CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

#105 APRIL 1999

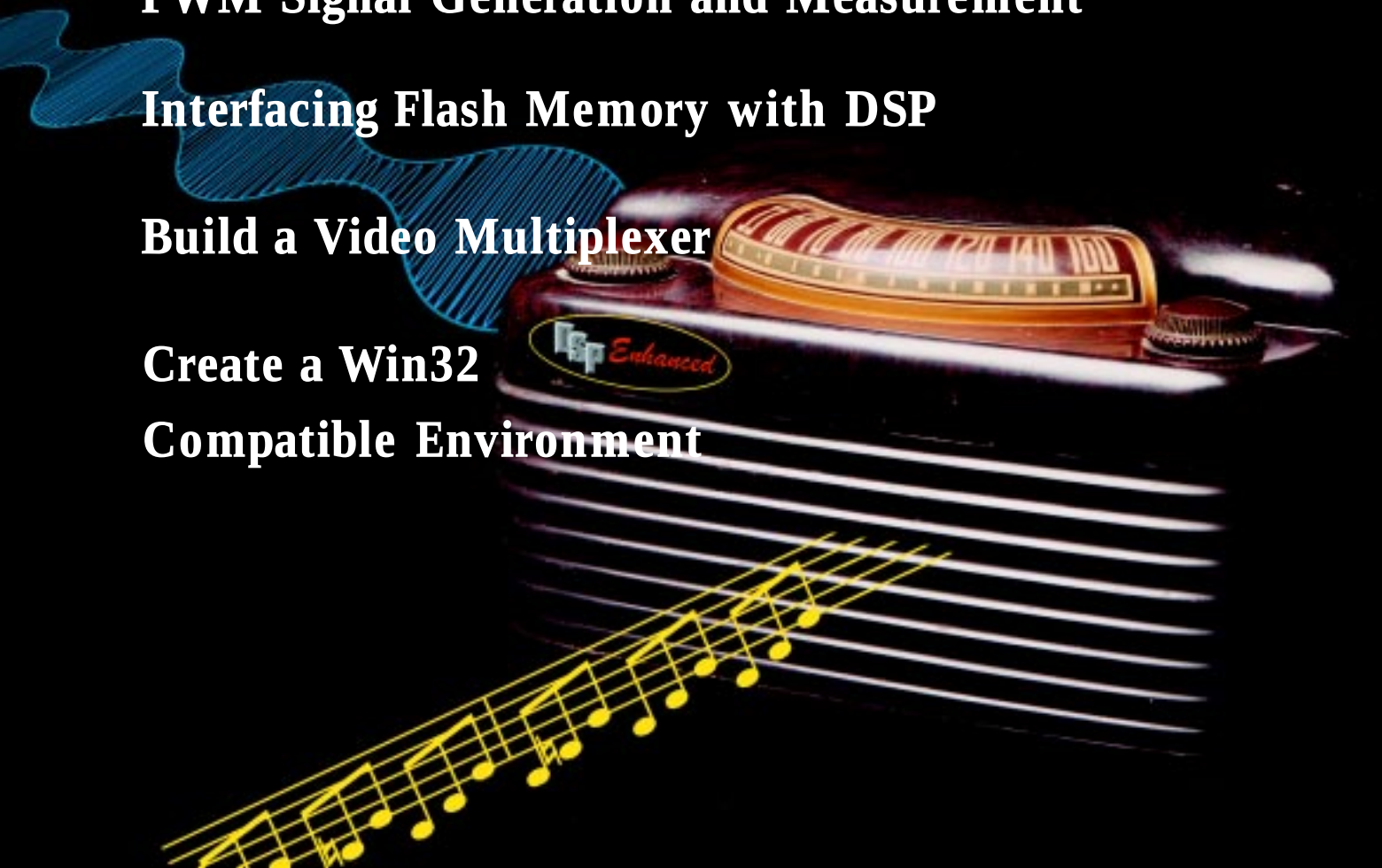
DIGITAL SIGNAL PROCESSING

PWM Signal Generation and Measurement

Interfacing Flash Memory with DSP

Build a Video Multiplexer

Create a Win32 Compatible Environment



TASK MANAGER

School's in Session

hh…the breeze is blowing gently, the air is brisk, the sky is blue—what a refeshing season! It's back-to-school time again!

Well, that's not true for my family here in Connecticut, but some of my friends are already thinking about pencils and notebooks. Did you know that the Japanese start their school year in April?

I remember the first time I heard that, just a few years ago. Seemed odd to me at the time, but maybe I've lived a sheltered life. There's always something new to learn, isn't there?

Truly, we're never so old, so wise, so mature that there isn't something we don't know. Or, what about when it's been so long since you heard about something that it's like learning about it for the first time? Have you ever experienced a familiar excitement when you go back to solving the problems you first tried to tackle years ago? Er…hang on a sec—do you even remember all those formulas and equations?

Perhaps you need to go back to school, too. Here's your chance, and from the comfort of your own chair! Starting this month, the Circuit Cellar academic year begins (and yes, we're in session 12 months a year). Our new quiz—Test Your EQ—lets you test your "engineering quotient."

We begin the monthly quiz in this issue with submissions from Bob Perrin, who I thank for approaching us with the initial concept. These questions are adapted from a much longer test given to all engineering applicants at Z-World. Give these problems the old college try and ask yourself, would you get the job?

As Steve mentioned in Priority Interrupt a couple months ago, our latest survey showed that readers were concerned that EEs don't always know even fundamental electronics. Test Your EQ lets you discover on an ongoing basis whether you make the grade. Maybe you've been involved on the management side of the team for such a long time now, you need a refresher course.

The answers to the problems in Test Your EQ will be posted on our web site (www. circuitcellar.com) each month, so you can check right away how you fared on the problem set. And although we will update the answers each month, past quizzes and their correct answers will be maintained on the site as well.

Of course, we also want you to take a turn on the "teacher side" of the desk. Submit your potential engineering quizzes to me via e-mail, fax, or regular mail, and for each half page of the magazine that we decide to fill with your questions, you will receive $50.

Wherever you happen to be today, it may look like spring or fall outside, but no matter. It's time to hit the books again. Turn to page 83 to go back to school now!

