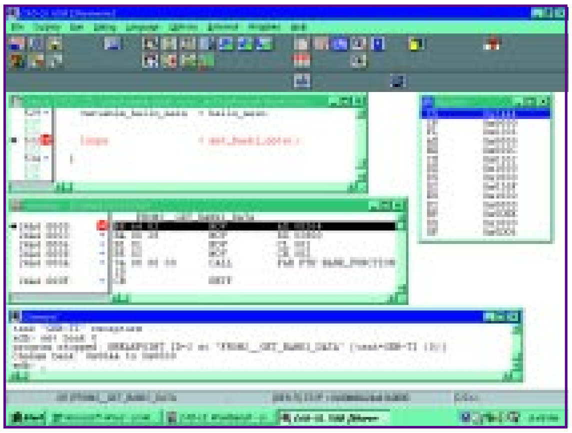


Photo 2—*This upper source window in the debugger shows the original source file and the central window shows the wrapper function with the bankswitch parameters. Note the extended address information for CS:IP in the bottom center panel.*

command files. These automatically parse the code in each bank and determine whether a direct in-bank call can be made or whether an interbank call, with its associated bank switch routine, needs to be made. You don't need to change your source code at all.

Before I describe how it works, you need some background on the application itself. You also need to know this to help you port the example to your hardware.

This example uses two memory banks, but at different physical locations under the 1-MB address. That's what I mean by pseudo-banking. There are two banks, defined as bank1 and bank2: bank 1 is at 28000–2BFFF and bank 2 is at 2C000–2FFFF.

No 'x86 processors I know of support banking hardware directly so I'm using a standard PC as the target. Writing to the diagnostic Port 0x80 on the PC simulates the banking hardware. This has the added advantage, at least on my target, of having LEDs that show the value written to the port. As well, a direct memory write to a VGA card is performed whenever a bank-switch is made, so a regular PC will show the effective bank number onscreen.

There is a common unbanked area of memory, here defined as bank0. It is located at 20000 and contains the wrapper functions for the interbank calls and a small routine (in the file `BSWITCH. ASM` in Listing 1) that drives the simulated bankswitch hardware.

THE WORKBENCH IDE

The CAD-UL tools are presented to the user in an IDE. I remember the good old days when you were glad to get a `REM` statement in a batch file, but thankfully, those days are gone.

However, the Workbench IDE and the XDB debugger are the only truly graphical tools. All the other tools, including compiler, assembler, and linker, can also be run from the DOS command line. The IDE's architecture is open so you can add your own tools, if necessary; this capability was used to add the Microsoft compiler and its options.

Photos 1a–b show the source files in the `bank1.omf` and `bank2.omf` folders. To add your own source files, select either bank and right click. Choose the Insert File to Project selection and use the browser to identify the source files. I recommend using the existing source files until you're more familiar with the tools.

LINK PROCESS EXPOSED!

Now, I'll take you step-by-step through the process of building a banked example. Figure 1 shows the steps in a somewhat easier-to-understand format.

Because the tools are compiler- and source-independent, I skipped the compile step. When you link a regular program with multiple source files and several libraries, the linker itself hides all the iterations it takes to resolve backward and forward references. But for this banking project, all the iterations are exposed.

There's not enough space here to fully explain the steps and multitude of linker commands needed to derive the final output. Suffice it to say that for *n* banks, you need 2*n*+1 linker runs and *n* runs through the assembler to generate one final output OMF file.

In step one, for all banks, generate banking symbol information: one set for the assembler and one as an input to step two. The file for the assembler run contains all the `CODE` symbols available for other banks to call (i.e., all functions defined in `test.c` and `test1.c`).

Step two generates an OMF file for the final linker run and an assembly file bank2.as that maps the calls from bank2 into any other bank.

Step three is where the remapping of function from bank-to-bank takes place. This includes bankcode.inc, a CAD-UL–supplied macro file that has the definition for the GEN_CALL macro. The -INCLUDE_LIST =... command line parameters assign bank numbers to the .cod files.

From this example, it's obvious that bank1.cod is number 1 and bank2.cod is number 2, but nothing stops you from assigning bank2.cod to number 3.

The output of this assembler run is bank2.obj. An assembler listing is included in Listing 2. Note that the bankcode.inc macros add context-sensitive comments.

The essential function of this run is to add a wrapper around the call, to get_bank1_data(). Now the call is replaced with a call to FROM2_get_bank1_data(). This routine fetches the "real" address of get_bank1_data and the destination and source bank number. It then performs a call to the user-supplied bank_function.

A similar section of code (with different parameters) is created for every interbank-called function and is placed in the unbanked section of memory. Whenever a program makes a call to, say, get_bank1_data(), then FROM2_get_bank1_data() is called instead of the "real function" in the other bank.

Step four is the final linker pass where all the different banks and the common code are linked together. The bank_function is stored in the bswitch.obj module, and that's the only common bank routine for this example.

In step five, you must now extract a symbol file and a hex file for each bank. They are loaded into the debugger separately. For production use, the hex files can be loaded into separate EEPROMs.

Annotated linker command files are included in the example with explanations of how to add, for example, a third memory bank, or how to move the memory allocation of the banks.

BUILDING THE PROJECT

If you wish to build the project with the CAD-UL CC86 compiler, then all the options are already set. Just select Project/Rebuild All from the menu or press Ctrl-A. If you want to use the Microsoft compiler from Visual C 1.52, then you need to check a few things first.

If your compiler is installed in the default location of c:\msvc\bin, everything is fine. Otherwise, select the ToolManager and correct the path for the Microsoft Compiler Tool. Photo 1b shows my settings in highlighted text. I ran the compiler from the CD-ROM with a minimal install configuration.

Once you confirm the settings, you can rebuild the project. The progress is shown in the output window.

You're almost finished with the Workbench, but before you start the debugger,

Listing 1—This code for the bankswitch routine bswitch.asm defines a stack for the banking routine and implements a pseudo-banking switch and a VGA screen write confirmation.

```
name bswitch
; bankswitch data stack
SWITCH_DATA    segment 'switch_data'
switch_stack  dd 10*256 dup (?)
switch_ptr    dw offset switch_stack
SWITCH_DATA    ends
SWITCH_CODE    segment    public    'code'
  assume cs: SWITCH_CODE
  assume es: SWITCH_DATA
; DX:AX    Function Address
; CL       destination bank
; CH       return bank

public bank_function
bank_function    proc far
;  08H[BP]  : Return segment call-function
;  06H[BP]  : Return offset call-function
;  04H[BP]  : Return segment Bankfunk
;  02H[BP]  : Return offset Bankfunk
;  00H[BP]  : old BP
  PUSH   BP
  MOV    BP,SP
  PUSH   ES
  PUSH   DI
  PUSH   AX
  MOV    DI,switch_data
  MOV    ES,DI
  MOV    DI,switch_ptr
  MOV    AX, SS:06H[BP]
  MOV    ES:0[DI], AX
  MOV    AX, SS:08H[BP]
  MOV    ES:2[DI], AX              ; save original return address
  MOV    ES:8[DI], CX              ; save return bank
  MOV    AX,SS:02H[BP]
  MOV    ES:4[DI],AX
  MOV    AX,SS:04H[BP]
  MOV    ES:6[DI],AX
  ADD    DI,10
  MOV    ES:switch_ptr, DI

;;;  add your specific bank code below here
; simulate bank switching
; no real bank switch occurs, current bank is written to VGA
; display memory to show which "virtual" bank is active
; Current active bank is stored at the diagnostic port 0x80
; and is written to the VGA RAM
  MOV    AL,CL                     ; load destination bank into AL
  OUT    080H, AL                  ; switch bank
  ;; write bankid to VGA
  call   write_bank_vga

;;; add your specific bank code above here
  MOV    SS:8[BP], CS
  MOV    WORD PTR SS:6[BP], OFFSET ret_off ; "push" return address
  POP    AX
  MOV    SS:4[BP], DX
  MOV    SS:2[BP], AX              ; "push" call address
  POP    DI
  POP    ES
```

(continued)

Listing 1—continued

```
  POP    BP
  RET                    ; call DX:AX
ret_off:
; restore call stack
  SUB    SP,8
  PUSH   BP
  MOV    BP,SP
  PUSH   ES
  PUSH   DI
  PUSH   AX
  PUSH   DX
  MOV    DI,switch_data
  MOV    ES,DI
  MOV    DI,switch_ptr
  SUB    DI, 10
  MOV    switch_ptr, DI   ; restore new 'stack' pointer
  MOV    AL, ES:9[DI]

;;; add your specific bank code below here
; simulate bank switching no real bank switch occurs,
; This is the reverse action as the bank switch above.
; The code returns to the caller bank (in AL)
  OUT    080H, AL       ; do the switch
  ;; write bankid to VGA
  call   write_bank_vga
  ;;; add your specific bank code above here
  MOV    AX, ES:0[DI]
  MOV    SS:6[BP],AX
  MOV    AX, ES:2[DI]
  MOV    SS:8[BP],AX ; return address (to origin bank)
  MOV    AX, ES:4[DI]
  MOV    SS:2[BP],AX
  MOV    AX, ES:6[DI]
  MOV    SS:4[BP],AX    ; return to caller address
  POP    DX
  POP    AX
  POP    DI
  POP    ES
  POP    BP
  RET
bank_function    endp
vga_seg    dw    0b800H

;; Write BANK <NR> to upper right corner of VGA
;; screen to show BANK-ID on PC if no Port 80
;; diagnostic card is present.
write_bank_vga proc near
  PUSH   ES
  PUSH   AX
  MOV    ES, vga_seg ; display bankid on VGA screen
  MOV    AH, 04fh     ; attribute "INVERS RED"
  ADD    AL, '0'
  MOV    WORD PTR ES:[158], AX   ; write "BANK "
  MOV    AL, ' '
  MOV    WORD PTR ES:[156], AX
  MOV    AL, 'K'
  MOV    WORD PTR ES:[154], AX
  MOV    AL, 'N'
  MOV    WORD PTR ES:[152], AX
  MOV    AL, 'A'
  MOV    WORD PTR ES:[150], AX
  MOV    AL, 'B'
  MOV    WORD PTR ES:[148], AX
  POP    AX
  POP    ES
  RET
write_bank_vga endp
SWITCH_CODE    ends
end
```

we should look at its settings. The easiest one to check is the communication parameters. The default port for connecting to your target is com1; if you need to use com2 or 3, select the ToolManager again. Click on the hammer icon, select the XDB tool, and just type in the new com port.

Now let's look at the section from the debugger batch file `bank.xbd`, a section of which is shown in Listing 2. If you haven't changed any of the linker commands, then it will work as is, but if you modify any of the memory mapping in the `bank.cmd` file, then you need to make the corresponding changes for the debugger.

WE'RE DEBUGGING—ALMOST

Before you can debug the program, you need a way to load it into your target PC. The board support package for the PC contains a prebuilt image that needs to be copied onto a blank formatted disk.

Open a DOS box and navigate to the `c:\cadul\mon86\bsps\pc\bootdisk` and run `make.bat`. This disk boots the ROM monitor and supports a null modem connection between target PC COM:1 and the host PC.

The initial connection speed of 19.2 kbps is a bit slow for any decent-sized development project. The communication speed can be changed by a command prompt, and you can see an example of the syntax in the `bank.xbd` debugger batch file.

Start the debugger by typing Ctrl-D in the Workbench or via the Tools / Debugger selection. The default invocation method displays a window with source and working directories: the initial batch file and the com port settings. You can override these temporarily, which is handy for trying out different options. If you want to make permanent changes, you need to edit the options in the Workbench ToolManager.

NAVIGATING THE DEBUGGER

When you start the debugger, it connects to your target before you can do anything, so make sure you have the serial cable inserted and the target powered up and waiting. After a successful sign-on, the batch file `bank.xdb` is loaded. This

Listing 2—Here you see the `bank2.as` assembly file (a), a section from `bankcode.inc` macro file with the definiton of the `gen_call` macro (b), and the resulting assembly listing created from `bank2.obj` (c).

a)

```
name bank2
%gen_call(_get_bank1_data)
end
```

b)

```
%*define(gen_call(name))
(extrn %name : far
%BANK_ID%()code segment public 'switch_code'
public %gen_name(%name)
%gen_name(%name) proc far
  mov  ax,offset %name
  mov  dx,seg %name            ; load far function pointer
  mov  cl,%get_bank_id(%toupper(%name)); load destination bank id
  mov  ch,%actual_bank         ; load source bank id
  call bank_function
  ret                          ; return to call
%gen_name(%name) endp
%BANK_ID%()code ends
```

c)

```
33 extrn _get_bank1_data : far
34 FROM2_code segment public 'switch_code'
35
36 public FROM2__get_bank1_data
37 FROM2__get_bank1_data proc far
38  mov   ax,offset _get_bank1_data
39  mov   dx,seg _get_bank1_data
40  mov   cl,1 ; load destination bank id
41  mov   ch,2 ; load source bank id
42  call  bank_function
43  ret        ; return to call
44 FROM2__get_bank1_data endp
45 FROM2_code ends
```

Listing 3—An annotated section from the XDB startup script file shows the extra commands defined to enable the debugging of banked applications.

```
!Enable banking extensions and define bank0 as common bank
SET OPTION /BANK=ON
SET OPT /COMBANK=0

!Load common code with symbols for bank 0
LOAD /DEB=1 OF "BANK0.BD"

!Enable bank 1 and load code and symbols
SET BANK 1
LOAD /DEB=1 OF "BANK1.BD"

!Enable bank 2 and load code and symbols
SET BANK 2
LOAD /DEB=1 OF "BANK2.BD"

!Define region that contains the bankswitch code
SET MAP/TYPE=BANK 0X20000 UNTIL 0X200FF

! Define unbanked memory region
SET MAP/ACC=SYSTEM 0X0 UNTIL 0X27FFF

!now initialize the Registers and Stack…
```

file runs through the commands in Listing 3. These commands enable the banking mode and download the separate hex files for each bank.

The center panel in the lower status line shows the extended addressing method used for banking addresses. Instead of the usual segment:offset address, there's an extra field for the bank. The address of `main()` in `crm.c` is displayed in BANKID ##CS:IP format as 0x01##0x2800:0x0000.

Because there is nothing like the protected mode descriptors available in real mode, each of the 20 bits of address information is mapped directly onto the original 1-MB address space with no translation. So, when bank1 is enabled, the code for `main()` is located at physical address 0x28000.

In Photo 2, you see the extended addressing method used for banking addresses in the center panel. I opened up the register and assembler windows to show the wrapper function for `get_bank1_data`.

DEBUGGING AT C LEVEL

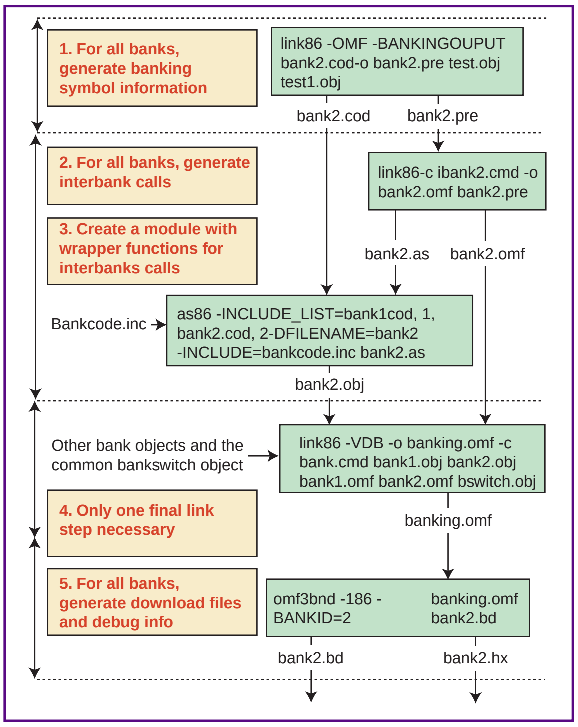
At the C source level, you have to look hard to know you're running a banked application. You can set breakpoints anywhere in the source files by selecting the file from the module window and double-clicking on the blue dot at the source file line. A stop sign is inserted to indicate a breakpoint. Right-clicking in a window brings up a context-sensitive window.

The major differences between this version of XDB and the SuperTAP emulator version is the lack of trace information and

Listing 4—Here's some code from an Am186ER banking example. Debugging the bank-switching code in assembly mode requires you to understand how XDB searches for a source code line using the execution address. If no match is found, no source is displayed. You also see this phenomenon if you break into a running program and end up in a library routine.

```
  bank 0   ;this assumes two banks selected by pcs6
      "    ;this also assumes that previously pcs6 is set up
      "    ;as an output—see applic3 for example code
      "    PUSH  DX                  ;used for call later
      "    MOV   DX,BANK_REG;
      "    IN    AX,DX
      "    AND   AX,NOT BANK_OR_MAP  ; clear bank bit
      "    SAL   CL,BANK_BIT         ; MOV 1 OR ZERO INTO POSITION
      "    OR    AL,CL
      "    OUT   DX,AX               ; CHANGE BANK
;any code after here is located in bank 1
;therefore any breakpoints here must be defined in  bank 1
   bank 1  POP   DX
```

Figure 1—To generate a banking application, these steps are necessary for all banks. This example application consists of an unbanked area and two banks, but only the steps for bank 2 are shown.