Eli

elizabeth.laurencot@circuitcellar.com

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

EDITORIAL DIRECTOR/PUBLISHER
Steve Ciarcia

MANAGING EDITOR
Elizabeth Laurençot

TECHNICAL EDITORS
Michael Palumbo
Rob Walker

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITORS
Ingo Cyliax
Ken Davidson
Fred Eady

NEW PRODUCTS EDITOR
Harv Weiner

ASSOCIATE PUBLISHER
Sue Skolnick

CIRCULATION MANAGER
Rose Mansella

CHIEF FINANCIAL OFFICER
Jeannette Ciarcia

ART DIRECTOR
KC Zienka

ENGINEERING STAFF
Jeff Bachiochi

PRODUCTION STAFF
Phil Champagne
John Gorsky
James Soussounis

PROJECT EDITOR
Janice Hughes

EDITORIAL ADVISORY BOARD
Ingo Cyliax Norman Jackson David Prutchi

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

ADVERTISING

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

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

CONTACTING CIRCUIT CELLAR INK

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

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

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

INSIDE ISSUE 105

READER I/O

ALTERNATE ROUTE

In "Digital Frequency Synthesis" (*INK* 99), Tom Napier used a Microchip PIC16C54 microcontroller for signal generation. Scenix Semiconductor makes parts that are improved versions of the PIC for applications such as this.

The SX18AC can replace the PIC and run 2.5× faster (50-MHz clock frequency instead of 20 MHz). If the SX is set for turbo mode, the internal divide by four is bypassed and the SX runs at the full 50-MHz clock rate.

Instructions such as GOTO, CALL, and returns take an extra cycle so the final speed is not quite 10× as fast. But, with a few changes to the program, there are other features that can make up the difference.

For instance, a fuse bit enables the arithmetic to use the carry flag in multiple precision calculations. This arrangement is faster and saves instructions by reclaiming some of the cycles lost by using turbo mode.

Using an SX series micro with a 24-bit accumulator gives about 1.9 MHz for the update rate, allowing a 600-kHz filtered output signal. A faster DAC and filter amplifier must be used to reproduce the higher frequencies. I used an R-2R DAC made with 120- and 240-W resistors.

In case you want to check it out, my Scenix version of the program is available at http://brouhaha.com/~eric/scenix/ in SXNCOEX.ZIP.

Richard Ottosen
ottosen@idcomm.com

WHICH NUMBERS COUNT

Steve, thanks for your efforts at *Circuit Cellar INK*. Please don't become big-advertiser driven. *INK* is the one magazine in which I look at every advertisement and read at least 90% of them. In my opinion, you have a good feel for what your readers want.

I know you've voiced your frustrations about advertisers wanting to direct the magazine. I think comments like the one from Eric Sells in Reader I/O (*INK* 98) tell the story for advertisers. What better testimonial than a company that's perceived as small and unknown becoming number two behind one of the giants?

Grant Powell
gapowell@gte.net

NEW PRODUCT NEWS

Edited by Harv Weiner

DSP-CONTROLLED SERVO DRIVES

UltraDrive's **G Series** servo drives use DSPs that are specifically designed for motor control. The drives provide 1/T velocity loop or torque-mode control, true performance matching with motor RMS horsepower modeling, and extremely low torque ripple. The G Series can be used with any servo motors (including linear brushless servomotors) incorporating either Hall-effect sensors (trapezoidal commutation) or Hall-effect sensors plus encoders (sinusoidal commutation).

The G Series offers PWM-switching frequencies from 10 to 20 kHz, and provides current loop bandwidths from 1 to 2 kHz. Four G Series models deliver from 5 to 20 A continuous current output and continuous output power from 1.2 to 4.8 kW. Integral power supplies can use universal AC input voltages from 90 to 265 VAC.

Software in the DSP's flash memory enables the user to define motor- and application-specific parameters. Parameter adjustments can be commanded on-the-fly to increase application-control flexibility. Windows-based software provides setup, tuning, and diagnostic utilities to reduce system setup time and commissioning. The software includes graphical plotting tools that turn a desktop or laptop computer into an advanced digital storage oscilloscope.

Flash memory provides an easy path for system firmware upgrades and eliminates batteries for parameter storage. Windows software and an upgrade disk are all that's required to update drives with firmware enhancements and customer-specific features for applications.

UltraDrive/Westamp
(818) 709-5000
Fax: (818) 709-8395
www.ultra-drive.com

8-/16-BIT MICROCONTROLLER

The **TSC80251G1D** enhances the C251 architecture core with serial communication interfaces. Running up to 24 MHz with an active power consumption as low as 24 mW (3-V version), the TSC80251G1D offers an excellent performance/power-consumption ratio. Applications include smart-card readers, cordless phones, ISDN PABx, networking, backplanes, and routers.

The TSC80251G1D features 16 KB of ROM, 1 KB of RAM, and several on-chip serial communication capabilities such as UART, serial peripheral interface, and multimaster I^2C. The device comes with three 16-bit timers, a keyboard interface, and complete programmable counter array with five modules including PWM, input capture, output compare, and timer/counters. It also includes an enhanced power-management unit with power-on reset, brown-out, and prescaler functions.

Available in commercial and industrial temperature ranges, the product is offered in 44-pin PLCC, 40-pin DIP, and space-saving 44-pin Thin QFP packages. The device operates from 2.7 to 5.5 V.

Pricing for a 44-pin PLCC in commercial temperature range starts at **$4.40** in quantities of 10,000.

Temic Semiconductors
www.temic-semi.de/nt/micro

NEW PRODUCT NEWS

PC REMOTE CONTROL

The **KeyRF PC** remote control enables you to receive signals from a keychain remote control transmitter (like a garage door opener) on your PC. It can be used for overhead-projection presentations, to flip channels on your built-in PC TV, or for home automation systems. The RF technology is superior to infrared units, allowing 360° transmissions over longer distances and through walls and objects. The receiver can be programmed to assign custom keyboard keys to all five transmitter-button combinations. The KeyRF plugs between the keyboard and the PC, and it does not require additional software to be installed on the PC.