1. For all banks, generate banking symbol information
link86 -OMF -BANKINGOUPUT bank2.cod-o bank2.pre test.obj test1.obj
bank2.cod
bank2.pre
2. For all banks, generate interbank calls
link86-c ibank2.cmd -o bank2.omf bank2.pre
3. Create a module with wrapper functions for interbanks calls
bank2.as
bank2.omf
Bankcode.inc
as86 -INCLUDE_LIST=bank1cod, 1, bank2.cod, 2-DFILENAME=bank2 -INCLUDE=bankcode.inc bank2.as
bank2.obj
Other bank objects and the common bankswitch object
link86 -VDB -o banking.omf -c bank.cmd bank1.obj bank2.obj bank1.omf bank2.omf bswitch.obj
4. Only one final link step necessary
banking.omf
5. For all banks, generate download files and debug info
omf3bnd -186 -BANKID=2
banking.omf bank2.bd
bank2.bd
bank2.hx

there's no event system available for setting multilevel conditional breakpoints.

There is a breakpoint dialog to add or modify an existing breakpoint, with options to check variables and stop or continue, depending on the value. Because there's no hardware support, all of these functions are emulated in software.

A unique feature of the monitor is the ability to use Protected-Mode Debug registers. If your target is a real-mode '386 or better processor, you can set breakpoints in ROM code. Known as hard breakpoints, these can be selected by default, either in the ToolManager or via the Options dialog when starting the debugger.

NO BUGS IN THE BANKS

When you port this example to your hardware, you must check out your own bankswitch routines. If you want to single-step through the anatomy of a bankswitch, there are certain things to be aware of.

The strangest concept is the duality of the bankswitch code. When you link it, you choose bank 0, the common code section—but this is the starting bank only. Halfway through the routine, the bank is switched.

If you are single-stepping, the debugger detects the bankswitch. It tries to find a source file with the new bank address, but can't.

The second part of the bankswitch routine, beyond the actual switch which was also linked to bank0, is now being executed in whatever bank has been enabled. For this example, that's either bank1 or 2. For your development situation, however, you could have a lot more memory banks available. Listing 4 shows an example from the AMD board.

Looking for symbols in the wrong bank occurs with breakpoints, too. If you set an assembly-language breakpoint in the bankswitch code while executing in bank2 and then run the code, the debugger displays an error—unless that breakpoint occurs while still in bank2. It will search the active breakpoint list and find nothing valid at that address because the valid address is qualified by the bank2 bankID.

To get around this feature, there are commands to change the banks manually. But be careful! The quickest way to crash a program is to stop in bank1, then manually change the bank to, say, 2 and forget to change it back. When you next type RUN, it's an accident waiting to happen.

REAL-WORLD APPLICATIONS

If you intend to use banking in a real-world application, I have a few recommendations for you.

Remember not to place any interrupt handling routines in a banked memory area. The first 400H locations in the 'x86 are used for vectors and they should point to routines in the common area. You can jump from there to a banked function, if you wish. Some error handlers don't require the quick response associated with an interrupt routine, so it might make sense to place those routines in a switched bank.

A separate stack setup for the bankswitch routine is set to 2560 bytes, but no overflow checks are made. Each bankswitch takes 10 bytes of stack space. Before implementing a production release, I recommend adding a check or confirm that you never nest more than 256 bankswitches.

The current implementation passes the bank number in an 8-bit register; that's sufficient for most applications. For more than 256 memory banks, consider upgrading to a '386 or a '486 with a numeric processor.

The banking function uses the system stack for temporary variables, which works fine for a simple example like this. If you're using a multitasking OS, you need to extend the Task Control Block to add functionality for the memory banking. This is left as an exercise for the reader!

TAKING IT TO THE BANK

Of course, this project has been through several revisions. This latest incarnation is the result of much work by John Hansen, Norbert Schulz, and Robert Wiesner—not to mention a new linker revision that supports Microsoft and Borland 16-bit compilers. And it won't stop here. Banking support for a '186 emulator has been proposed, and C++ support won't be far off. So keep your eye out for additional supported hardware and new features. EPC

Many thanks to John Hansen, Norbert Schulz, and Robert Wiesner for their hard work, generosity and patience.

Stewart Christie is manager of engineering at CAD-UL in Scottsdale, AZ. His work has ranged from military test equipment and avionics power controllers to highly automated burn-in equipment for HC11's and Transputers, to the research department of a major home appliance manufacturer. Stewart has experience in proprietary and commercial RTOSs, 4-bit Micro, Space Station ASICs, and most things in between. You may reach him at schristie@cadul.com.

SOFTWARE
Software discussed in this article is available for download via the Circuit Cellar ftp site.

SOURCES
Am186ES eval board, Am486CDP eval board
Advanced Micro Devices, Inc.
(408) 732-2400
Fax: 408-894-0547
www.amd.com

SuperTAP '186 emulator
Applied Microsystems Corp.
(800) 426-3925
(425) 882-2000
Fax: (425) 883-3049
www.amc.com

Real-Mode Evaluation Suite Ro40s1xe
CAD-UL, Inc.
(480) 945-8188
Fax: (480) 945-8177
www.cadul.com/ccink

Ingo Cyliax

Where in the World...

Part 1: GPS Introduction

Whether you're dispatching emergency personnel or just trying to find your way home, affordable and easy-to-access technology is making GPS applications more popular. Ingo sets the course for future projects right here.

Recently, a friend of mine was describing how the local police station was upgrading their communication system. With the new system, each police cruiser has a GPS receiver that locates the cruiser and periodically sends its location and ID to the dispatcher. A computer then processes the messages and updates a map with the locations of all the cruisers.

My friend's comment was that they could save a lot of money by just putting a bunch of LEDs on a map to indicate the location of the local donut shops. Well, there are many other uses for GPS besides tracking police cars. Besides being used for navigation, GPS is useful in surveying, remote sensing, data collection, geology, archeology, and other applications that haven't been thought of yet.

Let's do a brief review of GPS and how it works. For a more detailed description, check out Do-While Jones' series ("The Global Positioning System," *Circuit Cellar* 77–78).

In a nutshell, the global positioning system (GPS) is a satellite navigation system with 24 satellites orbiting the earth in 12-h orbits. The satellites are distributed such that, on average, there are 12 satellites visible in each hemisphere.

Photo 1—To interface a typical hand-held GPS receiver to a computer, you have to add an external power and communication adapter module. With those additions, the unit is capable of transmitting NMEA sentences to a computer.

The satellites are time synchronized using an onboard atomic clock. They continuously transmit the time and other information using a spread-spectrum carrier. Each satellite has its own pseudo random number sequence, which makes it possible to share the same carrier frequency.

There are two carriers, one is encrypted and only usable by the military if you have the "super-secret GPS password." Data on the civilian carrier is not encrypted and can be used by anyone with a GPS receiver.

A GPS receiver is a satellite receiver that listens for the signals and measures the time of arrival, comparing it to the GPS time that is sent in the data. This information provides a pseudo-range to each satellite that is received, and that range is used to compute the position. You need to be able to receive three satellite signals to get a two-dimensional fix and four signals to get a three-dimensional fix.

GPS receivers typically listen for all 12 satellites that should be in the hemisphere where the receiver is located. The receiver uses the strongest signals for its fix. The more satellites it uses, the better the accuracy.

How does it know which satellites are where and when to expect to listen for them? Each satellite transmits a database that contains the orbital data for all of the satellites. It takes some time to transmit this database, so receivers store the data in nonvolatile memory (along with the last known location) to preserve this information between power cycles.

When a GPS receiver is first turned on, it does a cold start, which involves cycling through all of the possible satellite codes until it receives a satellite with sufficient signal-to-noise ratio to download the orbital information and the current time. Because the receiver doesn't know if the satellite is approaching or receding, it also has to guess at the satellite's Doppler shift.

When it finds a satellite, the receiver downloads the orbital data and current time. It can then can compute the current constellation (another word for position of satellites in the sky) and tune in to the satellites that should be visible. If the receiver is then power cycled, it can use the data from the NVRAM and the current time from a battery-backed real-time clock to make a good guess at the initial constellation.

If the receiver has been off for a while or has moved a great distance, it may need to perform a warm start. In a warm start, the orbital information is accurate enough for the receiver to receive at least one satellite and start downloading more accurate orbital data immediately instead of having to seek for a satellite first.

GPS-receiver manufacturers make specification claims for the different kinds of starts. Cold starts can take up to 15 min. in some receivers. Warm starts typically take less then 2 min., depending on how good a signal the receiver has. Remember that these specifications are under ideal conditions, with a good antenna, and a clear sky. The boot times vary so you need to make sure they are acceptable for your application.

For embedded-system work, there are several GPS-receiver solutions available. Many hand-held and portable GPS receivers have serial interfaces that can be also used in embedded systems. They also have front panels and displays for user interfaces and many features you wouldn't need for a computer interfacing project.

Photo 2—A low-cost GPS receiver without a user interface has to be controlled by a computer. This receiver plugs into a laptop (or other computer) but contains its own battery pack.

Portable and hand-held receivers are available practically everywhere (e.g., sporting goods stores, department stores). Photo 1 shows a hand-held GPS receiver with an optional communications and power adapter that enables me to connect it to my computer via an RS-232 port and receive NMEA messages from it.

Another kind of portable GPS receiver, such as the one shown in Photo 2, attaches to your serial port or PCMCIA slot of your notebook and has no user interface. These receivers can only be used under computer control.

A common use for these portable/hand-held GPS receivers is to attach them to a laptop and use software like Delorme Street Atlas for car navigation. This is pretty fun and a good way to get comfortable with the technology. You can even use them on airplanes, provided you have a window seat for the antenna and it's OK with the flight crew. See the Navigation 101 sidebar for more details on navigating.

Besides hand-held GPS receivers, there are GPS-receiver modules. Motorola, Rockwell, and several other companies make these small PCB boards that contain all of the analog/RF section and a small microprocessor to perform the computation necessary to find satellites and get fixes. One of these modules, made by SiGEM, comes in a 32-pin SIMM module format.

GPS modules have an antenna connection, power, and one or more serial ports. The serial ports are typically TTL-level asynchronous serial protocol and can be connected directly to a USART in your project. If you want to connect these to an RS-232 port, you need a TTL-level–to–RS-232 converter line driver.

Both GPS modules and portable/hand-held receivers usually speak NMEA-0183 protocol, which I'll get to in a bit. Some modules also speak proprietary protocols that offer more functionality then NMEA-0183 but are specific to a module manufacturer (check the specs). I listed some places that carry GPS modules in the Sources section.

Several companies make ISA- and PC/104-bus GPS boards, which are internal GPS receivers. These typically use one of the GPS-receiver modules in a carrier board. The carrier board also contains a serial USART so the receiver looks just like a serial-port–based external GPS receiver to the computer.

If you want your GPS receiver module to actually receive signals, you need an antenna. Portable and hand-held receivers usually have an integral antenna.

Antennae come in two basic types—passive and active. An active antenna has a small preamplifier built into it, which is powered via the coax cable that connects it to the GPS receiver module. Active antennae are preferable because they provide a much better SNR than passive antennae.

Active antennae do cost more, and the GPS receiver needs to be able to support sending DC power to the antenna. An active antenna without DC power on the coaxial cable won't work at all. The impedance for coax used in GPS is 50 Ω and needs to be low loss and high quality.

NMEA-0183 PROTOCOL BASICS

NMEA protocols NMEA-0180/182 differ in the types of messages that are sent and are much more limited than NMEA-0183. NMEA-0183 is the most general and supports many different kinds of navigation equipment including GPS, which is why I'm going to talk about this protocol.

Its basic structure consists of sentences that are generated by a "talker." There is one talker and one or more listeners on a "bus." The standard doesn't dictate what the physical and electrical specs of the bus are, but typically, it's RS-232 or RS-422. It could also be optical or whatever fits your application.

A sentence is a CR/LF-terminated line that is encoded in standard ASCII. I like protocols that are human readable. It makes it easy to use a terminal or terminal emulator to monitor the protocol during debugging.

Each sentence starts with a $ and ends with a * and an optional 8-bit checksum expressed as two hexadecimal characters. The characters themselves are transmitted without parity. The checksum is computed by performing an 8-bit XOR over the text of the sentence, not including the start ($) and end (*) markers.

The sentence inside the markers consists of a list of comma-separated fields. The number of fields is predefined for a particular message type. If there isn't any data for a particular field, a two-comma "null" field can be sent. In other words, you leave out the data, but keep the field separator.

The first field of the sentence identifies what kind of sentence you have and encodes what kind of talker sent the message. NMEA assigns several talker IDs, which are the first two characters of this field.

For GPS, the talker ID is GP. Other examples of talker IDs are:

- LC—Loran C Receiver
- OM—Omega Receiver
- HC—Magnetic Compass
- P—Proprietary (typically PG if it's GPS)

Navigation 101

Common wisdom and geometry tell us that the shortest distance between two points is a straight line. Well, that's if you live in a flat world. On a spherical surface (the world we live on), the shortest distance is an arc.

Imagine going from Indianapolis to San Diego. Now, take a circle that has a center at the center of the earth and passes through both cities at the same time. The arc of this circle is called the great-circle route and is the shortest distance.

To find our way around, we adopted a polar coordinate system that uses angles to denote positions. Our coordinate system uses one angle (latitude) to describe how high we are above or below the equator. Latitudes go from 0° (the equator) to 90°S (the South Pole) and 90°N (the North Pole).

The longitude is the angle between a position and the longitude that goes through Greenwich, England. It's usually expressed as east and west, where 180° is the same longitude.

When dealing with navigation, it's easiest to deal with nautical miles (nM) and knots (nM/h). One nM is defined as the great-circle distance of one minute (′), and there are 60′ to a degree. So, the circumference of the equator is:

$$\frac{360^\circ \times 60\ \text{min} \times 1\ \text{nM}}{(\text{deg})(\text{min})} = 21{,}600\ \text{nM}$$

Incidentally, the circumference in kilometers is 40,000 km. The conversion from kilometers to nautical miles is:

$$\frac{40{,}000\ \text{km}}{21{,}600\ \text{nM}} = 1.8518\ \text{km/nM}$$

To compute a distance between any two points, simply find the angle of the great-circle route. This step is easy if the points are on the same latitude or longitude: just subtract the nonconstant angles. I've shown you how to do that for finding the distance to circumnavigate the world at the equator (from 0°N latitude, 180°E longitude to 0°N latitude, 180°W longitude).

To compute the distance for nontrivial routes (i.e., the distance angle between any two points on a sphere), you do some coordinate transformations of your polar coordinate system and arrive at the following formula:

dist = acos(sin[lat1] × sin[lat2] + cos[lat1] × cos[lat2] ×
    cos[lon1 – lon2])

Directions are a bit more difficult on a great circle route. They change at any given point on the route so you have to recompute the angle as you go along. This is where modern navigation equipment really shines.

It used to be that a pilot would precompute headings for great circle routes ahead of time and then have to keep track of how long to fly at a particular heading. Thus, the route was just a collection of segments. Modern navigation equipment continually adjusts the heading based on the location and that information is fed directly into the autopilot.

To compute the heading at a given position relative to another position, we use:

course = atan2(sin[lon1 – on2] × cos[lat2] cos[lat1] × sin[lat2]
    – sin[lat1] × cos[lat2] × cos[lon1 – lon2])

The course headings are considered true headings because they are referenced to the true North Pole (the pole of rotation) as opposed to magnetic headings, which are relative to the magnetic pole. Conversion between the true and magnetic heading depends on your location.

Well, these are the basics and will work limited by the precision of the operations you use to perform these computations. For example, if you use a 16-bit number to represent the angle, then the most accuracy you can expect is:

$$\frac{21{,}600\ \text{nM}}{2^{16}} = 0.33\ \text{nM}$$

$$0.33\ \text{nM} \times \frac{1852\ \text{m}}{\text{nM}} = 611\ \text{m}$$

which is good enough to get you to a city, but not good enough to find a house. Any practical computation would require 32-bit operations.

For high-accuracy work, you also need to account for the ellipsoidal nature of the earth and convert spherical coordinates (geocentric) to ellipsoidal coordinates (geodetic). For example, I computed the conversion factor from nautical miles to meters to be 1.8518. The earth is actually a bit fatter at the equator, so the correct constant is 1.8520 (exactly). Do-While Jones gives a good introduction about the transformation and models used for this in Part 2 of his series on GPS (*Circuit Cellar* 78).

Listing 1—This Tcl code for computing the NMEA checksum is simply the XOR of all the ASCII characters between the start ($) and end (*) tokens (exclusive).

```
proc nmea_checksum {sent} {
  set len [string length $sent]
  set sum 0
  set val 0
  for {set i 0} {$i < $len} {incr i} {
    set char [string index $sent $i]
    if {$char == "\$"} continue
    if {$char == "\*"} break
    binary scan "$char" c val
    set sum [ expr $sum ^ ($val % 128)]
  }
  return [format "%02X" [expr $sum % 256]]
}
```

Following the talker ID, there is a sentence-type ID, which is a three-character ID that tells what kind of information is encoded in the remaining fields. The types are defined by the NMEA standard.

At this point, it's probably best to show you an example. Here's a typical message sent by one of my GPS receivers:

$GPRMC,132515,A,3908.8652,N,08625.1367,W,0.387,4.6,290499,2.9,W*7V

You can see the start and end markers as well as a checksum. Listing 1 shows the routine I used to compute the checksum.