The KeyRF kit comes with a two-button keychain remote control transmitter, a radio-frequency receiver, a cable to connect to the PC keyboard port, and a user manual.

List price for the KeyRF kit is **$150**.

L3 Systems, Inc.
(425) 836-5438
Fax: (425) 868-8706
www.L3sys.com

NEW PRODUCT NEWS

SINGLE-BOARD COMPUTER

The **BasicBox** is an SBC system designed for education, experimentation, and embedded applications. The design is adopted from Jan Axelson's *Microcontroller Idea Book*.

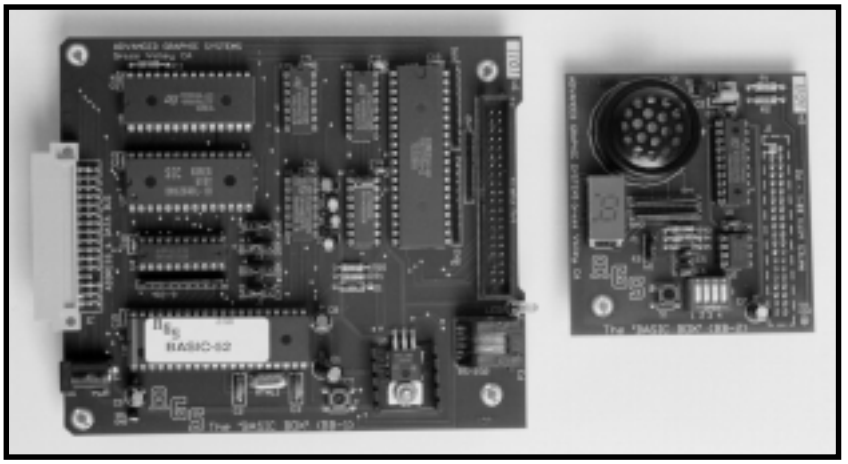
The standard board configuration contains an 8052 microprocessor with built-in BASIC-52 interpreter, 8 KB of RAM, 8 KB of EEPROM, a UART for RS-232 serial communications, and 35 lines of parallel I/O. The board is programmable in BASIC-52 or 8051 assembly language with either a PC or a Macintosh via a modular telephone type cable. The BasicBox is capable of expanded I/O via its DIN connector.

The BasicBox (BB-1) is packaged with a copy of Axelson's book, terminal software, power supply, serial cable (PC or Macintosh), demo programs, documentation, and the "Board of Education" (BB-2)—an experimentation board that gives the user an interactive learning environment.

The BB-2 contains a variety of devices designed to give feedback as the user becomes more familiar with the BasicBox. The devices include a speaker that gives audible tones when exercising the PWM, a seven-segment LED that can be manipulated both bit- and byte-wise, a pushbutton for experimenting with debounce algorithms, a DIP switch for discrete inputs, and an oscillator for measuring frequency and bit times.

The BasicBox is priced at **$249** plus shipping for single quantities.

Advanced Graphics Systems
(530) 887-1619
Fax: (530) 887-0107
www.ags-gv.com



GPS-BASED NETWORK TIME SERVER

The **NTS-90** network time server from TrueTime distributes time to precisely synchronize client computer clocks over a network. Using the GPS as its primary source, NTS-90 transfers the time over the network to client computers using the well-established network time protocol (NTP). Synchronization accuracy over the network is typically 1–10 ms.

With near plug-and-play operation, installation is easy and ongoing maintenance and support costs are very low. Client computers can continually be added to the network and directed to retrieve time from the NTS-90. An RS-232 command set provides versatile control and a single command configures the unit for immediate use. Other commands give status information, precision timing, and total control.

The NTS-90 is also a source for accurate serial time broadcasts and time on demand. A once-per-second time broadcast (accurate to the millisecond) is available via the serial port. The NTS-90 can also timestamp external events with millisecond accuracy.

The NTS-90 comes in a rack-mountable configuration. The rear panel supports the synchronization input connector (GPS or ACTS), a 15-pin AUI network connector, and a 9-pin RS-232 communications/initialization connector.

TrueTime, Inc.
(707) 528-1230
Fax: (707) 527-6640
www.truetime.com

FEATURES

FEATURE ARTICLE

Ethan Bordeaux & Stefan Hacker

Integrating Flash Memory in an Embedded System

Turn on to the power and ease of flash! Ethan and Stefan show us the best uses for flash memory, and create an interface between a DSP and flash. It's easy to tie into your system, and you can't beat having in-system programmability.

Until recently, the most flexible external boot memory was an EPROM. But, if you needed to erase or update the data, you had to remove the EPROM from the system, expose it to ultraviolet light, place it in an EPROM burner, and insert it back into the design.

A better solution is to update code and data while the nonvolatile memory is in the system. This is the type of functionality built into flash memory.

We're sure you can imagine lots of applications and system configurations where such functionality is useful.

One application is an embedded speaker-independent voice-recognition unit, as in a hands-free car kit or voice-activated appliance. The user programs a number of keywords to perform operations like dialing a phone number or turning on and off a device. The processor would need to store these voice patterns in external nonvolatile memory and be able to retrieve them for comparison purposes.

Flash memories are also an asset in systems that need to save data during a power outage or brownout. The processor moves its code and data contents from volatile internal memory to an external nonvolatile memory and, on revival of the system, continues at the last saved state.

In this article, we explain some of the benefits and basic functionality of flash memories. We cover an example interface between an ADSP-218*x* DSP and an Am29F040 flash memory as well as hardware and software structure.

FLASH VS. EPROM

Even with the in-system programmability (ISP) of flash memory, there are times when a conventional EPROM may be a better choice for your design. Table 1 is a partial list of considerations for choosing a byte memory.

EPROMs cost less than flash memory with similar storage capabilities, but there are many reasons to consider using flash memory. ISP is an obvious advantage. They can potentially offer all of the functionality of an EPROM and an SRAM in a single package.

Because flash memory is such a hot commodity in today's semiconductor market, many manufacturers are focusing research and development, along with their advanced manufacturing processes, on the flash market. This translates into flash memories with low power consumption during operation and powerdown, more aggressive operational voltages, and faster access times (e.g., Intel's Strato flash family).