The first field, "GPRMC", means that the sentence originated from a GPS receiver (GP) and is a recommended minimum specific-to-GPS/transmit data (RMC) sentence. The generic RMC format is:

RMC,<time of fix>,<status>,<latitude>,<latitude sense>,<longitude>, <longitude sense>,<speed over gound>, <course>,<date of fix>,<magnetic deviation>

So, the sample message was a fix on April 29, 1999, at 13:25:15 UTC, and the location is 39°08.8652′ north and 86°25.1367′ west. The current course is 4.6° true, and we are traveling at a speed of 0.387 knots. The magnetic deviation is 2.9° west. The "A" for status means the data is good. The "V" means that the receiver is not locked in.

Note that although the message has a speed and heading indication, the receiver was actually stationary. I'll get to that in a bit.

Other common messages sent by GPS receivers are the GGA (global positioning fix data), GSA (GPS DOP and number of satellites), and GSV (satellites in view).

The formats for these are:

- GPGGA,<fix time>,<latitude>,<latitude sense>,<longitude>,<longitude sense>,<GPS quality[012]>, <num satellites>, <HDOP>,<sea level>, <units[M]>, <geoidal separation>, <units[M]>,<null>,<null>

- GPGSA,A,<fix mode[0123]>,<sat. PRN>,<PDOP>,<HDOP>,<VDOP>

- GPGSV,<num of msgs>,<msg num>,<sats in view>,<sat PRN>,<elev.>,<asim.>,<signal strength>,<sat PRN>,<elev.>,<asim.>,<signal strength>, <sat PRN>,<elev.>,<asim.>,<signal strength>

With respect to the GPS quality and fix mode information in the GGA and GSA formats, the messages are 0 for invalid, 1 for GPS, 2 for DGPS and 0 for none, 2 for 2D, and 3 for 3D, respectively. In the GSA format, PRN stands for pseudo random number for satellite, PDOP stands for 3D dilution of precision, HDOP is horizontal dilution of precision, and VDOP means vertical dilution of precision.

An NMEA talker will send a group of messages at a time. Different receivers send different kinds of messages and the rate of

	CEP	DRMS	2 DRMS
hpos	40 m	50 m	100 m
vpos	47 m	70 m	70 m
spherical	76 m	86 m	172 m
velocity	–	0.1 mps	0.2 mps
time gps	95 ns	140 ns	280 ns
time utc	115 ns	170 ns	340 ns

Table 1—Take a look at the positioning errors for the civilian standard position service (SPS).

the messages varies, typically from 0.5 to 8 s. In some GPS receivers, rate and message types can be adjusted either with a proprietary message or through the user interface, if it has one.

ERROR AND PRECISION

Although GPS is extremely precise when it comes to navigation equipment, it has some limitations. But first, let me explain the quality measurements that describe the accuracy of location fixes statistically.

The first term, the circle error of probability (CEP), is defined as the size of the circle that encompasses 50% of all the location fixes. So, if you collect 100 position fixes and select the 50 closest ones, the circle that includes these is how certain you can be of your location. A CEP of 100 m means that 50% of all position fixes are within 100 m.

Similar to the CEP, the distance root mean square (DRMS) describes 63% of all the fixes, 2 DRMS describes 97% of the fixes, and 3 DRMS describes 100% of the fixes. Figure 1 illustrates this pattern.

For military use, the figures in Table 1 are much better. For security reasons, the military has turned on selective availability (SA) to purposely introduce uncertainty into the position signal. There's much discussion in GPS mailing lists and usenet newsgroups about whether the SA error gets better or worse during times of conflict, but all we need to know is that SA exists and has some peculiarities to be aware of.

In the RMC message I showed you, you saw that there was a speed and heading in the stationary fix. This happens because of SA. The SA dither changes over time, so the position fix seems like it's slowly drifting. The speed that the GPS receiver measures is actually the rate at which the time is changing.

If you average a stationary position over a long time (two or more days), you can achieve high accuracy because the random-number generator used to dither the signal for SA has a zero mean over several days. Figure 2 shows a plot of SA for 1 h.

Besides the long-term mean of the SA dither, there's another effect we can use. The SA dither is the same for locations within a locality (i.e., the error of the position is the same for positions that are apart). If you use a base station with a known accurate location, you can measure the current error of another measurement of which you do not know the location.

For example, if you know the current location (*A*[lat,lon]), and the location that a GPS receiver reports to you (*B*[lat,lon]), you can compute the current error with:

$$\text{err(lat,lon)} = \text{B(lat,lon)} - \text{A(lat,lon)}$$

If you take a GPS reading somewhere in the field at *C*(lat,lon), you can compute an accurate position, by adding in the offset:

$$\text{real(lat,lon)} = \text{C(lat,lon)} - \text{err(lat,lon)}$$

Systems use this technique and broadcast the current error information via radio signals. The US Coast Guard uses such a system near coastal waterways via long-wave radio stations, and their system is free to use. In many cities, differential GPS may be available via FM broadcast station subcarrier. These services usually require a subscription and are not standardized.

In future articles, I'll show you how to use a stationary GPS receiver and a roving GPS receiver and do differential GPS using postprocessing. Basically, I'll show you how to map out donut shops with fairly high accuracy, in case you ever need to put some LEDs on a map. RPC.EPC

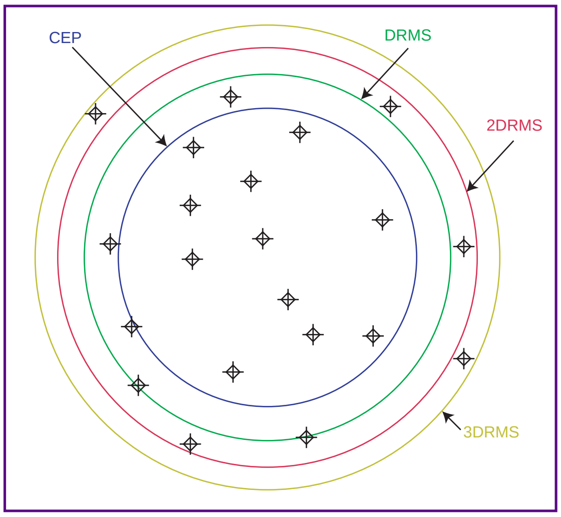


Figure 1—Here's a graphical representation of CEP and the various degrees of RMS distances. The CEP (50%) has half of the fixes, DRMS (63%), 2 DRMS (90%), and 3 DRMS (100%).

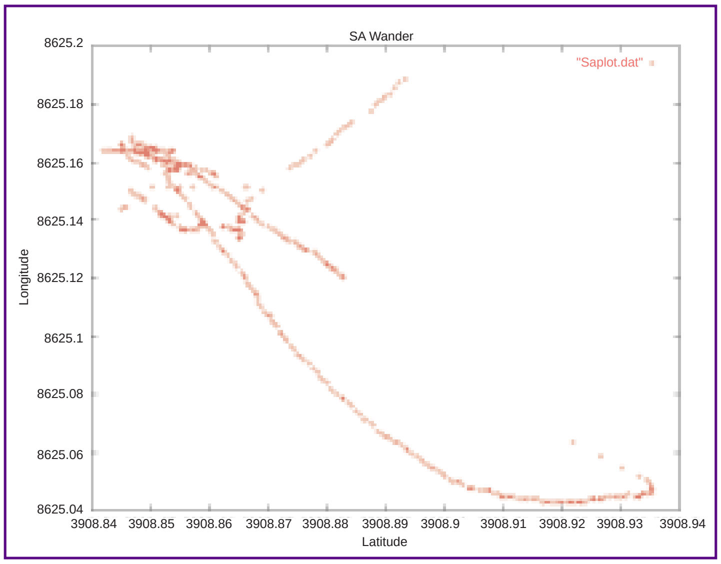


Figure 2—As this plot shows, SA causes the fixes on a stationary GPS receiver to wander over time. This track was taken for ~1 h. Each longitude tick mark of (0.02′) is about 28.8 m, while each latitude tick mark (0.01′) is 18.5 m.

Ingo Cyliax has written for Circuit Cellar *on topics such as embedded systems, FPGA design, and robotics. He is a research engineer at Derivation Systems Inc., a San Diego–based formal synthesis company, where he works on formal-method design tools for high-assurance systems and develops embedded-system products. You may reach him at cyliax@derivation.com.*

REFERENCES

Differential GPS info, www.eskimo.com/~archer/gps.html

B.Clarke, *Aviator's Guide to GPS*, McGraw-Hill, New York, NY, 1998.

Navstar GPS service info, www.navcen.uscg.mil/policy/dgps/rctm104/default.htm

NMEA-0183 and GPS info, www.vancouver-webpages.com/pub/peter/index.html

E. Williams, *Aviation Formulary V1.24*, www.best.com ~williams/avform.htm

SOURCES

GPS units/service
Tucson Amateur Packet Radio
(940) 383-0000
Fax: (940) 566-2544
www.tapr.org/tapr/html/gpsf.html

NMEA
(252) 638-2626
Fax: (252) 638-4885
www.nmea.org

Applied PCs

Fred Eady

Data Serving via the Internet

Part 1: Setting Up

You know it's true: Fred loves embedded Internet appliances. Once you see the basics of the EmWeb server, you'll understand why Fred is using it with a Net186 board to get data over the wires—perhaps even into your fridge!

It's been said that the next real war will be fought on the Internet. Well, last week I was reading a tidbit about an Internet refrigerator. No kidding, this thing keeps its own food inventory and is even capable of going shopping online when you run out of hotdogs.

To add credence to the Internet war theory, a foreign hacker recently decided to take some U.S. government web pages hostage. Well, that only proves that the next war will indeed be Internet based and the first thing the enemy will do is wreak havoc with the opponent's chat rooms and cut off all opposing refrigerators.

All my life, I thought that the first thing invaders would go for would be command and control. You know, power, telephones.... I'll keep my hands out of politics and diplomacy and stick to embedded stuff. Maybe Martians will invade us and I won't have to worry about my refrigerator *or* the Internet.

You probably know I'm on an embedded-Internet-web-client/server-TCP/IP-Ethernet mission, and this sortie will fly us over familiar territory. The equipment will be standard issue with the addition of some special tactical electronics and software. We'll be flying with AMD's Net-186 demo board loaded with Agranat's EmWeb and EmStack.

The nature of the mission is to move electrons through the Internet and get you to devise your own Internet widget. It's nice to know your equipment before takeoff, so let's take a walkaround of EmWeb.

EmWeb, SOUNDS LIKE...

Embedded. Agranat's EmWeb is built around two main components—the EmWeb compiler and the EmWeb server. All you have to do is add C code and the right amount of html source in the right places.

The EmWeb server is a C code component, and the source html can be generated using standard web-authoring tools. To use the functionality of the EmWeb server, the application programmer adds some custom embedded software. A fully compliant http/1.1 protocol engine that sports plenty of software knobs for user

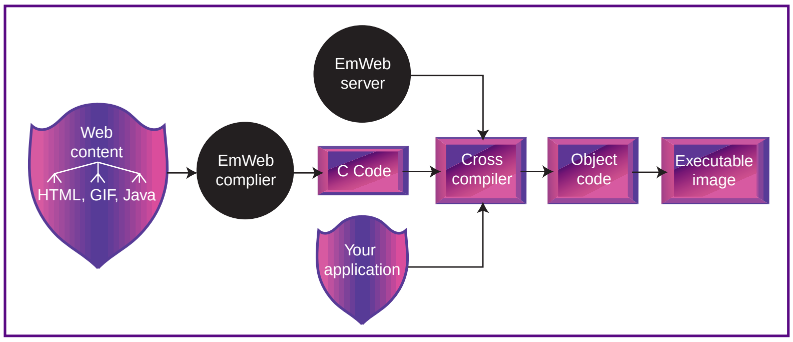


Figure 1—As you can C, EmWeb crunches everything into a tidy and compact embedded image.

customization powers the EmWeb server. This server is built using only the components you need to do the job at hand. In addition to providing a foundation to help build this server, the EmWeb compiler is capable of compressing the web content. This arrangement makes the server right at home in most any embedded environment.

No OS is needed to put the EmWeb server online. All you need is good ol' TCP/IP and some embedded network hardware.

Listing 1—Notice the apostrophes that delineate the actual C code. This example returns "NetFax is Idle", "NetFax is Sending", or "NetFax is Receiving".

```
NetFax is <b>
<EMWEB_STRING C='
   static const char* NetFaxStatusDisplay[3] =
     { "Idle", "Sending", "Receiving" };
   static int RotatingFaxStatus;
   RotatingFaxStatus = ( RotatingFaxStatus + 1 ) % 3;
   return (char*)NetFaxStatusDisplay[ RotatingFaxStatus ];
'>
```

html is html. Standard html source is processed within EmWeb by adding special EmWeb tags that can include C or C++ code to define an interface between dynamically generated data on web pages, the system software, and web content.

Once the programmer defines and codes the web content, the EmWeb compiler is invoked to process the web content files, strip out and interpret the special EmWeb tags, and generate C code. The compiled C code becomes the web content that is compiled and linked with EmWeb server and custom user code. The result: a dynamic and interactive GUI that can be accessed with any web browser.

The compiler also generates a binary image that may be directly loaded into an embedded platform's memory area. The EmWeb concept is depicted in Figure 1.

PLAYING TAG

The whole idea here is to dynamically serve pages and documents that contain particular real-time data. By particular, I mean things like current time, temperature, or machine status.

In the EmWeb environment, this is done via special tags that are integrated into the html source content. These tags contain C source code that enables the dynamics of the served web pages.

Chances are, if you read *Circuit Cellar*, logical thought processes come easy for you. After contemplating this dynamic server concept, you could probably write code to effect it. But if that were so, I could (and would) stop writing this piece right now. There would be no need for the EmWeb server or the words in this article.

My point is, EmWeb server relieves you, the embedded web programmer, of the figure-out-and-program effort by providing extension tags that don't require you to write extra or supporting routines to make them work.

Besides, the embedded web-server market is fast moving and, frankly, unless you develop embedded web servers (and that's all you do), you don't have time to roll your

Listing 2—Before compilation, both 10s would show. After compilation the EMWEB_PROTO-surrounded 10 would disappear.

```
<blockquote>
The count is: <b>10</b>
<br>
The count is: <b><EMWEB_PROTO>10</EMWEB_PROTO></b>
</blockquote>
<p>
```

own server code and stay competitive. Let's examine some of the server tags, beginning with `EMWEB_STRING`.

`EMWEB_STRING` enables you to insert C or C++ code into your html source to return a value to be displayed by way of the browser GUI. The only code you write is a function to access the resources needed to compute the return value. Your function must end with a return statement that points to the value. If no value is to be inserted, a `NULL` return is required.

The function is usually a C code fragment placed within the confines of the EmWeb server tag. Listing 1 shows how `EMWEB_STRING` is typically used.

Remember that EmWeb html web content is created the normal way using your favorite html editor. This is good and bad. The good side is that your web content can be easily created. The bad side is that if you will be displaying dynamic values, you can't see them when you static test the html-based page.

EmWeb server provides a prototype tag, `EMWEB_PROTO`, which enables you to insert the expected value on the page where it would normally appear. This tag takes advantage of the fact that browsers ignore tags they don't understand. It gets better. The EmWeb compiler strips out the `EMWEB_PROTO` tag and content and doesn't add this comment or test content to your final product.

If you work by committee, you can send EmWeb-server-tagged html to everyone and they can see it just as it would be if an application were behind it. Most often, it's easier to write code than please an end user. With the prototype tags, you can propagate your ideas to the user community for critique before you pour the final source-code concrete. An example of `EMWEB_PROTO` usage is shown in Listing 2.

What if the dynamic value you need to display is not a string? Our look at `EMWEB_STRING` revealed an internal C code fragment that ultimately returned a pointer to a `NULL`-terminated character string. Typical C stuff.

What if the requested data is an integer or, better yet, an IP address? No problem. The EmWeb server lets you specify a type for the data as an attribute of the `EMWEB_STRING` tag. Again, you write no special code for this conversion as the EmWeb server performs the type-to-html translation automagically.

There are a variety of value types including signed and unsigned integers, IP addresses, MAC addresses, and hex integers. While I'm on the TCP/IP bandwagon here, Listing 3 shows the code that performs the unsigned-integer–to–dotted-IP–to–html trick.

EMWEB_ITERATE and EMWEB_INCLUDE are two more tags that make the EmWeb server embedded-friendly. EMWEB_ITERATE is simply another attribute of EMWEB_STRING. EMWEB_ITERATE allows dynamic code snippets to be repeated as long as the return value is not NULL. The beauty of this is that repetitive operations such as table building can be invoked with a single reference to one iteration-equipped EMWEB_STRING statement.

Sometimes it's necessary to assemble an html page using pieces from other documents. Assembling this type of page is done by the server without help or influence from the browser. This process is called Server Side Include.

To the browser, the page it receives is the "single" document it received by issuing a single request. The advantage of using Server Side Include is that a document can be created once and used many times on multiple pages without having to be rewritten from scratch each time it is used.

The downside of Server Side Include is that some servers have to examine the documents in the document pool for special extensions so that the selected content will be placed in the correct areas of the final document. If the server has the resources to provide this service, it's a small price to pay for the flexibility.

In the embedded world, those types of resources (in the quantity needed) are normally not around when you need them. EMWEB_INCLUDE is the EmWeb server tag that implements Server Side Inclusion the frugal embedded way.

The EMWEB_INCLUDE tag comes in three flavors. You can include a document with <EMWEB_INCLUDE COMPONENT= "document">.

In this context, the COMPONENT attribute implies that the "document" is actually a URL that represents the name of another document located on the same embedded platform. Or, if a dynamic document inclusion is required, you can use <EMWEB_INCLUDE C="C Code">.

The C= attribute determines the URL to be included at runtime. The "C Code" is actual C code that has been embedded into the html content and is executed during runtime when the document is served.

A string containing the URL to include is returned. If a NULL pointer is returned, no document is included. This concept could be used to interrogate a system and serve pages that relate to just that particular system from the document pool.

The final flavor uses iteration to invoke actual C code multiple times until a NULL pointer is returned. Each time a non-NULL pointer to a URL string is returned, the contents of that document are served. This line of code behaves somewhat like the EMWEB_ITERATE variant of the EMWEB_STRING tag:

```
<EMWEB_INCLUDE EMWEB_ITERATE C=
  "C Code">
```

So far we've looked at some tags and code that could be assembled to form some pretty good web server content. You've heard me say it many times: to be successfully embedded, the hardware and software must be efficient.

As you progress through the design and buildup of web page content, you may come to the conclusion that some of the content must be duplicated in different

Listing 3—Just in case you're wondering, 10.1.2.3.

```
Default Gateway Address:
<EMWEB_STRING EMWEB_TYPE=DOTTED_IP
  C='static uint32 example = 0x0a010203;
  return &example;
'>
```

areas of the design. Rather than embed the source multiple times in the web content, there should be a mechanism that allows the code to be reused rather then rewritten.

The EmWeb server solves this problem by employing the EMWEB_MACRO tag. The reuse property of the EMWEB_MACRO tag is applied in both the EMWEB_STRING and EMWEB_INCLUDE tags. An ID= attribute in the place of the C= attribute enables the programmer to name dynamic EMWEB_STRING or EMWEB_INCLUDE tags.

The named tag is considered global and can then be accessed from anywhere in the web content. Normally, a macro inserts itself fully into the code at each instance in which it is called. Here, instead, the EmWeb server macro is referenced. Thus, the macro takes up only the space in which it is originally written.

For example, if you replace the C= in Listing 1 with ID="faxStatus", you could reference the code in Listing 1 from anywhere in the content with <EMWEB_MACRO ID="faxStatus">. An added benefit of the EmWeb macro implementation is that the macros can be used without having to recompile the web source content.

One last tag before I move on and add more logs to this fire. The tags I've discussed—EMWEB_STRING and EMWEB_INCLUDE—enable us to insert C and C++ code into homegrown html files to generate dynamic content.

As you well know, html and C weren't initially designed to be mixed. C code most always includes an #include file. There will be times when code in the EMWEB_STRING constructs will need to reference information from that infamous #include file. Thus, the last tag I'll discuss is the EMWEB_HEAD tag.

Just like its cousins, the purpose of the EMWEB_HEAD tag is to enable the web programmer to insert C code into the html source code. Although you can put anything you want in the EMWEB_HEAD area, the code within the EMWEB_HEAD tag is usually C declarations or #include files that are needed by supporting EMWEB_STRING and EMWEB_INCLUDE routines. An EMWEB_HEAD tag looks like <EMWEB_HEAD C="C Code">.

You don't know what an EmWeb archive is yet, but when the EmWeb compiler is run, the code specified in all the EMWEB_HEAD tags in the archive is gathered up, combined, and placed into the start of the generated C code output. C code specified in an EMWEB_HEAD tag appears before any code specified in the EMWEB_STRING and EMWEB_INCLUDE tags.

It is recommended that all #include, #define, and globals be placed in one EMWEB_HEAD structure in the index.html file. That way all EMWEB_HEAD code throughout an entire archive is combined, which avoids multiple includes of the `includes`, if you get my drift. Listing 4 shows how `EMWEB_HEAD` and `EMWEB_STRING` work together.

WHAT'S AN ARCHIVE?

While we're discussing archives, let's define a full EmWeb application. There are four major components. The first is one or more archives (html documents and C source code that you compile to produce web pages).

In effect, an archive is a repository for html and C code that becomes your dynamic web content. Documents within the archive can be html files, Java programs, graphical images, or any resource that can be identified by a URL. The archive is really acting as a run-time database for the EmWeb server.

Archives can be loaded dynamically, produced as binary images for separate loading, created as distributed archives, and partitioned to accommodate the creation of derived archives. A derived archive enables the static portions of a "host" archive to be modified and subsequently reloaded without having to rebuild the entire system.

The second major component is the customized EmWeb server. Basically, the default EmWeb Server serves static documents only. Well, you've read this far, so you know that EmWeb server can also deal dynamic pages, too.

Because EmWeb is targeted for embedded apps, it's designed to be as small or as large as the application requires it to be. Being an embedded chameleon requires the ability to switch certain attributes on and off, depending on the environment. The EmWeb server does this by enabling the programmer to pick and choose the final server's capabilities.

The EmWeb server capabilities fall into these functional areas:

- archive and document maintenance
- authorization and security
- distributed processing
- file access
- network
- request context access
- system

Archive and document maintenance involves permissions for:

Listing 4—Some folks like chocolate. I like `switch` statements. `myFunction` is a utility function that can be used by any embedded C/C++ code in the html archive.

```
<EMWEB_HEAD C='
static const char * myFunction(int value){
  switch (value) {
      case 0:
        return "Zero";
      case 1:
        return "One";
      case 2:
        return "Two";
      default:
        return "some other number";
    }
}'>
<p>
<ul>
<li>myFunction called with <EMWEB_STRING C="return myFunction(0);">
<li>myFunction called with <EMWEB_STRING C="return myFunction(1);">
<li>myFunction called with <EMWEB_STRING C="return myFunction(2);">
</ul>
```

- allowing compression of the archive
- URL redirection ability
- dynamic loading of archives
- demand loading of archives
- document cloning

Archive compression can be a big advantage in embedded systems with small memory footprints like the Net186. URL redirection allows redirection of URLs within the archive and provides a way to handle http requests for external resources.

I mentioned how being an embedded chameleon can be a good thing. Well, the EmWeb server can load archives according to circumstances. That's embedded chameleon! Demand loading involves only loading an archive if it contains a requested resource.

The final, most interesting feature is document cloning. A document may have separate clones, each with its own URL, but store only a single copy of the original document.

Let's take a quick look at the remaining items in the list. EmWeb security is basic http authentication and is used to restrict access to documents and archive resources.

Distributed processing enables EmWeb to be implemented as a network of servers presenting the face of a single URL hierarchy to browsers. Access to the local embedded file system is also supported so http requests that require file access outside the EmWeb archive can be accommodated.

As for the network, TCP/IP is the major player. Request context access is a set of functions provided by EmWeb that permit user code to get information about the current http request.

"System" refers to basic functions like time and date, provided by the EmWeb server, and namespaces are used to define an interface between EmWeb and an external database. The final two components are application code and network support.

In a nutshell, EmWeb server is a TCP/IP network-based, self-contained http-compliant, static and dynamic document web server that's capable of being compressed with its C-laden web content and placed on a limited-resource embedded platform. Whew!

DON'T CHANGE THAT URL

Well, I still have to declare the hardware and show how I got the bytes from the EmWeb compiler to the Net186. I also need to tell you about the compiler and walk some info through the system.

In the meantime, keep an eye on your web sites and stock up on hotdogs because it doesn't have to be complicated to be embedded. APC.EPC

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

SOURCES

EmWeb
Agranat Systems, Inc.
(978) 461-0888
Fax: (978) 462-1080
www.agranat.com

Net186 eval kit
Advanced Micro Devices, Inc.
(818) 878-9988
Fax: (818) 878-9081
www.amd.com

MICRO SERIES

Glen Reuschling

USB Primer

Building a USB Host Controller

Part 4 of 4

If you're looking for detail, you have no excuses left: USB is a mystery no longer. By describing the step-by-step construction of his low-speed USB host controller, Glen reveals the inner workings of USB hardware.

Last month, I introduced my host controller. Now, I want to show you how it all came together.

First, a note on the choice of parts that went into it. I chose the TI TUSB-2040 four-port USB HUB for the core of the µSIE because it's a stand-alone part that requires no programming and is available in a DIP package.

HUB support chips were chosen based on the application examples in the TUSB2040 datasheet. CPLDs, which support in-circuit programmability, are a must for the initial prototyping of your host controller.

As we go through the process, you may want to have these files (available on the Circuit Cellar web site) handy:

- HUB01C.PLD—programming for low- and full-speed clock generation, the DPLL, and bus time-out detect logic
- HUB04C.PLD—dual-port RAM address control logic
- HUB02C.PLD—µSIE functions

To make signal tracing easier, the net names used in Figures 1 and 2 match the variable names used in the .PLD files.

A simple way to implement subroutines within state-machine logic is to use a multiphase clock. One level of state machine controls the clock signals to the next level down.

The top-level state machine controls the idle, start of frame, and send/receive states. At the next level down are the state machines that generate send and receive transactions. The third and fourth levels are responsible for bit and byte manipulation.

Although the complete µSIE (including the dual-port RAM) can be put into a single, large CPLD, it's less expensive to implement a design using several smaller devices. But always pick a bigger part than you think you need. To help debugging, leave a few extra pins on your CPLD for test points.

GETTING STARTED

Start with a development card or prototyping system for your favorite 8-/16-bit microcontroller. A debugger for downloading and running code is essential. And be prepared to generate a number of different test routines.

Set aside some memory space for a dual-port RAM; 1 or 2 KB is enough. This RAM will be the main driving engine for your host controller. A good storage oscilloscope or logic analyzer is a must for debugging the hardware.

CLOCKING THE µSIE

Several system clock signals must be generated for the µSIE [3]: a four-phase full-speed 12-MHz clock, a four-phase 1.5-MHz low-speed clock, and the 1-ms frame period clock. The µSIE uses a 48-MHz crystal oscillator as the primary source for all timing, so generating the 12- and 1.5-MHz four-phase clock signals becomes straightforward.

The USB Specifications call for the 1-ms frame clock to be adjustable [1, p. 124], but this is from the perspective of a USB slave device. Many USB slave devices use ceramic resonators instead of crystals for their clock so some provision was included to compensate for their lower accuracy and stability.

The 1-ms frame clock is implemented as a divide-by-*n* counter, with *n* giving the adjustability called for [4]. Because the host controller is the bus master and sets the 1-ms frame period, it doesn't need to be software adjustable. So *n* is fixed equal to 48,000.

Regarding the 1-ms frame clock as implemented in HUB01C.PLD, the more switching the CPLD does, the more power it dissipates. At 48 MHz, the Lattice part gets quite warm, so the divide-by-*n* counter was broken up into several stages, each being clocked by a slower signal from the stage before.

FEEDING THE µSIE BABY

The first test routine to run on your 8-/16-bit micro is one that, upon a hardware interrupt generated by the host's 1-ms frame clock, writes a few bytes of data out to a starting location in the dual-port RAM. Make the first byte (i.e., ByteCount) the number of bytes in your test packet.

The first HDL state machine to download to the CPLD is one that, when initiated by the 1-ms frame clock, starts by reading the number of bytes to transfer and loads the data sequentially from the dual-port RAM into a shift register [5] and clocks it out as a serial stream. Output clocking is a shift-right least-significant-bit-first operation.

Exactly how the data is read from or written to the dual-port RAM depends on what stage of the build-and-test process you're at. Understanding the final form of the dual-port RAM's read/write logic must wait until after the inner workings of the µSIE have been explained.

Using the serial bitstream generated by the first test routine, start implementing in HDL code the bit-stuffing and NRZI-encoding functions [1, pp. 121–122]. Bit stuffing is accomplished by counting the number of 1s (in a row) sent and, when that count hits six, clocking a 0 into the NRZI encoding section without clocking the serial-out shift register [6].

USB is a two-wire medium, so there can be up to four signal states. Data information is encoded as a differential signal; control information is encoded as a DC-level signal [1, p.115].

The USB uses three of the four possible states—J logic state, K logic states, and the single-ended zero (SE0) state used for sending control information [7]. Also, the NRZI state machine needs a disconnect or floating-output state.

The Sync Pattern determines the initial state and initial transition for the NRZI state machine. When the host controller initiates a transaction, it takes the bus from a floating state to the Idle state.

The first Sync Pattern transition is from Idle to K. So, the NRZI state machine must start in the disconnect state, go to the J state for one bus-clock period, then transition to the K state. After this startup, the NRZI encoder runs itself.

In NRZI encoding, data is encoded as transitions, not levels. A 0 gives a transition; no transition indicates a 1. When viewing the USB signal on your logic analyzer, don't look at it as a sequence of 1s and 0s. The transitions between 1s and 0s represent the bits in the USB serial signal.

DIFFERENTIAL LOGIC LEVELS

The differential logic levels for low-speed USB communication are the reverse of those of the full speed, but the SE0 control signal is still the same. Reversing prevents full-speed devices from responding to low-speed signals.

But, this reversing is only from the perspective of the slave controller. The host still sends out all transactions in full-speed differential logic levels.

The last hub device before the receiving low-speed device is responsible for reversing the sense of the differential signal levels [1, p. 224]. So, the µSIE doesn't need to implement or support the full-/low-speed signal-level reversal.

Now that your prototype host controller can generate a serial bitstream of the right form, issues of USB protocol must be dealt with (i.e., what bytes of data and in what order must they be written to the dual-port RAM).

Due to definition overload, the information in chapter 8 of the USB Specification, "Protocol Layer," can be difficult to interpret. Unfortunately, this chapter is essential for USB hardware design.

Figure 3, which shows a Start Of Frame (SOF) packet and a complete USB hub Control Read sequence, may help elucidate the hardware-level details of the USB transaction protocol. It's an expanded version of the Control Read sequence in Figures 8–12 of the Specification [1, p. 154].

The Control Read sequence shown here is the GET DESCRIPTOR device request. Because the USB uses the duration of the SE0 state to encode control signals, time-delay information is critical.

The next USB test signal the host controller needs to generate is the SOF packet [1, p. 149]. So,

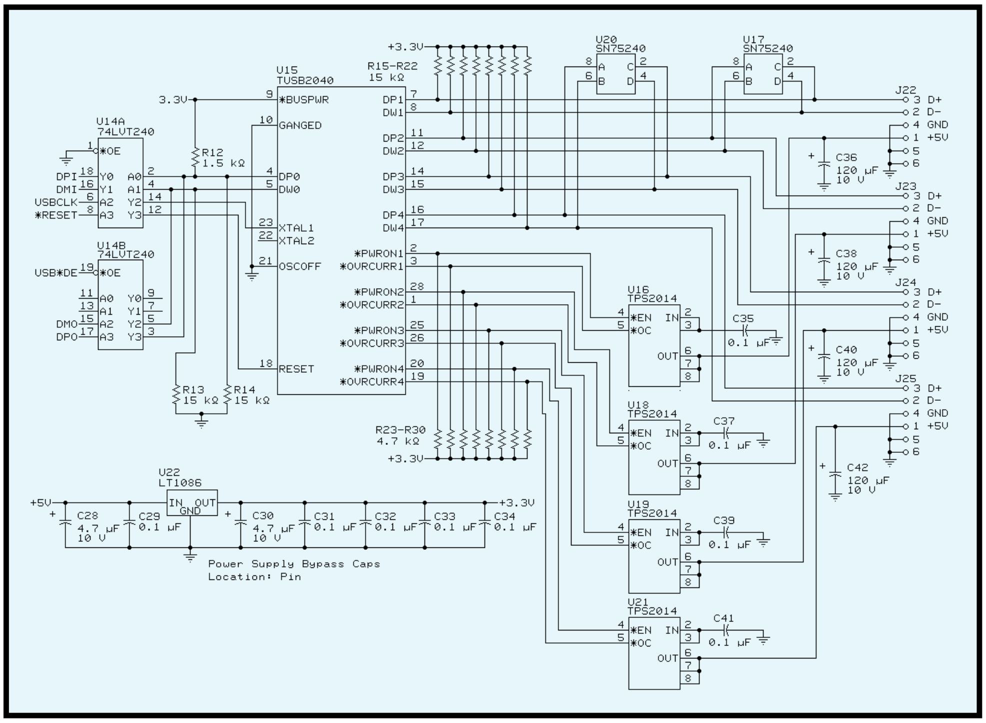


Figure 1—*This schematic shows the wiring diagram for the TUSB2040 four-port USB hub. The TUSB2040 is a +3.3-V device, so a buffer (U14) was included to translate between +3.3- and +5-V logic levels.*

the second test routine to download and run on your 8-/16-bit controller is one that, upon a hardware interrupt generated by the host's 1-ms frame clock, increments an 11-bit frame counter, constructs the next SOF packet, and writes that packet to your starting location in the dual-port RAM.

I let the microcontroller generate all data fields and the µSIE generate all control signals. My controller does Sync Pattern, PID, Frame Count, CRC5, CRC16, and ACK; the µSIE generates the Idle and End of Packet (EOP) states.

Sync Pattern is seven 0s followed by a 1—that is, the byte 80h shifts out least significant bit first [1, p. 123]. When this byte is run through bit stuffing and NRZI encoding, it becomes [Floating, Idle,K,J,K,J,K,J,K,K] in USB form or [HiZ,1,0,1,0,1,0,1,0,0] as logic levels.

The 5-bit CRC is easily generated via a 2-KB look-up table, with the 11-bit frame count as the address offset [8].

In addition to the bytes fed to it via the dual-port RAM, the host needs to append the final EOP and Idle control states. The EOP is a SE0 state, and the Idle is a J state. After these are sent, the USB signal lines are floated in anticipation of a possible return signal.