So, if your design requires the absolute lowest power consumption, a flash memory may be the best nonvolatile solution. And because operational voltages on flashes have now reached 1.8 V, they can gluelessly interface to systems that operate below LVTTL levels.

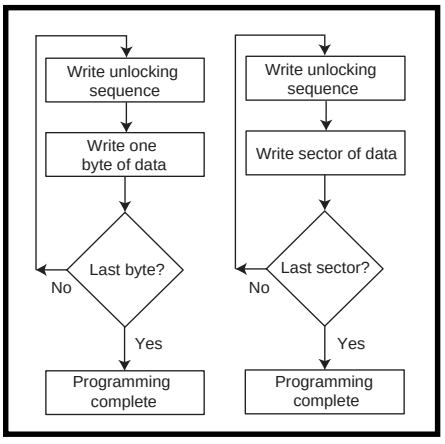


Figure 1—*A tale of two programming models: the left-hand side (Am29F040B) requires an unlocking sequence after every byte is programmed, and the right-hand side (AT29C040A) requires an entire sector of memory to be programmed each time you write to the flash.*

	EPROM	Flash memory
Price per megabit	$1–3	$2–10, depending on features
Typical packages	PDIP, PLCC, BGA	BGA, PDIP, PLCC, TSOP
Typical operational voltages	2.5 V, 3.3 V, 5 V	1.8 V, 2.5 V, 3.3 V, 5 V
Power consumption	30–200 mW	15–50 mW (read), 45–200 mW (write/erase)
Typical access time	60–200 ns	18–150 ns

Table 1—*Even though flash memories are typically more expensive than EPROMs, a wide range of voltages, fast access times, low power consumption, and inherent ISP can make them powerful and flexible memory ICs.*

FLASH ROBUSTNESS

Flash memories are given a rating for the number of write cycles they can sustain before the part is not guaranteed to operate properly. Flash memories from a few years ago sustained ~10,000 write cycles. Assuming your design is supposed to last 10 years, this translates to ~2.7 memory rewrites a day.

Some systems are deterministic enough to guarantee this rate, but often it's unknown exactly how many times the flash memory will be programmed or erased in its lifetime. Some flash memories now guarantee upward of 1,000,000 write cycles (or 270 rewrites a day) before failure. This alleviates some of the flash programming concerns and enables them to fit into a wider variety of designs.

INTERNAL VS. EXTERNAL

There's been a lot of talk recently about embedding flash memory on processors, microcontrollers, or DSPs and the increased level of system flexibility that would provide.

The strongest argument for including flash memory on a processor is for prototyping. If a manufacturer provides a processor with both a ROM and flash-memory variant, the user can use the flash device for prototyping and the ROM-coded processor for production.

But, processors with embedded flash memory cost 2–10´ more than their flash-less counterparts and are aimed at prototyping environments. And, flash memories frequently can't operate as fast as onboard SRAM or ROM. The entire processor speed can be compromised by including flash memory.

Lastly, the size of a flash memory embedded with a processor is typically 10–30k words. Although this is often adequate for a portion of program or data memory, many times, a design needs access to a much larger memory space, where it can grab many code and data overlays or use a nonvolatile memory source as a virtual hard drive.

Applications like digital cameras require many megabytes of nonvolatile storage for saving each "roll" of digital film. Algorithms such as voice recognition need large external memories for voice look-up tables. To store 30 words for a speaker-independent voice-recognition system, up to 512 KB of external memory is needed.

These applications can't be supported by processors with on-chip flash memory. These systems require a simple and flexible method of gluelessly connecting an external flash for the optimal solution.

EMBEDDED ALGORITHMS

The simplest way to understand a flash memory is to think of an SRAM programmed via a finite state machine (FSM). When reading information from the flash memory, you have normal access to all memory locations, much like an SRAM or EPROM.

But, when specific operations need to be performed on the flash, whether it's erasing, writing information, or protecting memory segments from erroneous erasure, the processor must use the flash's embedded programming algorithms (EPAs).

A series of commands are written from the processor to the flash. These commands unlock the flash so it can accept data, erase sectors, or perform other programming tasks. We explain a few of the common functions here.

BYTE/SECTOR WRITE

Byte/sector write lets the host processor place data into flash memory. The processor first writes the unlock sequence to the flash and then writes the address and data for the first value to be programmed.

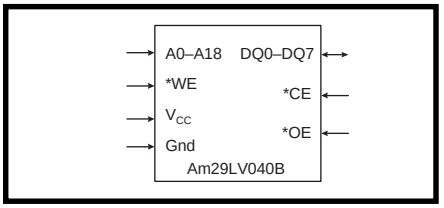


Figure 2—*The Am29LV040B has memory strobes similar to an EPROM, with the exception of a write pin (WE).*

Depending on the programming methodology implemented on the flash, additional unlocking commands need to be written for each word, or a sector (typically between 64 bytes and 64 KB) of memory can be written in a burst fashion, as you see in Figure 1.

In both types of flash memories, the processor writes the unlocking sequences and data to be programmed into the flash and the flash latches the data into memory. This sequence enables the processor to continue executing algorithmic data while the flash memory handles programming itself.

Keep in mind that the contents of the block you want to program must be erased beforehand. A convenient feature of sector-programmed flash memories is that they usually erase the sector before programming it with data.

One methodology is not inherently better than another, but each is better suited for certain systems. Consider these points before choosing a flash-writing architecture.

First, decide what type of data you're writing. Will your data be partitioned as a large chunk of information (e.g., a JPEG file for a digital camera) or will the external memory save single bytes of information?

If you're only making small incremental changes to the flash memory, it might make sense to choose a flash with a byte-programming protocol. But, if you write large pieces of data from the processor, a sector-programmed flash may be more efficient.

Second, what kind of processor is writing to the flash? Low overhead DMA and fast byte-port accesses are available in some embedded processors.

For example, the ADSP-218*x* DSPs integrate a byte-wide DMA port that supports a variety of ICs, including flash memories. Knowing how your processor will interface with a flash and its EPAs is useful.