The EOP and Idle control states are specified as time durations, which are different for low and full speeds [1, p. 125]. It's easier to think of them in terms of bus clock periods, the EOP being two and the Idle state being one bus clock period, respectively.

For USB, the word "transaction" refers to the process of data transfer between the host controller and the slave USB device. It's not a hardware-level concept but a protocol- or software-level concept. At the hardware level, the basic communication unit is the packet.

A transaction consists of a sequence of packets (initiated by the host) being sent back and forth between the host controller and a USB slave device. The host's µSIE sends and receives individual packets. Completing a full transaction is a software issue for the 8-/16-bit microcontroller.

The USB Specification calls for ten different packet types, each one being identified by its PID and representing a different handshake message or data message type [1, p. 146]. A USB transaction starts with the host controller sending a Token packet, or, in the case of low-speed transactions, a Special packet followed by the Token packet.

The Token packet initiates communications, but after the initial Token packet is sent, there may or may not be any further packets sent or received. For example, the SOF transaction consists of only the SOF Token packet.

Here's a classic example of the definition overloading that makes reading the USB specification difficult for the uninitiated. Depending on the context, "SOF" can refer to a transaction type, a packet type, or a Token PID.

Building the SOF packet starts with the microcontroller placing the `ByteCount` byte, then the sync byte, the PID byte, and two bytes for the frame count and CRC5 into the dual-port RAM. For example, with a frame count of 54h, the SOF packet passed to your µSIE is [04h,80h,A5h,54h,90h].

The µSIE input state machine reads in and shifts out these four bytes of data, then goes to the SE0 state for two bus clock periods, then to the Idle or J state for one period, and then disconnects [9].

Congratulations! You've just completed your first USB transaction.

THE USB HUB

The next step is a full-speed USB hub Control Read sequence. The '2040 hub controller chip was enlisted to do all of the bus housekeeping functions, including line driving, slew rate control, hot-swap power management functions, and low-/full-speed device detect. This is a nonstandard use for this part, but it does the job and saves work in re-creating (as separate circuitry) all of the functions it performs.

The '2040 is a USB slave controller and, before its functions can be accessed, it must be initialized just as any other USB device. All USB hubs are full-speed devices, so the clock-generating section contains logic for a 12-MHz full-speed clock signal.

Fortunately, it's only necessary to talk to the '2040 hub to initialize it. Listening isn't required at this stage, so the first USB communication that the host controller will initiate is a full-speed Control Read sequence with its own '2040 hub.

Note that in wiring the '2040 device, because control information is sent as a DC-level signal, it's crucial that you pay attention to the correct pull-up and pull-down resistors in your design.

The Control Read sequence consists of a number of transactions, each consisting of the exchange of two or three packets. For example, the setup transaction consists of a SETUP token packet followed by a DATA0 data packet and ending with an ACK handshake packet.

At this point, a logic analyzer or storage oscilloscope becomes a necessity to monitor the serial signals going to and coming from the '2040. Monitoring ensures you're getting the right signals out to it, in the right order, and with the right timing, and lets you verify that the Control Read sequence is completing correctly.

The first packet in the first transaction of the Control Read sequence is a SETUP packet. The SETUP token is interpreted by the hub the same as an OUT token.

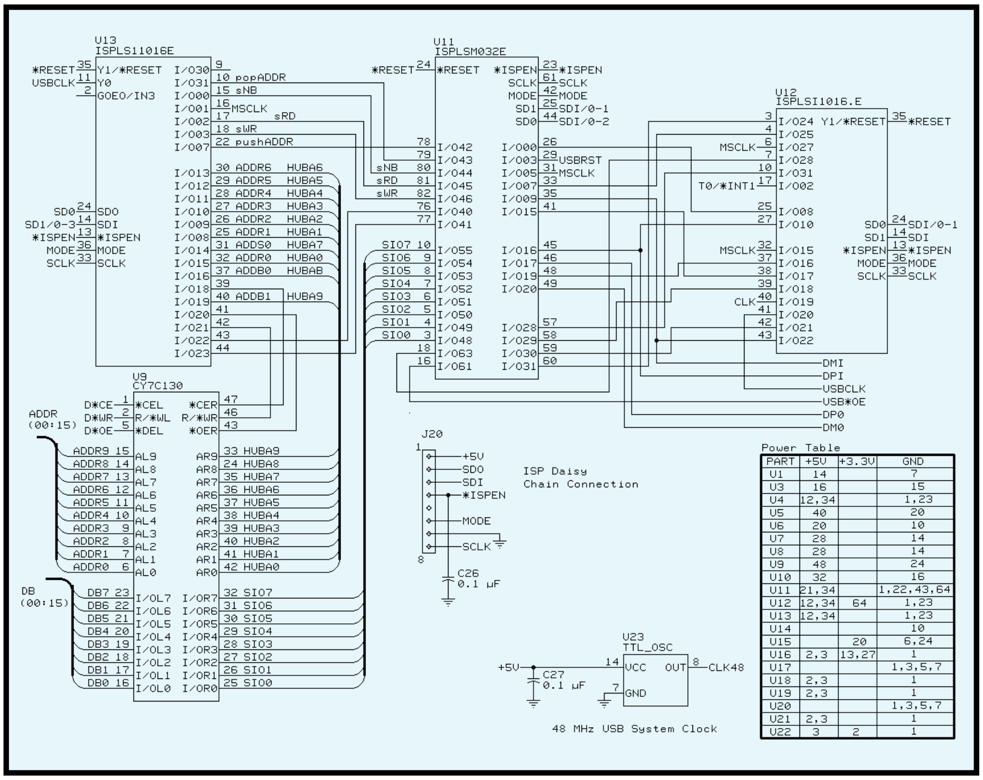
On powerup, all USB devices default to address 00h and endpoint 00h. These will be the initial ADDR and ENDP values for the TUSB2040 part.

To get the CRC5 value for this or any SETUP, IN, or OUT packet, combine the ADDR and ENDP values to form an 11-bit address. Then use this value to offset into the CRC5 look-up table [10].

As with the SOF packet, building the SETUP packet starts with the microcontroller placing the `ByteCount` byte, then the sync and PID bytes, and two bytes for the ADDR, ENDP, and CRC5 into the dual-port RAM. Your result should be [04h,80h,2Dh,00h,10h].

The purpose of this SETUP packet is to send a message to endpoint 0 of the device at address 0 to expect incoming data on the next packet out. But after getting this message packet, the receiving device will only listen for so long before timing out, which brings up the issue of bus turnaround time.

The bus turnaround time is basically the time period (after the end of a packet's transmission) that any USB device still expecting to receive another packet will wait before "hanging up." The USB Specification calls for at least 16 (but no more than 18) bus clock periods for this interval [1, p. 161].



Power Table

PART	+5V	+3.3V	GND
U1	14		7
U3	16		15
U4	12,34		1,23
U5	40		20
U6	20		10
U7	28		14
U8	28		14
U9	48		24
U10	32		16
U11	21,34		1,22,43,64
U12	12,34	64	1,23
U13	12,34		1,23
U14			10
U15		20	6,24
U16	2,3	13,27	1
U17			1,3,5,7
U18	2,3		1
U19	2,3		1
U20			1,3,5,7
U21	2,3		1
U22	3	2	1

Figure 2—*This schematic shows the host-controller support logic. The nets ADDR[00:15] and DB[00:07], left side of the dual-port RAM, connect to your 8-/16-bit microcontroller. Programming for the three CPLD devices is available via ftp.*

The second packet in the Setup transaction is a DATA0 packet. DATA0 and DATA1 are the two packet identifiers used to distinguish data packets. The first data packet in any transaction is always labeled with the DATA0 PID.

The second data packet begins with the DATA1 PID, and the third with the DATA0 PID. They then alternate between DATA1 and DATA0, except for Control sequences, which end with the exchange of an empty DATA1 packet.

BACK TO THE µSIE

So far, the dual-port RAM read/write logic has only had to output one packet at a time. Because of the short bus turnaround time allowed for full-speed USB devices, your DATA0 packet must be ready to go right after the SETUP packet is sent, which means it must already be in the dual-port RAM along with its associated SETUP token packet.

That's just one example of the general case, that, given the short bus turnaround time, the host controller must be able to complete a full transaction all at one time.

On the microcontroller side of the dual-port RAM, transactions are stored in a semi-linked-list fashion (i.e., the ByteCount byte, which starts each data sequence, is used as an offset to the next ByteCount byte). If this count is zero, the SIE knows that no more data is to be sent and goes to an idle state until the beginning of the next 1-ms frame clock.

To keep the 8-/16-bit microcontroller from causing conflicts by trying to access the same segment of the dual-port RAM at the same time as the µSIE, its memory is divided up into four 256-byte pages [11]. Each page is in turn divided into 128-byte output and 128-byte input segments [12].

The four memory pages form a cyclic queue, with the micro (via its I/O pins) keeping track of the current page the µSIE is on. While the µSIE is accessing the current page, the micro is writing to the output segment of the next page and reading from the input segment of the previous page.

Along with these modifications to the dual-port RAM address control logic, changes have to be made to the µSIE programming. Two changes are the additions of an "expect return flag" and a "data ready flag" [13]. Because the value of ByteCount for low-speed transactions never exceeds 16, the lower four bits of this byte are used for ByteCount and the upper four bits are used for control flag information [14].

With this modification, your µSIE will know from the ByteCount byte that the SETUP packet expects no return handshake and that there's a second packet waiting in the dual-port RAM to be sent out before the bus turnaround time expires. This is probably the biggest single leap in your HDL code development, so expect to spend some time working through it.

INITIALIZING THE USB HUB

This DATA0 packet contains the USB device request information necessary to start the hub initialization process [1, chap. 9]. All USB devices must be initialized by the following minimum exchange of information.

On startup, all USB devices default to ADDR 0 so the host issues a GET DESCRIPTOR device request to the default control endpoint, ENDP 0, at ADDR 0. The device address is set to a new value and the device descriptor is requested again at the new address.

The GET DESCRIPTOR request consists of [1, p. 176]:

- byte 1—bmRequestType, 80h, device request
- byte 2—bRequest, 06h, request code
- byte 3—wValue, 00h, descriptor index
- byte 4—wValue, 01h, descriptor type
- byte 5—wIndex, 00h, not used
- byte 6—wIndex, 00h
- byte 7—wLength, 12h, 18 bytes is standard descriptor length
- byte 8—wLength, 00h

As was the case for the 5-bit CRC, the microcontroller is responsible for generating the CRC data. A 16-bit CRC is generated using a simple algorithm combined with a look-up table [15]. Make sure that the two CRC16 bytes are placed in the right order in the dual-port RAM to ensure that the resultant serial data clocks out correctly.

The Setup transaction expects a handshake after the DATA0 packet is sent. When sending this packet, the expect return flag bit (bit 4 of my `Byte-Count` byte) must be set. If all has gone correctly, the DATA0 packet passed to your µSIE should be [1ch,80h,C3h, 80h,06h,00h,01h,00h,00h,12h,00,E0h,F4h].

The third and final packet in the Setup transaction is the return of an ACK handshake. If you see the '2040 hub return this packet (D2h), then your µSIE is probably working correctly, and you've completed your first two-way USB transaction.

The complete Control Read sequence consists of three more In transactions and an Out transaction. The next step is to write a test routine for your 8-/16-bit micro to complete each transaction (one per 1-ms frame period).

THE FIRST IN TRANSACTION

The first In transaction consists of an IN packet sent by the host controller, followed by a DATA1 packet from the hub. It ends with the host controller returning an ACK handshake packet.

So far, the µSIE can only send packets; it can't receive them. Because we can't read the returning DATA1 packet, the µSIE can't issue an ACK response. But, this returning DATA1 packet makes a great test signal to start the development of the input section of the µSIE.

The first section of the host controller's input to implement is the digital phase lock loop (DPLL). The implementation I chose is basically a copy of the DPLL state machine outlined in the USB white paper [2, p. 2].

There is one known bug in my implementation of this DPLL [16]. Occasionally, when syncing to a faster clock, the DPLL skips the fourth phase clock state. This could pose a problem for any µSIE implementation that, like mine, uses this fourth phase clock signal.

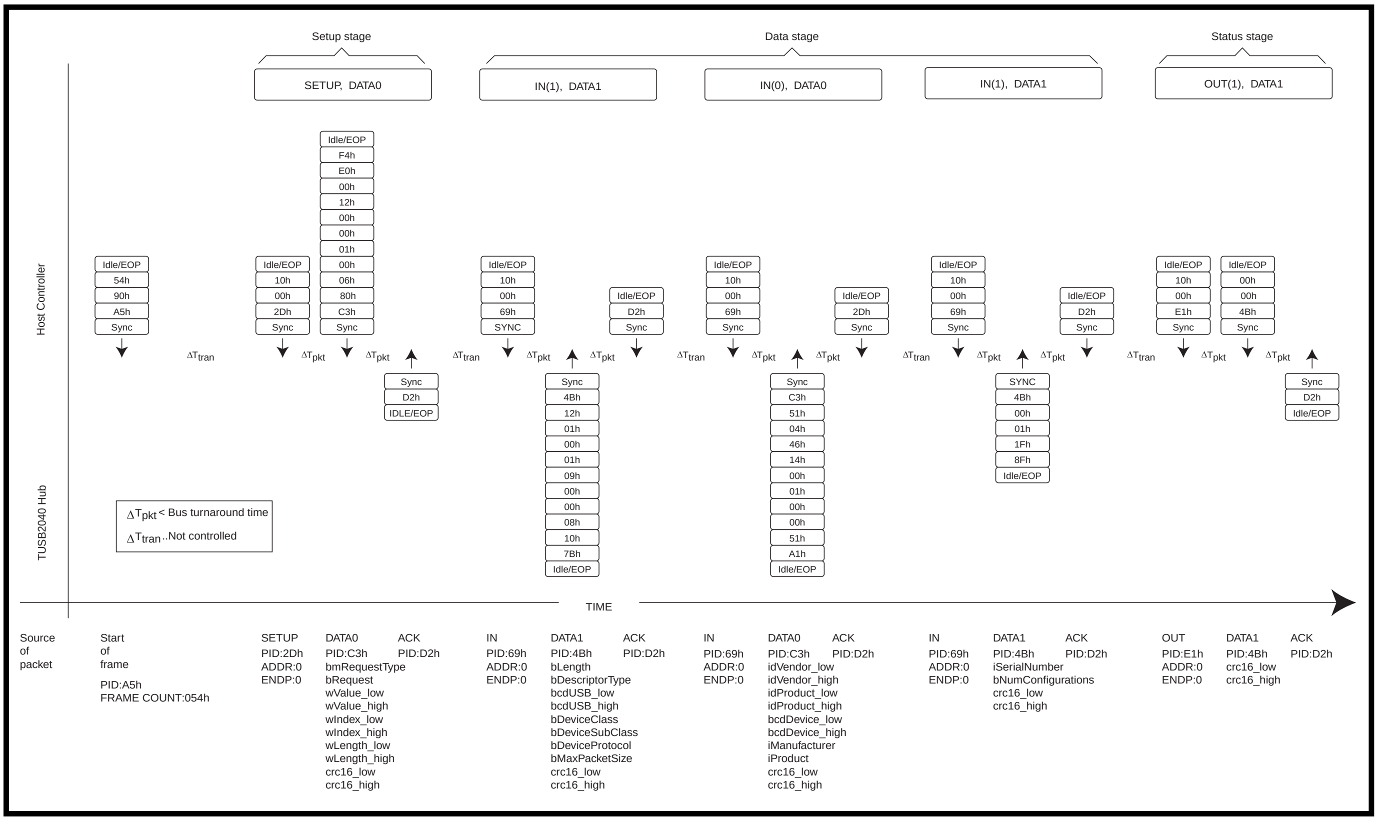
There are ways around this bug, but I haven't spent the time to fix it yet. So far, it hasn't presented any problems because the hub runs off the same 48-MHz clock as the µSIE, and low-speed transactions don't last long enough (in bit times) for this skipping of clock states to occur.

The bus turnaround time detect is a counter clocked at the bus cycle period [17]. The counter is turned on while the DPLL state machine is in its Sync Pattern detect phase. If the DPLL doesn't get beyond its first two states, no Sync Pattern is detected. If this condition persists for 18 bus clock periods, the counter sets a timeout flag [18].

Once a receive clock is established, the NRZI decoding can be done. Recall that the NRZI encodes data as transitions in the serial signal stream. So, to pull the signal back out, the transitions need to be detected.

A two-bit shift register helps accomplish this detection. As the USB signal is clocked in, a comparison is made between the current and previous signal states. If they are equal, a 1 is clocked out; if not, a 0 [19].