SECTOR/CHIP ERASE

Flash memories are partitioned into a number of sectors, which enables the programmer to erase one sector of the memory at a time.

In general, a series of commands is written out to the flash to start the erasing procedure. Even though the commands are latched into the flash in a few microseconds, actually erasing the flash takes a couple seconds.

Some flash memories don't allow the processor access to the flash during a sector erase. Others, however, have simultaneous read/write architectures that permit memory accesses to one block while the other block is erased.

AUTOSELECT/PRODUCT ID

This EPA determines specific information about the flash in your system. The information includes the manufacturer ID (necessary because many companies make pin-for-pin–compatible flash devices), device ID, and sector-protection feedback. This data is useful for external flash programmers/burners.

SECTOR PROTECT

The sector-protect EPA disables both the programming and erasing of a particular memory sector. This feature is useful in systems where you both boot and continually write/erase the flash. You can set the boot sector to be protected and leave other memory segments unprotected for a chip erase.

DATA POLLING

The time it takes a specific EPA to finish varies greatly, even on the same device. For example, the Am29F040B flash memory takes from 7 to 300 µs to program a byte of data, and sector erase time can vary from 1 to 8 s. There are provisions for the flash memory to signal when an operation completes.

Data polling is a common method for determining when an EPA is finished. Once the processor writes the data into the flash, it is latched inside the part and the processor isn't needed to control the actual writing of data into memory. But, the flash still needs to place the new information into its memory bank and signal when it's ready to handle another EPA.

One method is to poll the status of one of the data bits to see if the EPA has completed. For example, the Am29F040 provides data-polling capability on data pins DQ7 and DQ6.

By reading back DQ7's value during a byte write, you can determine if the programming is done. If DQ7 has the inverted value of the programmed value (e.g., the byte value programmed into the flash is 0x0F [DQ7 = 0] and DQ7 is read back as a 1), the EPA is not complete. If DQ7 reads back as the value programmed on that bit, the EPA is done and the next byte can be programmed.

DQ6 can be used in a similar way. While the flash is still in a byte-programming EPA, successive reads of

Listing 1—*This code shows you the function* `PROG_BYTE`.

```
* Flash application server
* int error PROG_BYTE(char c_byte, char d_byte, int addr_lo, int
*    addr_hi);
* Byte program:
*   c_byte : 0xA0              // Third input to AMD EPA
*   d_byte : value to program  // holds data to be programmed to flash
*   addr_lo: low 16bit         // lower part of 22-bit address
*   addr_hi: high 6bit         // higher part of 22-bit address
* Register usage summary:
*   modify : addr_lt, addr_ht, d_btmp
*   destroy: ar, ax0, ay0, af
*   calls : init_seq, cmd_write, calc_adr, DQ7_poll

prog_byte:
  call init_seq;               // EPA unlock sequence
  call cmd_write;              // EPA command word write
  ar = dm(d_byte);             // fetch byte from input register
  dm(d_btmp) = ar;             // store byte in destination register
  call calc_adr;               // compute registers from address
  call DQ7_poll;               // check for internal completion
  rts;                         // return from function call
```

DQ6 cause the value to toggle. When the EPA finishes, DQ6's value stops toggling and reads as the same data value. The data-polling scheme you choose depends on your processor and whichever method is easier to implement in software.

Function	Purpose
BDMA_SETUP	Initializes the byte-wide memory port for flash transfers
PROG_BYTE	Write one byte of information to flash
READ_BYTE	Read one byte of information
SECT_ERASE	Erase one sector of information
MEM_IDENT	Identify the manufacturer
AUTO_INC	Automatic incrementing of address for multiple-burst transfers

Table 2—*In our software, we adopted a function-calling scheme that hides much of the underlying protocols and architecture of the flash memory.*

READY/*BUSY PIN

A lower-overhead method of determining EPA completion is via a READY/*BUSY pin. Some newer flash memories include this pin to signal the status of the flash at any moment. Essentially, the pin is at a logic low when the flash is in any EPA and at a logic high when it's ready to read data or in standby.

This method is useful on processors where the status of external flag pins can be easily tested. ADSP-218*x* DSPs provide a variety of I/O pins and support for externally generated interrupts, which can be connected to test the status of READY/*BUSY.

The tradeoff is that this method ties up an additional pin and increases the total number of signals in your design.

FLASH TO DSP

Now that you're acquainted with some typical flash EPAs, here's an example using an ADSP-2184L DSP and Am29LV040B flash memory.

The Am29LV040B (see Figure 2) has a basic set of memory strobes. Other flashes may provide READY/*BUSY strobes, hardware pins for locking the contents of the memory, or methods of reading information for synchronous burst transfers. We chose this memory primarily for its simple hardware and software interfaces.

The ADSP-2184L is a 16-bit fixed-point DSP. This processor family contains a number of external interfaces including an external byte-wide DMA port that can be configured to support 8-bit memories, including flash. Figure 3 shows the pins that the DSP uses to connect to an external memory.

There are 14 address and 24 data lines available externally on the DSP. This configuration enables direct external-program memory execution because its opcodes are 24 bits wide. But, when the DSP accesses external byte space, only D8–D15 are connected to the flash memory data bus, allowing D16–D23 to become address bits and creating a total address reach of 4 MB.

The DSP can supply the necessary chip-, read-, and write-select strobes to the flash. Because the ADSP-2184L operates at speeds up to 40 MHz and flash memory may be just 5–6 MHz, the on-chip wait-state generation logic enables a flash interface without external glue logic.

The ADSP-2184L does not operate out of external byte-wide memory. That memory is typically too slow to support the processor's operational speeds. Instead, the DSP transfers the information from the byte-wide memory into internal SRAM and operates from this memory space.

There are hardware provisions for packing data and program-memory words (16 and 24 bits) into internal memory. Byte memory space is intended to be a large region of memory where the processor can fetch instructions or data to be operated on or to save externally to the chip for later use. Software development tools let code and data be saved as memory overlays, which can be loaded into the DSP under control of an overlay manager.

FLASH SOFTWARE

The code modules of our ADSP-2187L flash software enable the programmer to access flash memory via function call APIs. Table 2 lists the base functions, along with their purpose.