Bit unstuffing is just the reverse of bit stuffing. If the number of 1s in a row equals six, the input NRZI decoding state machine is clocked once without clocking the input shift register [20]. This, in effect, throws away



Setup stage
Data stage
Status stage
SETUP, DATA0
IN(1), DATA1
IN(0), DATA0
IN(1), DATA1
OUT(1), DATA1
Host Controller
TUSB2040 Hub
Idle/EOP 54h 90h A5h Sync
Idle/EOP 10h 00h 2Dh Sync
Idle/EOP F4h E0h 00h 12h 00h 00h 01h 00h 06h 80h C3h Sync
Sync D2h IDLE/EOP
Idle/EOP 10h 00h 69h SYNC
Sync 4Bh 12h 01h 00h 01h 09h 00h 00h 08h 10h 7Bh Idle/EOP
Idle/EOP D2h Sync
Idle/EOP 10h 00h 69h Sync
Sync C3h 51h 04h 46h 14h 00h 01h 00h 00h 51h A1h Idle/EOP
Idle/EOP 2Dh Sync
Idle/EOP 10h 00h 69h Sync
SYNC 4Bh 00h 01h 1Fh 8Fh Idle/EOP
Idle/EOP D2h Sync
Idle/EOP 10h 00h E1h Sync
Idle/EOP 00h 00h 4Bh Sync
Sync D2h Idle/EOP
ΔTtran ΔTpkt
ΔTpkt < Bus turnaround time
ΔTtran ..Not controlled
TIME
Source of packet
Start of frame PID:A5h FRAME COUNT:054h
SETUP PID:2Dh ADDR:0 ENDP:0
DATA0 PID:C3h bmRequestType bRequest wValue_low wValue_high wIndex_low wIndex_high wLength_low wLength_high crc16_low crc16_high
ACK PID:D2h
IN PID:69h ADDR:0 ENDP:0
DATA1 PID:4Bh bLength bDescriptorType bcdUSB_low bcdUSB_high bDeviceClass bDeviceSubClass bDeviceProtocol bMaxPacketSize crc16_low crc16_high
ACK PID:D2h
IN PID:69h ADDR:0 ENDP:0
DATA0 PID:C3h idVendor_low idVendor_high idProduct_low idProduct_high bcdDevice_low bcdDevice_high iManufacturer iProduct crc16_low crc16_high
ACK PID:D2h
IN PID:69h ADDR:0 ENDP:0
DATA1 PID:4Bh iSerialNumber bNumConfigurations crc16_low crc16_high
ACK PID:D2h
OUT PID:E1h ADDR:0 ENDP:0
DATA1 PID:4Bh crc16_low crc16_high
ACK PID:D2h

Figure 3—*Here you see an SOF packet for frame count 054h, followed by the Control Read sequence for the TUSB2040's `GET DESCRIPTOR` device request. The actual USB signal is formed by shifting out, least significant bit first, in the order they appear and in NRZI fashion, the bytes shown in this diagram.*

the stuffed bit. Just a reminder here: input clocking is a shift-right least-significant-bit-first operation.

After you have a decoded signal, you need to start looking for the Sync Pattern. Separating it from the data signal and detecting its end will enable the µSIE to lock to the correct timing for reading data bytes out of the serial input shift register.

The scheme I decided upon was to look for the value 8h in the upper four bits of the input shift register [21]. But, you can't start looking too soon because the first few bits to come through the decoding process may be interpreted as a 1 preceded by 0s.

The first segment of the hub's descriptor comes as the following DATA1 packet of Sync Pattern, PID, data and crc16: [80h,4Bh,12h,01h,00h, 01h,09h,00h,00h,08h,10h,7Bh]. If this is the data output you see from the '2040 (via your µSIE), then your input section is working correctly.

The EOP SE0 state is detected using a counter clocked in the 4× domain. For example, with 12-MHz full-speed transactions, the counter is clocked at 48 MHz [22].

The counter is gated by the detection of a SE0 state. If the counter detects two full bus clock periods, the `EOP_flag` is set [23]. In my implementation, the ending Idle state is not detected.

Because a host controller isn't allowed to return either a NACK or STALL handshake, the µSIE doesn't have to implement these functions [1, p. 151]. The only transaction responses the host controller can return are an ACK handshake for transactions that are completed correctly and a no-handshake return for transactions that don't complete correctly.

The simplest choice is for the host controller to acknowledge everything and let the microcontroller take care of error checking and repeating a transaction if and when errors occur. Since I chose this option, the handshake packet need not originate with the µSIE but can be generated by the 8-/16-bit micro and loaded into the dual-port RAM with all the other packets in a transaction. An ACK packet is just the SYNC and PID bytes: [02h,80h,D2h].

To finish the In transaction, the host controller must issue an ACK handshake. If the EOP for the DATA1 packet has been detected successfully, the µSIE reads in the next packet to send from the dual-port RAM (i.e., the ACK packet.

The final transaction of any Control Transfer is the Status Stage, which is nothing more than the exchange of an empty DATA1 packet. Basically, it gives the sending party a chance to send a parting ACK or NACK response back to the receiving end [1, p. 155].

BURPING THE µSIE BABY

If all has gone right so far, you now have a µSIE correctly locking on to the incoming USB signal and outputting data bytes in a coherent fashion from its serial-in parallel-out shift register. The next step is to get this data into the dual-port RAM in a way that is readable by the 8-/16-bit micro.

There are two address counters in the dual-port RAM address logic—one for reading output data, one for writing input data. When the µSIE goes into receive mode, it captures the current write address but doesn't write to it. It then clocks the write address counter.

As the µSIE inputs data and writes it out to the dual-port RAM, it keeps a count of the total number of bytes received. When the EOP is detected, the µSIE writes that byte count out to the address it saved at the beginning, thus creating a semi-linked list with the same structure as the input data.

CONFIGURING THE USB HUB

The Control Read sequence simply wakes the '2040 up. It still has to be configured before it can be used for downstream communications [1, p. 167]. Configuration is done by issuing the `SET CONFIGURATION` device request and setting the configuration value to 01h.

In general, `SET X` device requests are Control No-Data transactions. Setting this configuration value to the nonzero constant corresponding to its hub configuration gives the host access the '2040's hub controller functions.

Once these functions are accessed, the host controller, using an Interrupt In transaction, can read the Port Status Change bit map at the '2040 hub's endpoint 1 to see if any devices are attached [1, p. 262]. A `GET PORT STATUS` request returns specific information about a port's attached device.

The `SET PORT FEATURE` request is then issued several times with different wValues until the hub and device are ready. For example, wValue 04h is reset, wValue 01h is port enable, wValue 08h is power on, and so on [1, p. 254].

Configuring the USB hub is an example of bus enumeration [1, p. 169]. At startup, all USB devices require the same initialization and configuration sequences and share almost all of the same standard device requests.

When it detects the presence of a new device on the USB, the host controller enables an initial 100 mA of current to bus-powered devices. Next, there is a wait period for the attached device to power up and be ready to accept communications [1, p. 242]. This period is specified by `bPwrOn2PwrGood`, which is returned as part of a hub's Hub Descriptor [1, p. 250].

A `RESET` device request is sent to the hub controller the device is attached to. The hub then issues a standard USB `RESET` control signal, which is at least 10 ms of the SE0 state [1, p. 119].

Following a reset, the first device request issued is always `GET DESCRIPTOR` at ADDR 0 and ENDP 0. Then, the `SET ADDRESS` request is sent, changing the default ADDR to a new unique nonzero value. `GET DESCRIPTOR` is issued again at the new address.

Configuring the attached device starts with the host controller issuing a `GET CONFIGURATION` request. If this value is zero, the device is not configured and a `GET DESCRIPTOR:CONFIGURATION` request is sent [1, p. 184].

The returned variable, `bConfigurationValue`, is used for the subsequent `SET CONFIGURATION` request. Once the configuration value is set, you have access to all device requests specific to that configuration class. USB devices may support more than one configuration, so you must specify a particular configuration value during device initialization.

The final step is to enable full power to those devices requiring more than the initial 100 mA of current.

BACK TO USB SCHOOL

Not every device supports every USB device request. There's a lot of information in chapters 8, 9, and 11 of the USB Specification that may be essential for communications with your USB device: for example, variables like `bInterval` (the interval period, in frame counts, between device accesses) or `bPwrOn2PwrGood` (the time for a device's power supply to ramp up).

Experiment with the standard device requests by issuing them to the '2040. Having feedback from an actual USB device and comparing its responses to the Specification's text will help you make sense of the material.

Low-speed transactions are prefaced by a special PID (`PRE`) sent at full speed. The µSIE doesn't append the usual EOP and Idle states when sending this low-speed preamble, but (after a pause of four bus cycle periods to give downstream hubs time to reconfigure) goes from the `PRE` PID to the Sync Pattern of the next low-speed packet [1, p. 160].

ON YOUR OWN

Now, plug a low-speed USB device into the host controller, power up, enable the device's features, and start teaching the attached micro to speak USB. Once the host controller is up and running, read the '2040's Port Status Change bit map. A NACK response indicates no change on the input ports.

Next, plug a low-speed USB device into the host controller. Doing a second read of the Port Status Change bit map should get a byte of data back with the bit set corresponding to the port you plugged into.

Check the port's status to see what has changed, power on the device, do a reset, wait 10 ms, then start the bus enumeration process. If necessary, get configuration values and do a set configuration to enable the device's features.

What those features are and how you talk to the USB device after configuration is device-specific [1, p. 34]. So, from here on out, you're on your own.

I hope you'll take up the USB mantle, create your own embeddable low-speed USB host controller, and make your work available as HDL shareware. Happy USB'ing. ■

Glen Reuschling is a manufacturing engineer by day and a serious hobbyist by night. His most recent home project resulted in this article. You may reach him at wildiris@cruzio.com.

SOFTWARE

Programming for the three CPLDs was done using the hardware description language CUPL. Source code is on the *Circuit Cellar* web site.

REFERENCES

[1] USB Implementers Forum, *Universal Serial Bus Specification*, Rev.1.0, www.usb.org, 1996.
[2] USB Implementers Forum, *Designing a Robust USB Serial Interface Engine (SIE)*, www.usb.org.
[3–23] see `REFS.TXT` on the Circuit Cellar web site.

SOURCES

ispLSI1016E, **ispLSI1032E**
Lattice Semiconductor Corp.
(503) 681-0118
Fax: (503) 681-3037
www.latticesemi.com

TUSB2040
Texas Instruments, Inc.
(972) 644-5580
Fax: (972) 480-7800
www.ti.com

A Familiar Face

Jeff Bachiochi

Driving 144 LCD Segments I^{2}C Style

As the I^{2}C bus rolls into its third decade, Jeff puts the Philips technology to use in an eight-character LCD. Get onboard to see how to cascade a couple I^{2}Cs to get the right number of segments for this display.

although the I^{2}C bus has almost reached drinking age, it's far from the peak of product maturity. This fact is reinforced by the number of manufacturers adding an I^{2}C port as a standard peripheral device to their micros.

It's not that you can't create a software I^{2}C port on practically any micro; it's been done that way for years. But, by having hardware support built in, communications no longer requires gobs of processor overhead.

Philips' Inter-Integrated Circuit (I^{2}C) concept has blossomed into more than 500 compatible devices developed by more than a dozen manufacturers. Even so, Philips remains the number-one I^{2}C manufacturer, and their peripherals include memories, data converters, I/O ports, real-time clock/calenders, DTMF generators, and LCD drivers.

This month, I use cascaded drivers to create an I^{2}C interface to an eight-character LCD. This display is similar to those found at many gasoline pumps and consists of 18 segmented characters. As you can see in Figure 1, each character includes a decimal point and an apostrophe.

The glass is constructed using three backplanes (common to all characters) as well as six frontplane inputs for each character. Each frontplane input consists of three character segments. That's a total of three segments × six FP inputs × eight characters, which equals 144 character segments.

The Philips PCF8566 can drive up to 96 segments (using four backplane commons). Because my display uses only three backplanes, the segment capacity of the PCF8566 in 1:3 mode is ¾ of 96 (i.e., 72 segments). By using two '8566s, I can get just the right fit—72 × 2 = 144 segments.

Ordinarily, this would cause all kinds of synchronicity problems. But as you will see, these problems have all been dealt with quite nicely.

THE CIRCUIT

After power has been applied to the '8566s, all segments are initialized to the off state and the 1:4 drive mode is selected, the I^{2}C is initialized, and the internal data pointers are all reset. The '8566 has a built-in oscillator that is enabled by tying the OSC pin to ground.

In this configuration, the clock is output via the CLK pin. If OSC is tied to V_{CC}, the CLK pin becomes a clock input pin. Thus a second device can slave off of the clock outputted by the first device.

A power-savings mode reduces the clock by a factor of six but retains the same minimal frame rate of 64 Hz. The lower clock speed means that I^{2}C communications will suffer a loss in speed, but this low-power mode can be changed at will.

Just because both devices are being clocked from the same oscillator does not mean that all timing will necessarily be in sync. A SYNC pin assures that all devices stay synchronous. Each device monitors this pin as an input until the beginning of the last active backplane signal, at which time it outputs a falling sync pulse.

While in the input mode, if a device sees the SYNC signal go low, it synchronizes its frame with that SYNC pulse. This arrangement ensures that segment data will not be out of step with any other device (especially important to the master, which supplies the backplane signals to the glass).

The '8566 has an internal LCD bias generator consisting of a resistor divider (and voltage follower/buffers) between V_{dd} and the V_{lcd} input pin. The V_{lcd} can be driven from some external temperature-compensation circuitry if necessary. For nominal environments, only a contrast pot is necessary.

ADDRESSING

Most I²C devices have a predetermined address consisting of most significant fixed address bits and three user-addressable lesser significant bits. Usually A0–A2 are tied high or low to give each device one of eight possible addresses.

The '8566 has only a single address bit, SA0. When cascaded, each device in the cascade uses the same address (logic input on SA0) so each device will receive every command. Remember that these are all on the same bus, not daisy-chained.

Depending on how the device is configured, it may require from 24 data bits (for a 1:1 or static display) to 96 data bits (for a 1:4 display) to fill the display RAM. The next bit will want to go to the second device.

So, how does a device know if the data is meant for it? There are actually two counters in each device—the data pointer and the subaddress counter. The data pointer keeps track of the bits received and, based on the configuration, bumps the subaddress counter when 24, 48, 72, or 96 bits overflow.

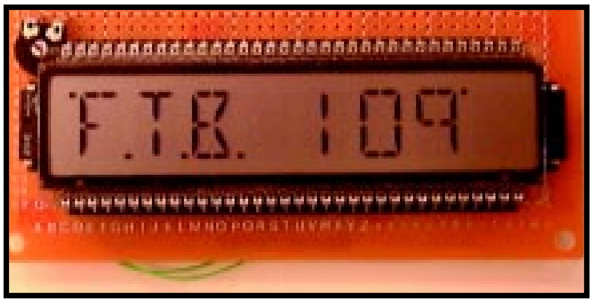


Photo 1—*Two I²C PCF8566 LCD segment drivers can drive all 144 segments on this eight-character display. The driver ICs can be seen behind the LCD's glass.*

Each device has three subaddress input bits, A0–A2. The user sets all three address input bits low on the master (subaddress = 0). The second device is assigned subaddress = 1 (001b), and so on.

As devices receive data (because SA0 is the same), only the device with the matching subaddress will store the data. Up to eight devices can be cascaded in this fashion.

When fewer than eight devices are used, the data pointer and subaddress counter must be reset to zero on sending the last piece of data. Otherwise, the following data will go into the bit bucket, as the subaddress will no longer match any devices in the circuit.

REGISTERS

Many I²C peripherals have multiple registers. Usually these registers are accessed using a three-byte sequence. The first byte holds the device address (to whom), the second byte is the register number (where the data goes to/comes from), and the last byte is the data (what it is).

The five registers in the '8566 can be addressed and have data written to them all within the second byte. Although doing this requires only a two-byte sequence, multiple commands can be contained in the same sequence, as well as data.

Table 1 for the functions of the five command registers. By setting the C bit to a 1, multiple register commands can be sent in the same transmission sequence. Also, note that a transmission sequence must have at least a single register command before any data is sent. On receiving any data bytes, the data pointer and device select registers are automatically incremented.

SHOW ME

One of the nice points of developing with I^{2}C is the flexibility of combining a number of small circuits that may have been developed independently into unrelated configurations. It's like writing modular code that can be used repeatedly, thus saving precious time.

Mode Set:			C 1 0 LP E B M1 M0	
	where	C =	0	last command
			1	commands continue
		LP =	0	normal speed
			1	power saving slow speed
		E	0	display off
			1	display on
		B	0	1/3 bias
			1	1/2 bias
		M1 M0	00	1:4 backplanes
			01	1:1 static
			10	1:2 backplanes
			11	1:3 backplanes
Load Data Pointer:			C 0 0 P4 P3 P2 P1 P0	
	where	C =	0	last command
			1	commands continue
		P4–P0	5-bit binary value 0–23 display RAM address	
Device Select:			C 1 1 0 0 A2 A1 A0	
	where	C =	0	last command
			1	commands continue
		A2–A0	3-bit binary value 0–7 subaddress counter (device address)	
Bank Select:			C 1 1 1 1 0 I O	
	where	C =	0	last command
			1	commands continue
	when	MODE=1:1 static		
		I =	0	to RAM address bit0
			1	to RAM address bit2
		O =	0	from RAM address bit0
			1	from RAM address bit2
	when	MODE=1:2 backplanes		
		I =	0	to RAM address bit0&1
			1	to RAM address bit2&3
		O =	0	from RAM address bit0&1
			1	from RAM address bit2&3
	when	MODE=1:3 backplanes		
		not used		
	when	MODE=1:4 backplanes		
		not used		
Blink:			C 1 1 1 0 A BF1 BF2	
	where	C =	0	last command
			1	commands continue
		A =	0	normal blink
			1	alternation blinking (see Bank Select register)
		BF1 BF0 =	00	off
			01	2 Hz
			10	1 Hz
			11	0.5 Hz

Table 1—*Only five registers are needed to completely control each PCF8566. Notice that the first bit of each command can indicate that additional commands will follow in the same I^{2}C string. The control word, which appears on the first line of each register description, is shown so that the most significant bit is leftmost and the least significant bit is rightmost.*

To get a quick feeling of success, I grabbed a PicStic-1 (although any PIC could have been used) and wrote a short program with PicBasic's built-in I²C routines. PicBasic's routines don't rely on hardware, so any of the I/O pins can be used to create an I²C bus.