Each function expects input data to be located in specific registers and output data to be placed into specific registers. Once you understand the rules, it's simple to tie the function calls into your software. An example of the assembly code used to program one byte of memory on the ADSP-2184L is in Listing 1.

The code shows the first layer of the software interface between the flash and DSP. Because many of the basic building blocks for each of the EPAs are similar, an additional set of functions (`INIT_SEQ`, `CMD_WRITE`, `CALC_ADR`, and `DQ7_POLL`) were written for `PROG_BYTE` to call.

This function assumes that the flash software housekeeping function (`BDMA_SETUP`), which configures the DSP for byte transfers, has already been called.

There are several steps to the basic program flow for `PROG_BYTE`. First, place the appropriate values into the addresses pointed to by `c_byte`, `d_byte`, `addr_lo`, and `addr_hi`.

Be sure that the values contained in the ar, ax0, ay0, and af registers are no longer needed. If they are needed after `PROG_BYTE` completes, the background register set on the ADSP-2184L can be used for programming operations and then switched back when the function ends.

`PROG_BYTE` calls two functions, `INIT_SEQ` and `CMD_WRITE`, which write the first three bytes of information into the Am29F040B's FSM to unlock the memory.

The third step is a memory transfer between the input data location and the API. This transfer is necessary to save the value of the data word for use later when polling the data.

`CALC_ADR` creates the address where the data is stored in the flash. It performs the last byte transfer to the flash.

Lastly, `DQ7_POLL` (which continually tests the status of the DQ7 bit) is called to determine when the flash has transferred the data into memory and when it's free to enter a new EPA.

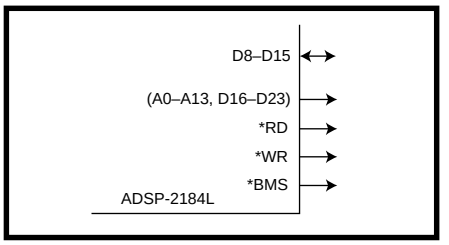


Figure 3—*The ADSP-2184L provides all the memory strobes required to control the flash memory.*

NOW IT'S YOUR TURN

This flash interface is easy to tie into your application code. With a set of reference functions for interfacing these devices, you can add new functions as your system needs change and flash technology develops.

Ethan Bordeaux works for Analog Devices as a 16-bit DSP product line applications engineer. He has worked with embedded speech processing applications including speaker identification, speech recognition, and adaptive echo cancellation systems. You may reach him at ethan.bordeaux@analog.com.

Stefan Hacker is a DSP applications engineer at the European DSP support center for Analog Devices. His focus is on OEM accounts using 16- and 32-bit DSP products in consumer and industrial applications. You may reach him at stefan.hacker@analog.com.

SOFTWARE

The functions described here and instructions for linking them into your design are available via the Circuit Cellar web site.

REFERENCES

Advanced Micro Devices, *Am29-F040B*, Datasheet, 1998.

Analog Devices, *ADSP-2100 Family Users Manual*, 1995.

Atmel, *AT29C040A Datasheet*, 1998.

C. Leidigh, "Flash Memory Buyer's Guide," *Communications Systems Design*, March 1998.

SOURCES

Am29F040B, Am29LV040B
Advanced Micro Devices, Inc.
(408) 732-2400
Fax: (408) 732-7216
www.amd.com

ADSP-218*x*
Analog Devices
(781) 329-4700
Fax: (781) 329-1241
www.analog.com/dsp

AT29C040A
Atmel Corp.
(408) 441-0311
Fax: (408) 436-4200
www.atmel.com

FEATURE ARTICLE

Andrew Lillie

Embedded PWM Signals

Andrew shows that embedded microcontrollers are quite capable signal generators and measurement tools. The modular design of the MPC555 enables the mixing and matching of submodules, so you can get the I/O you need.

Applications have grown in complexity over recent years, and microprocessor suppliers have responded with more sophisticated embedded controllers. Today's microcontrollers have on-chip peripherals that control everything from automobiles to electronic xylophones.

One microcontroller application is PWM, a process used to control devices such as stepper motors, communication systems, battery management, audio applications, and thermal write/erase heads. The auto industry uses PWM to generate control signals.

The MPC555 embedded processor was designed for automotive applications like electronic throttles, regulator valves, and position motors. It carries a modular I/O system (MIOS) that is capable of eight PWM channels and 10 period- and pulse-measurement channels. MIOS1 is the implementation of the MIOS architecture used in the MPC555.

In this article, I outline the initialization of the MIOS module and explain the clock management and programming necessary to generate a 25% duty cycle, 2.5-MHz waveform. I also give you programming examples in assembly and embedded C. Let's look at how the MPC555 submodules are integrated and how they communicate with each other and the PowerPC core.

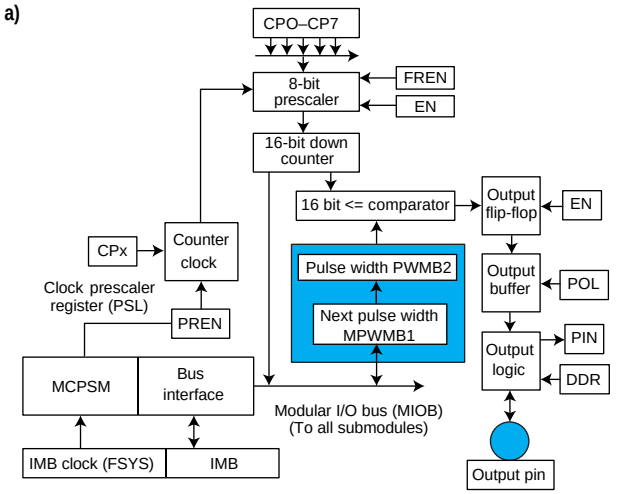
SIGNAL GENERATION

The MIOS consists of a library of flexible I/O and timer functions including counters, input capture, output compare, pulse and period measurement, and PWM. It's easily configured for different kinds of applications because it is composed of submodules.

The PWM can be initialized and programmed by setting up the appropriate registers. The eight PWM channels are controlled by one system register and three registers that are unique to each channel.

Bit	Name	Function	Description
0	PREN	Prescaler enable	This active high read/write control bit enables the MCPSM counter. PREN is cleared on reset: 0 = MCPSM counter disabled 1 = MCPSM counter enabled
1	FREN	Freeze enable	When set, this active high read/write control bit makes it possible to freeze the MCPSM counter if the MIOB freeze line is activated: 0 = MCPSM counter not frozen 1 = Selectively stops MIOS1 operation when the FREEZE signal appears on the IMB3
2–11	—	Reserved	
12–15	PSL	Clock prescaler	This 4-bit read/write data register stores the modulus value for loading into the clock prescaler. The new value is loaded into the counter the next time the counter equals one or when disabled (PREN bit = 0). Divide ratios are: 0000 = 16 0001 = No counter clock output 0010 = 2 … 1111 = 15

Table 1—*Here are the bit settings for the MCPSM, address 0x306816. This register controls the MIOS counter prescaler submodule. The MIOS counter is the common clock between the eight PWM channels on the MPC555.*



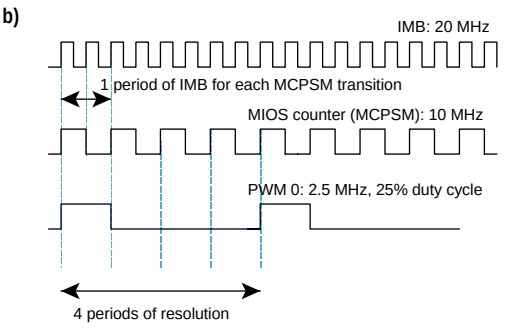


Figure 1a—*In the MPC555 MIOS submodule integration, the CP0–CP7 bits are the prescaler bits 8–15 in Table 2 and are unique to each PWM channel.* ***b—****This timing diagram shows resolution dependencies and MCPSM generation. Note the four counts of the MIOS counter per period of the PWM channel.*

STARTING THE MIOS COUNTER

To use the PWM channels, start the MIOS counter, which is slaved to the intermodule bus (IMB). The IMB runs at half the system clock, so let's assume it's operating at 20 MHz. Figure 1 explains the MIOS submodule integration.

The MIOS counter prescaler submodule (MCPSM) divides the system IMB clock to generate the counter clock. It also synchronizes all the submodules with the same division of the system clock.

The clock signal is prescaled by loading the value of the clock prescaler register into the prescaler counter every time it overflows. This arrangement permits all prescaling factors between 2 and 16. The MCPSM is controlled by the MCPSM status/control register. Counting is enabled by asserting the PREN bit in the control register (see Figure 1a and Table 1).

Because the PWM uses one count of the MIOS counter per transition of the PWM signal, the maximum frequency that the MIOS modulators can generate is half the MIOS counter clock frequency. Later you'll see that this affects the resolution of the generated waveform. Figure 1b shows the interaction of the clock signals between MPC555 modules.

Following the register description in Table 1, set the first bit high to enable the counter and write 0010 to bits 12–15 to set the divider to 2 for the fastest counter possible. A faster counter results in faster waveforms and more accurate measurements. All other bits should be 0.

In hexadecimal, write 0x8002 to register 0x306816 (MCPSMCR). If you program this register using a background debug mode tool, enter something like `write -w 0x306816 = 0x8002`. `-w` indicates that a 16-bit word is to be written.

To automate the programming process further or include the PWM in an application, you can write a header file in C similar to Listing 1. In the application code, the command for bit-wise programming is `mcpsmscr.fren = 0;`. To write the whole register at once, use `MCPSMSCR = 0x8002;`.

Programming in C makes it easy to identify the bits and their functions. But for real-time debugging and testing, assembly is faster because the code doesn't need to be compiled and loaded each time.

Bit	Name	Function	Description
0	PIN	Pin input status	PIN bit reflects the state present on MPWMSM (MIOS PWM (Submodule) pin. Software can thus monitor the signal on pin. The PIN bit is read-only. Writing to it has no effect.
1	DDR	Data direction register	When the PWM function is used, the DDR bit has no effect.
2	FREN	Freeze enable	This active high read/write control bit enables the MPWMSM to recognize the freeze signal on the MIOB: 0 = MPWMSM not frozen even if MIOB freeze line is active. 1 = MPWMSM frozen if MIOB freeze line is active.
3	TRSP	Transparent mode	TRSP bit indicates that the MPWMSM double buffers are transparent: when the software writes to either the MPWMA or MPWMB1 register, the value written is immediately transferred to respectively counter or register MPWMB2: 0 = Transparent mode deactivated 1 = Transparent mode activated
4	POL	Output polarity control	POL bit works in conjunction with the EN bit and controls whether MPWMSM drives the pin with the true or the inverted value of the output flip-flop.
5	EN	Enable PWM signal generation	The EN bit defines whether MPWMSM generates a PWM signal or is used as an I/O channel: 0= PWM generation disabled (pin can be used as I/O) 1= PWM generation enabled (pin is output only)
6–7	—	Reserved	
8–15	CP	Clock Prescaler	This 8-bit read/write register stores the two's complement of desired modulus value for loading into the built-in 8-bit clock prescaler. The value loaded defines the divide ratio for the signal that clocks the MPWMSM period counter. Table 3 gives the clock divide ratio according to the CP values; also see Figure 1.

Table 2—*Here are the bit descriptions for the PWM status/control register (0x306006). Each of the eight PWM channels has a PWM status/control register, allowing independent programming. Bits 8–15 determine the clock divider for each PWM channel.*

Prescaler Value (CP in hex)	MIOS prescaler clock divider
FF	1
FE	2
FD	3
FC	4
FB	5
…	…
02	254 (2^8–2)
01	255 (2^8–1)
00	256 (2^8)

Table 3—*These clock prescaler bits determine the prescaler used to divide the MIOS counter clock for each PWM channel.*

CHANNEL PROGRAMMING

The next step is to program the individual PWM channels. Each channel is controlled by the period, pulse, and status/control registers.

The PWM period register sets the number of divisions per period of the waveform (i.e., the resolution of a single period). Choosing the resolution of the wave requires special consideration because it involves a compromise.