Under normal circumstances, I'd create lookup tables for a complete character set, but because I just wanted to get something up on the display, I opted for a shortcut. I blew off the table in favor of a few constants for those characters I wanted to display.

After computing the constants based on the segment positions in the RAM registers, I compiled, downloaded, and ran the program. Yes, I had a display, but there were gaping holes among the characters (missing segments).

Listing 1—*Very little BASIC code was needed in order to see to some results from the circuit shown in Figure 1, thanks to the PicBasic compiler.*

```
' I2WRITE to 8566 LCD driver
' Write a command followed followed by optional data
'
symbol  ADDR=$3E       'actual address byte (shifted right once)

START:  B0=$48         ' MODE register value
  B1=B0+128            ' same w/high bit set
  I2COUT ADDR,B1,(B0)  ' Write only command (required syntax)

LOOP: FOR B2 = 0 TO 23
  LOOKUP B2, ($80,$AC,$8,$20,$8,$A8,$64,$AC,$4A,$20,$0,$0,
      $0,$0,$A0,$4E,$A8,$8,$A,$8C,$C,$80,$0,$0),B3
  I2COUT ADDR,B0,(B3)  ' Write data
  NEXT B2
  END
```

ANALYZE THIS

The LCD is an 8540M3-RPH-0.25, a standard product of Excel Technologies. Refer again to Figure 1 to see how each character is divided into six three-segment connections. Connections to these frontplane segments determine the order of the input data to the '8566.

The display RAM for a 1:3 mode (three backplanes) is three bits deep. Each character requires six data RAM addresses (three bits each). The character data is of the format: AP DP D2 B C D1 A2 J G2 I L K A1 H G1 F E M. Notice that it takes 18 bytes of data for a full eight characters.

I pored over the constants; they seemed to be correct. I looked back at the datasheet and found a snag. My display had three backplanes. The '8566 driver's example showed a seven-segment display with decimal points connected using three backplanes. This setup required eight bits going into three 3-bit RAM registers. That's nine bits of storage.

To make it easy to use, the designers decided to skip the last bit. In other words, in the 1:3 mode, the next data didn't start where the last left off but skipped the ninth location. Unfortunately, that ninth bit has a segment on my display associated with it. Not only did I get some blank segments, but my data was shifted one bit for each byte of data. Whoa.

Fortunately, I was able to save the day by using the 1:4 (four backplane) mode. When the '8566 is initialized for 1:4 mode, the RAM registers become four bits deep. In the 1:4 mode, the eight bits of the seven-segment display (with decimal points) fit exactly into the RAM registers without any need to toss out any bits, so I can pad my data to put don't cares every fourth bit.

Now the character data format is AP DP D2 *x* B C D1 *x* A2 J G2 *x* I L K *x* A1 H G1 *x* F E M *x*. Notice that it now requires 24 bits for each character.

You can see from BASIC Listing 1 how easy it was to get some test code written. This padding maneuver cleaned up the display quite nicely (see Photo 1). The error in misunderstanding the datasheet could have been disastrous if I hadn't been able to fix it in software and if the prototyping stage had been skipped. Designing straight to fab can be costly.

I think I'll add a few lines of code so the message can be scrolled. There are many possibilities now that I have a simple platform to work with. By adding a serial EEPROM to the PicStic I can prepare some canned messages and swap them in and out just by replacing the eight-pin DIP.

The PicStic has plenty of I/O pins left, so I can connect a PC serial port to a couple of pins used as a software serial port. Then the same BASIC program can program an external serial EEPROM with a message and display it as well.

Someday I'd like to design a project using some custom glass, but until the quantities make that practical, I'll stick with standard LCDs. ▣

Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on Circuit Cellar's *engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

SOURCES

PCF8566
Philips Semiconductors
(408) 991-5207
Fax: (408) 991-3773
www.semiconductors.philips.com

PicBasic
microEngineering Labs, Inc.
(719) 520-5323
Fax: (719) 520-1867
www.melabs.com

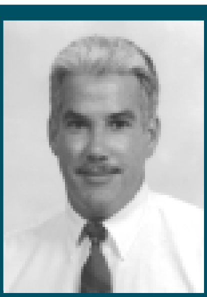
PicStic-1
Micromint, Inc.
(800) 635-3355
(407) 262-0066
Fax: (407) 262-0069
www.micromint.com

8540M3-RPH-0.25
Excel Technologies Int'l Corp.
(908) 874-4747
Fax: (908) 874-3278

SILICON UPDATE

Tom Cantrell

DSP Doings



Quick! Think of the latest in silicon! Did you think of DSPs? If not, you're not alone, but the suppliers are already forging ahead in this high-stakes market. Check out how the competition is heating up.

It's easy to underestimate the role of DSPs. They are so deeply embedded that their presence often goes unnoticed. But, that's not to say the market for DSPs isn't thriving.

Just tally up all the gadgets we rely on, such as modems, cellular phones, audio/video, disk drives, and motors. You'll soon realize that with volume rapidly approaching a billion units a year, the DSP biz is nothing to sneeze at. Of course, as usual, the silicon wizards are serving up more MACs (multiply-accumulate operations, at the core of most DSP routines) than McDonalds.

Outside the lab, there are a lot of machinations taking place in the boardrooms of the major players, not surprising given the high stakes on the table. Top suppliers TI, Lucent, Motorola, and Analog Devices (together composing almost 80% of the DSP market, according to Dataquest) are all making strategic alliances and acquisitions to position for the next wave of growth.

TI MARCHES ON

With a third of the market share, TI remains the leader in DSP, as they have for years. Judging by the barrage of new parts, they have no intention of coasting.

TI's offerings span a broad spectrum from the low-cost MCU-like 'C2*x* family to the rocket-science VLIWs of the 'C6*x* line. In between are the 'C3*x* entry-level floating-point models and the midrange (100+ MIPS these days) 'C5*x*. There's action all across the board.

Each family covers a range of price and performance. For instance, the 'C6*x* lineup is best known for superchips like the 'C6202, featuring megabits of on-chip memory, hundreds of pins and megahertz, and $100+ price tags.

Yet now, with the introduction of the 'C6211 and 'C6711 (fixed and floating point, respectively) you can get on the VLIW bandwagon for as little as $20 or so in volume.

These chips contain the same big V8 motor (i.e., execute up to eight instructions at once) as the upscale models (see Figure 1). But instead of the multimegabits of on-chip SRAM that the luxo-models include, the 'C6*x*11s rely on external memory.

Of course, this raises the small matter of memory bottleneck. Hmm, need to feed up to 256-bit long instructions at 150 MHz through a 32-bit data bus? Sounds like a problem, unless you've got a good source of <1-ns memory chips.

The tried and true answer is cache, and the TI chips adopt the two-level scheme that's so popular with the desktop CPU crowd. Separate 32-Kb L1 program and data caches (the former direct-mapped and the latter two-way set associative) talk to a 512-Kb L2 unified program and data cache.

According to TI, the cache hierarchy delivers up to 80% of the performance of a traditional on-chip memory solution. However, designers need to be aware of the determinism issues associated with cache (e.g., the possibility of sample timing jitter). The chip does permit all or a portion of the L2 cache to work as RAM, which is one way to avoid any unwanted cache side effects.

Although 'C6*x* grand-challenge designs are exciting, DSPs are also compelling when it comes to workaday world apps like motor control. In fact, TI predicts that of the billions of motors that ship every year, fully 10% are candidates for digital controls.

Hey, motors have worked fine for years without DSPs, so what's the big deal? Turns out, the precision and high-speed control capability of a DSP can directly translate into major system-cost reductions.

For example, many motor subsystems need big, and expensive, capacitors to deal with mains ripple. A DSP and a clever designer can cut capacitor cost by compensating in software.

Similarly, the PWM outputs can use spread-spectrum techniques to reduce EMI and the cost of suppressing it. According to TI, a DSP-based design can cut the cost and size of a washing machine drivetrain by one third.

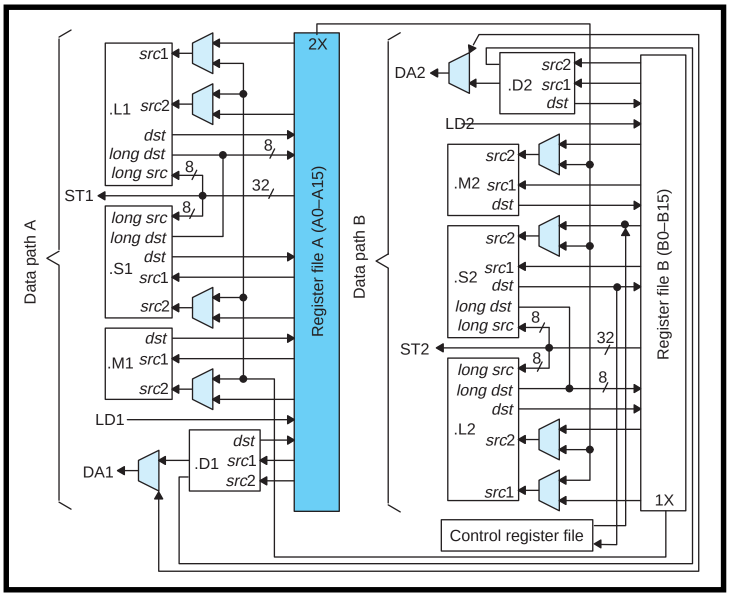


Figure 1—*In 1997, the TI 'C6x eight-instruction/clock VLIW set the standard for performance-at-any-price DSPs. Now, you can get one for $20.*

known as the M-Series. They include the ADSP-2185M, '2188M, and '2189M with 0.66, 1.5, and 2.0 Mb of on-chip RAM, respectively. At only $7.50 (25k quantity), the '2185M seems like quite a bargain, considering the on-chip SRAM and 75-MIPS performance.

The '218*x* parts are well-suited for portable apps, with low-voltage operation (2.5-V core, 2.5- or 3.3-V I/O), multiple power-reduction modes, and optional 144-pin ball grid array (BGA) packaging that cuts board space to 1 cm².

Needless to say, a part intended for washing machines has to be lean and mean on all fronts. Unlike high-end designs, the DSP can't just live in the clouds contemplating inner loops and expect other worker-bee chips to pick up the control slack. It has to combine the number crunching of a DSP with the practical functions and peripherals of an MCU.

As far as TI's concerned, the result is something like the TMS320F241 in Figure 2. Like other highly integrated 16-bit fixed-point DSPs, the '241 wraps the DSP core (multiplier, shifters, and various RAMs) in a collection of MCU-like peripherals including 10-bit ADC, clocked serial port, UART, timers (including unique deadband and quadrature encoder features), and even a CAN module.

Carrying the MCU mimicry to its logical conclusion, the '241 even incorporates 8K × 16 flash memory. Don't worry about fancy packages and fussy specs, the '241 comes in a plain 68-pin PLCC or 64-pin QFP and extended temperature (–40° to +85° and even –40° to +125°C) versions are available.

All together, TI offers about a dozen variants in the '24*x* family. Also, one of their partners, Technosoft, offers the MotionChip, which is a '24*x* DSP preprogrammed with complete library of motion-control software that handles all manner of motors (DC brush and brushless, AC, induction, stepping, etc.) and feedback and control algorithms. Setup and tuning are performed with easy-to-use visual software (e.g., tweak accel/decel ramp onscreen) running on a PC.

DSP INSIDE?

Analog Devices is extending the lineup of their popular ADSP-21*xx* 16-bit fixed-point parts with three new pin- and code-compatible '218*x* parts

As popular as it is (Analog Devices claims 30,000 programmers have signed up over the years), the 10-year old '21*xx* family is getting a bit long in the tooth. Enter the new '219*x* family (see Figure 3), which is expected to debut later this year. Upward code compatible with the '218*x*, the '219*x* incorporates a number of features designed with C in mind, including bigger address space and extra addressing modes.

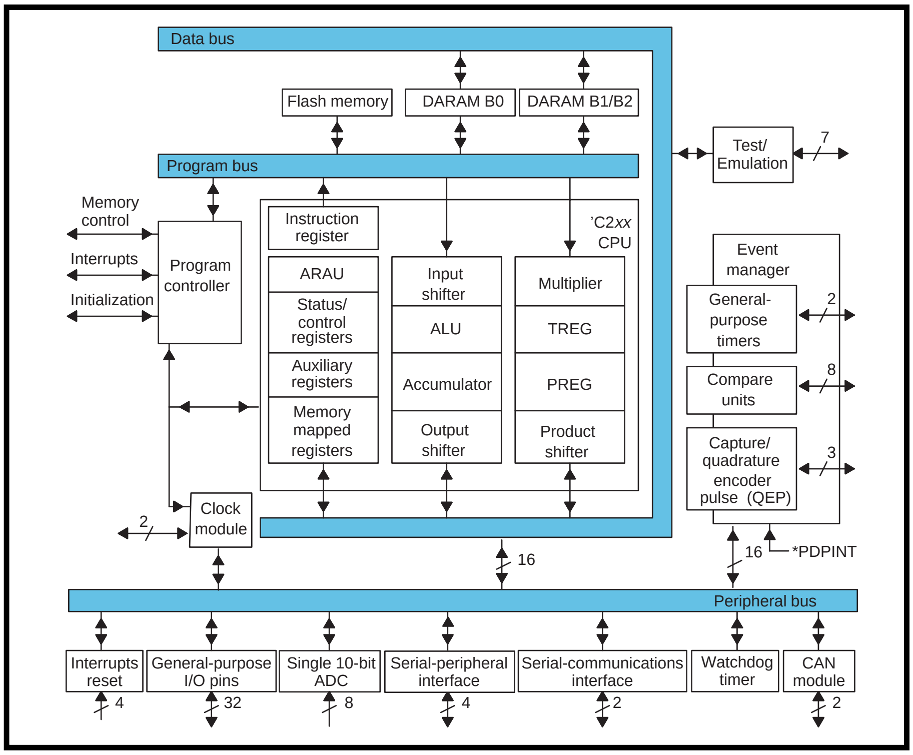


Figure 2—*Is it an MCU or a DSP? Only the programmer knows for sure. The TI 'F241 goes beyond the usual multiplier and RAM by incorporating all the features of an MCU, including flash memory.*

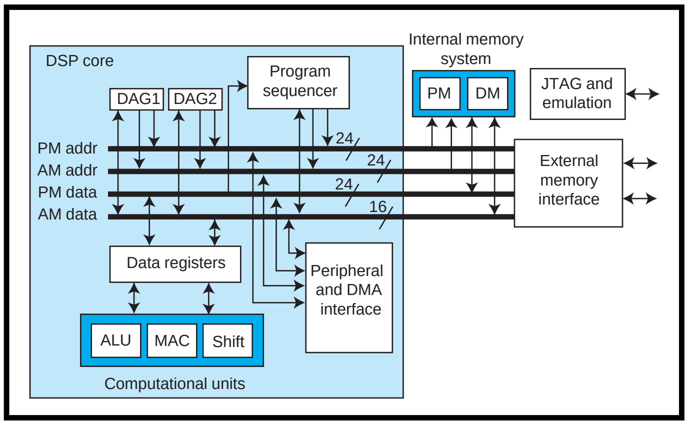


Figure 3—*The Analog Devices '219x upgrade will keep longtime '218x customers happy while they (and everyone else) wait anxiously for a look at the forthcoming joint ADI/Intel design.*

Thanks to Analog's acquisition of Edinburgh Portable Compilers, a new C compiler is in the works that optimizes register usage to reduce pressure on the local stack and includes intrinsic support for fractional and complex data. Also, the purchase of White Mountain DSP ensures timely emulator support, made easier by the incorporation of a JTAG-based test and debug port.

For the performance-at-any-price crowd, the Analog Devices SHARC was the chip that kicked off the subsequent wave of SuperDSPs when it was introduced five years ago ("When the SHARC Bites," *Circuit Cellar* 45). Now, like TI, Analog Devices is making moves at the low end of the high end with the announcement of the $10 '21065XL, notable for packing a lot of punch in a tiny package (see Photo 1a).

Meanwhile, at the highest of high ends, the DSP story is all about multiprocessing. Consider Bittware's Goblin, which packs up to seven SHARCs (each with its own 64 MB of DRAM for a total of 448 MB!) onto the CompactPCI board you see in Photo 1b.

Another intriguing development at Analog is the announcement of a joint development deal for a new 16-bit fixed-point design with, of all people, Intel. After all, it was Intel who first promulgated the controversial native signal-processing pitch that boils down to "we don't need no stinking DSP."

Maybe Intel plans to combine the new DSP with a CPU core ('*x*86 or StrongARM). After all, TI is apparently doing well in the cellular-phone biz with the hybrid ARM7 CPU + 'C5*x* DSP chip. Or, is Intel thinking it might be good to take some of their eggs out of the PC basket?

A STAR IS BORN

Let's make a deal is also the name of the game at Lucent and Motorola. The number-two and -three players (respectively), whose combined share matches that of number-one TI, are joining forces.

Their Star*Core joint development initiative intends to deliver a new high-end architecture and requisite tools (especially C compiler) that both companies can incorporate into their individual product lines. Of course, neither company will abandon their currently popular architectures.

In fact, the agreement includes cross-licensing between Lucent and Motorola of each company's existing DSP cores ('16*x* and '56k, respectively). And, Lucent picks up Motorola's M•Core embedded RISC.

Although the initial SC140 parts (see Figure 4) won't be available until next year, the released information indicates that the Star*Core designers have been paying close attention to technical and market trends (er, not to mention TI's 'C6*x*). Able to execute six instructions per clock (including four MACs) and running at up to 300 MHz, the SC140 is capable of crunching numbers at a formidable rate, up to 1.2G MAC/s with on-chip data bandwidth of up to 4.8 GBps.

Interestingly, the SC140 designers are sensitive to the pragmatic design constraints. For instance, all hand-held communications gear suffers from battery-itis (i.e., battery life is never long enough, even though the battery always seems too big from a packaging and cost point of view).

Star*Core attacks power consumption on both electrical and architectural

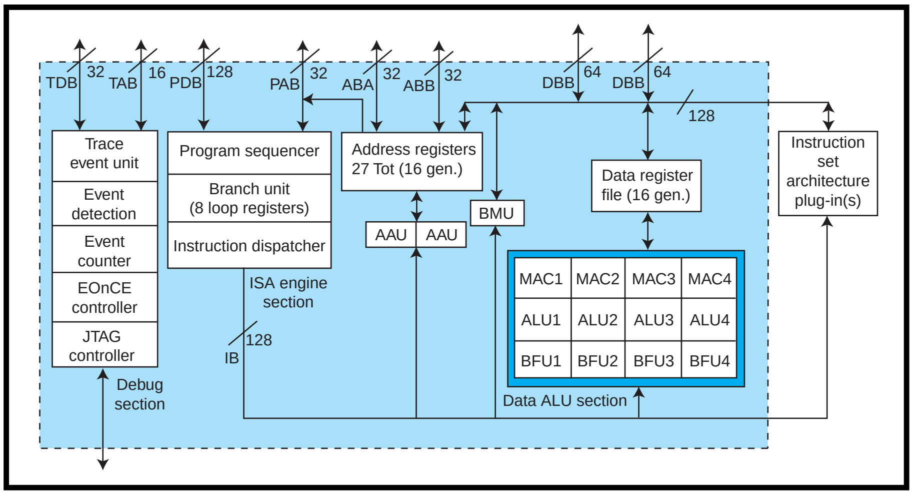


Figure 4—*Two plus three equals one? That's the new DSP-market math lesson that the number-two and number-three suppliers, Lucent and Motorola, want to teach number-one TI with their new Star*Core joint venture.*

fronts. The easiest way to brute-force chip power consumption is to reduce the operating voltage, so Star*Core is specified for 1.5-V operation. It even works with a measly 0.9 V for apps that can get by with a "slow" (120 MHz!) clock. Standby life is extended with low-power modes and circuit design that minimize static power consumption.

VLES IS MORE

The performance versus efficiency tradeoff revolves around the fact that performance is largely driven by highly parallel execution of the 20% of code composed of inner loops, yet overall code size is dominated by the remaining 80% of serial control code.

The VLIW approach, as exemplified by TI's 'C6*x* (and Intel's forthcoming Merced, if you peer behind the '*x*86 CISC curtain), has emerged as the best way to achieve top performance. VLIWs rely on the compiler to extract maximal parallelism from a program, rather than relying on complicated and expensive on-chip superscalar logic.

The compiler can peruse the entire program at leisure (at least up to the point where compile times get painfully long). But, on-chip logic is limited to a small window and can't be overly sophisticated, lest critical paths and overall clock rate suffer.

Relying on the compiler to parallelize the program is the way to go, but nobody said it's easy. The fancy optimizations (e.g., software pipelining that overlaps execution of multiple iterations of the same loop) call for some real headscratching.

As the Star*Core designers point out, their complement of full-featured and autonomous function units makes the compiler's job easier by avoiding the resource bottlenecks and scheduling restrictions that characterize more specialized designs.

The problem with the classic VLIW approach is the 80% of non-DSP control code that offers little opportunity for parallelism, nor especially misses it (i.e., handling buttons, screen, and power-up self-test usually aren't a bottleneck).

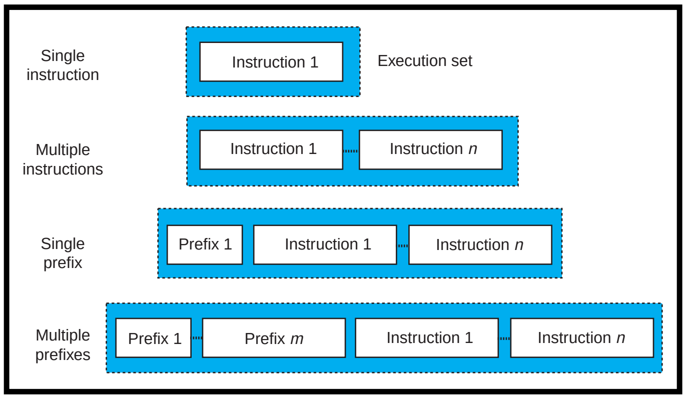


Figure 5—*The Star*Core variable length execution set (VLES) further refines the VLIW concept. The compiler schedules from one to six 16-bit instructions, with one or two optional prefixes, for parallel execution.*

Star*Core uses a hybrid RISC/VLIW approach called variable length execution set (VLES). VLES starts with minimalist 16-bit instructions packed in groups of one to six, with one or two optional 16-bit prefixes, as you can see in Figure 5. That arrangement makes the longest possible instruction word 128 bits (compare to the TI ’C6*x* approach of 8 × 32 = 256-bit instructions).

Its designers claim that Star*Core not only achieves better code density than other high-end DSPs, but it also matches the code density of popular embedded RISCs like ARM.

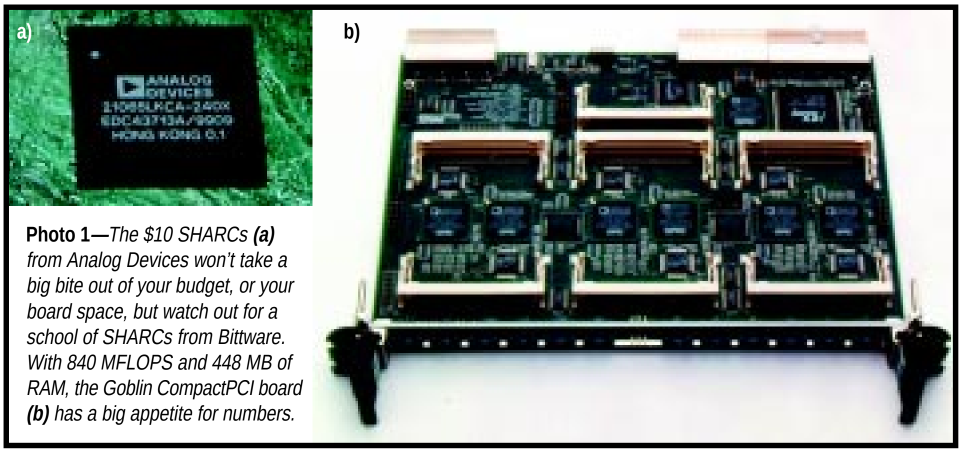


Photo 1—*The $10 SHARCs* ***(a)*** *from Analog Devices won’t take a big bite out of your budget, or your board space, but watch out for a school of SHARCs from Bittware. With 840 MFLOPS and 448 MB of RAM, the Goblin CompactPCI board* ***(b)*** *has a big appetite for numbers.*

WORLD BEYOND DSP?

At the DSP Expo there was a panel session called “The Future of DSP: Separate or Merging Architectures.”

My opinion is they’re already (and have always been) more merged than you might think. “Regular” computers were doing digital signal processing long before the first DSP hit the street. As far as I’m concerned, any system with an ADC or DAC is doing signal processing, regardless of what the big chip has stamped on it.

The Star*Core folks claim the SC140 is higher performance than the ’C6*x*, but guess what: Motorola makes the same claims for the PowerPC with AltiVec DSP extensions. Much like GM uses the same underpinnings for a Chevy, Olds, and Buick, the difference between CPUs and DSPs increasingly boils down to little more than chrome and tailfins.

Talk about convergence, now you can even put a DSP on the web. Precise Software Technologies just introduced a TCP/IP stack with all the trimmings (PPP, SNMP, HTTP, SMTP, and POP3) for the TI ’C54*x*, and have announced plans to port to a variety of other DSPs.

Although the difference between a RISC with MAC and a DSP with a C compiler may be more a matter of marketing than anything else, one approach to DSPs is quite different—doing it with FPGAs. The interesting thing about using FPGAs for signal processing is that it inspires novel techniques, rather than just throwing multipliers and megahertz at the problem.

The IEEE Field-Programmable Custom Computing Machines (FCCM) conference, which covers architectures, tools, and applications (including DSP), is an exciting and fun place for FPGA aficionados. One paper, “Field Programmable Gate Array Based Radar Front-End Digital Signal Processing” by T. Moeller and D. Martinez of MIT, related the authors’ experiences evaluating various FPGA implementation techniques for a 512-tap FIR application currently served by an ASIC.

Duplicating their ASIC parallel multiplier-based design wouldn’t work, even with the largest FPGA. But in the authors’ words (and a valuable lesson for would-be FPGA users), “By more carefully examining the FPGA structure and optimizing an architecture for that structure, a more optimal design was produced.” To wit, a distributed arithmetic approach that exploits the FPGA look-up tables in Figure 6 and copious local interconnect resources can cut the required FPGA’s size in half!

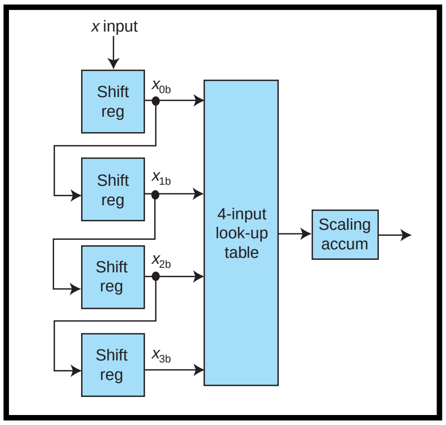


Figure 6—*Distributed arithmetic uses each look-up table (LUT) in a Xilinx FPGA to divide and conquer the MAC (multiply-accumulate) function at the core of DSP inner loops.*

Xilinx knows a good thing (a DSP market growing 25% per year) when they see it and offers an ever-growing list of DSP functions via the LogiCORE program (see "FPGA Tool Time," *Circuit Cellar* 94), including distributed and conventional arithmetic filters, constant coefficient and conventional multipliers, correlators, FFT/DFT, Viterbi decoders, and all the rest. They make a pretty strong case that their few cents per megaMAC compares quite favorably to hardwired DSPs.

What will be the next clever design trick to come out of the lab? As long as there's no shortage of signals to process, you can count on the silicon wizards to deliver something—CPU, DSP, or FPGA—to help get the job done.

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.

RESOURCE

Star*Core information,
www.starcore-dsp.com

SOURCES

ADSP-219*x*, '21065XL
Analog Devices, Inc.
(617) 329-4700
Fax: (617) 329-1241
www.analog.com

Goblin CompactPCI board
Bittware, Inc.
(603) 226-0404
Fax: (603) 226-6667
www.bittware.com

Web DSP
Precise Software Technologies, Inc.
(800) 265-9833
(613) 596-2251
Fax: (613) 596-6713
www.psti.com

MotionChip
Technosoft US, Inc.
Fax: (516) 287-9897
www.technosoft.ch

TMS320F241
Texas Instruments, Inc.
(800) 477-8924
(972) 995-2011
Fax: (972) 995-4360
www.ti.com

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

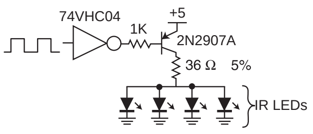
CIRCUIT CELLAR Test Your EQ

Problem 1—Why is this code nonportable?

```
#define MOTOR_SWITCH 0x10
int status_word;
status_word |= MOTOR_SWITCH;           /* motor on*/
status_word &= 0xffff ^ MOTOR_SWITCH;  /* motor off */
```

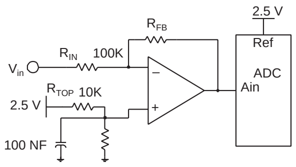
Problem 2—You're the senior engineer in your company's R&D group. Johnny, a junior engineer, was told to design an infrared emitter circuit to drive four separate IR LEDs. The emitter will be installed in a weather station, and the four IR emitters will provide a wider "data beam" than a single emitter, thereby relaxing alignment tolerance between the weather station and the receiver. The receiver is located inside a nearby instrumentation shack behind a window isolated from the temperature extremes experienced by the weather station.

Johnny designed the circuit shown here. Assume a nominal forward voltage drop of 2 V across the IR LEDs and an intended forward current of 20 mA (give or take a bit).



As the senior engineer, you're ultimately responsible for all designs that come out of your group, so you decide to reject Johnny's design. Why?

Problem 3—For this circuit, select standard 1% resistors for R1 and R2 to allow a 0–5-V input range. Assume an ideal op-amp, and a 0–2.5-V input range for the ADC. Hint: the amplifier is inverting, so full scale (5 V) at the input must map to 0 V at the ADC input, and 0 V at the amp input must map to 2.5 V at the ADC input.



Problem 4—You're responsible for sustaining an existing microprocessor-based controller. A customer calls to complain that your boards are failing with a high degree of regularity. The part failing most often is the RS-232 transceiver.

The application is fully self-contained, except for two RS-232 cables running into the box. You determine the cables are being subjected to 4–8-kV ESD events and replace the RS-232 transceivers with ±15-kV ESD-hardened transceivers from Maxim.

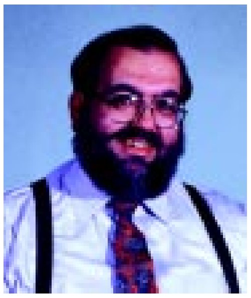
You use a commercial ESD generator and the human body model to discharge energy into the RS-232 lines of the test unit. Discharging ESD (up to ±15 kV) into one channel has no adverse effects on the system, but discharging 6 kV into the second channel consistently blows the CPU, RTC, and SRAM.

What's going on?

Have you forgotten the basics since getting that management job? Did your university ignore real circuits and tell you that HTML was the only path to fortune? Each month, Test Your EQ presents some basic engineering problems for you to test your engineering quotient. What's *your* EQ? The answers are posted at www.circuitcellar.com along with past quizzes and corrections. Questions and answers are provided by Ray Payne of Under Control, Benjamin Day of Schweitzer Engineering Laboratories, and Bob Perrin of Z-World.

PRIORITY INTERRUPT

All in the Family



The theme of the first issue of *Circuit Cellar INK* was “Inside the Box Still Counts.” The words sound pretty straightforward today, but at the time it meant a lot for me to say it on the cover of my own magazine. Back then, I was winding up my writing career at *BYTE*. I was also looking to set the record straight. After 11 years, I had been given the “opportunity” to continue Ciarcia’s Circuit Cellar if I would refocus my *BYTE* column from general engineering applications to IBM PC and advertiser hardware reviews (my response was something to the effect of standing in front of the editor-in-chief and pointing my finger down my throat).

My real response was this magazine. I believed that there were thousands of us out there who respected the appliance value of PCs but who were logical enough to know that somewhere down the food chain there still had to be people bright enough to design these appliances. Someone has to know what’s inside the box!

I only bring this up now because putting together an Internet publication has made me think a lot about our direction and the pitfalls of the past. *BYTE* showed us a concrete example of how to screw up a good thing by abandoning everything they were doing right and jumping on a fantasy vision for the future. Today, Yahoo, AOL, Amazon.com, and ChipCenter are successful examples of this kind of focused direction, but keep in mind that they were all new ventures without any history to abandon.

The problems arise in managing the gray area between tradition and expansion. We certainly have a successful history and I don’t want to compromise it while I seek to expand our goals. I have to select the things that we’ve always done successfully and enhance them, not replace them, with new technology. In other words, I’m trying not to do the *BYTE* thing.

I have to be honest and say that I don’t know what an online magazine is supposed to look like. I haven’t read a lot of them (I’m still a paper kind of guy) so it may take a while to settle on an agreeable look and feel. Ultimately, my primary objective is to expand editorial coverage to areas that had to be neglected because of the sheer economics of print space. New people and new projects are on the way.

Speaking of new projects, it’s that time of the year again: next month we will be announcing the *Circuit Cellar* Design2000 contest for the print magazine. Having *Circuit Cellar Online* enabled us to consider having an Internet-specific contest as well. Seems we weren’t the only ones who liked the idea, so the Microchip Internet PIC 2000 contest, conducted by *Circuit Cellar* and hosted by ChipCenter, will be officially announced in *Circuit Cellar Online* on September 1st. As the name implies, this design contest will focus on innovative uses of PIC processors connected to the Internet. Internet PIC 2000 is the perfect complement to the launch of *Circuit Cellar Online*. What better way to demonstrate the uniqueness of online embedded control?

I’m trying to be careful how I do all of this. Yes, *Circuit Cellar Online* has enough substance and uniqueness to stand as an independent magazine, but that’s not my intention. In the greater scheme of things, *Circuit Cellar* in print and *Circuit Cellar Online* are one magazine. Together they share the responsibility of providing the editorial excellence that educates a core group of technologists. And, I’m absolutely convinced of one thing: there are still plenty of you who live by “inside the box still counts.”

Steve

steve.ciarcia@circuitcellar.com