The resolution of the wave is inversely proportional to the maximum frequency that can be produced. A higher resolution requires more cycles of the MIOS counter per period and therefore a lower overall frequency.

The period register contains the binary value corresponding to the number of MIOS clocks allocated to the period of the waveform. The lowest possible resolution is two bits, with one clock each for high and low transitions.

Why would you want a higher resolution? Higher resolutions allow finer control over the signal's duty cycle. With 2-bit resolution, you're limited to a 50% duty cycle. With 16-bit resolution, you have almost 65,000 possible duty cycles. Continuing this example, you can program the PWM channel with a period resolution of four with `write -w 0x306000 = 0x0004`.

The PWM pulse register sets the number of divisions of the period register that are high. That means the ratio of the pulse register to the period register determines the signal's duty cycle.

The pulse-register value must be less than the value contained in the period register. In this example, enter `write -w 0x306002 = 0x0001` for a 25% duty cycle on channel 0.

The third register is the PWM status/control register. The last eight bits of this register set the clock divider for the particular PWM channel. This clock divider operates on the MIOS counter submodule programmed earlier.

Each PWM channel uses a divider to slow the MIOS counter clock, but remember that this also affects the period and pulse-width registers. Table 2 describes the bit assignments for the PWM status/control register. Bits 8–15 determine the clock prescaler, whose values are shown in Table 3.

PUTTING IT TOGETHER

With the 20-MHz IMB divided in half for a 10-MHz counter, you need to program the PWM channel to divide the MIOS counter by 1. Now there's room to split the period into four parts and allow a 25% duty cycle at 2.5 MHz. Here, use `write -w 0x306006 = 0x54ff` for PWM channel 0.

SUBMODULE CONFIGURATION

The MPC555 includes a MIOS dual-action submodule (MDASM) that makes pulse-width and period measurements. It can also be used to capture waveforms and generate single and continuous pulses. Let's use the MDASM to measure the period of the waveform that was generated earlier using the PWM.

To measure the output of the PWM with the MDASM, you need to get the signal from the PWM channel (here, channel 0) to the first MDASM channel. You can do this with a jumper wire and the pins available on the eval board.

The chosen MDASM channel must be configured to measure the period of the signal. You can calculate the period of the waveform by reading (from the register) the number of counts that occur during the period of the signal.

SETTING UP THE COUNTER

The MPC555 MIOS has a modulus counter submodule (MMCSM) that works as a free-running counter to which events can be referenced when they are detected. It can also be used for complex counting and timing functions.

The MDASM and the MMCSM work together to measure inputted waveforms. For this example, set the counter to be free-running and to automatically roll over when it reaches its maximum value. The counter is con-

Listing 1—*This C header file for the MIOS counter prescaler module provides two methods to address the register: bit-wise in lowercase or word-wise in upper case.*

```
#ifndef _MIOS_H
#define _MIOS_H
//MIOS Counter Prescaler Submodule Status/Control Register(MCPMSCR)//

typedef struct {
  unsigned  pren:             1;
  unsigned  fren:             1;
  unsigned  reserved_bit2_11  10;
  unsigned  psl:              4;
} Mcpsmscr;
#define mcpsmscr (*(Mcpsmscr *)0x306816)
#define MCPSMSCR (*(volatile unsigned short *)0x306816)
#endif
```

Listing 2—*The MDASM module outputs the value of the MMCSM counter in a long word when the `read -l` command is used with register 0x306058. There are four counts of the 10-MHz clock between `ABCD` and `ABC9`.*

```
> read -l 0x306058
  (0x306058)
  00306058   ABCDABC9   00028002
      …          …          …
```

trolled by the MMCSM status/control register described in Table 4. The MDASM can be referenced directly to an external clock if one is available.

I set up the MDASM to count on rising edges (register bits 3–4) and to use the MMCSM clock. The clock prescaler is programmed by the same values as the PWM prescaler bits listed in Table 3, and I set it up to follow the MMCSM clock with a prescaler division of 1. Therefore, the counter will run at 10 MHz, or 100 ns per count.

According to Table 4, you enter 0x0eff into register 0x306036 to set up the MDASM counter by using `write -w 0x306036 = 0x0eff`.

To ensure that the counter starts properly, reset it by loading all zeros into the modulus latch register. The modulus latch register is a read/write register that contains the 16-bit value of the counter used by the MDASM. `write -w 0x306032 = 0x0000` resets the counter. With the clock running, you can set the MDASM to detect the waveform periods and reference them to the MDASM counter.

Bit	Name	Function	Description
0	PINC	Clock input pin status	This read-only status bit reflects the logic state of the clock input pin.
1	PINL	Modulus load input pin status	This read-only status bit reflects the logic state of the modulus load pin.
2	FREN	Freeze enable	This active high read/write control bit enables the MMCSM to recognize the MIOB freeze signal.
3, 4	EDGN, EDGP	Modulus load falling/rising edge sensitvity	These active high read/write control bits set falling/rising edge sensitivity, respectively. 00 = Disabled 01 = MMCSMCNT load on rising edges 10 = MMCSMCNT load on falling edges 11 = MMCSMCNT load on rising and falling edges
5, 6	CLS	Clock select	These read/write control bits select the clock source for the modulus counter. 00 = Disabled 01 = Falling edge of pin 10 = Rising edge of pin 11 = MMCSM clock prescaler
7	—	0	
8–15	CP	Clock prescaler	This 8-bit read/write data register stores the two's complement of the desired modulus value for loading into the built-in 8-bit clock prescaler. The new value is loaded into the prescaler counter when the next counter overflow occurs or the CLS bits are set to select the clock prescaler as the clock source. Table 3 gives the clock divide ratio according to the CP values.

Table 4—*The MMCSM status/control register (0x306036) controls the counter used by the MDASM to reference transitions on the inputted waveform.*

CHANNEL CONFIGURATION

Next, initialize the MDASM to perform input period measurement. Like the PWMs, the MDASM channels have their own configuration registers. Each is programmed and read using the data A, data B, and status/control registers.

The data A register contains a value for the counter when the last event occurred. The data B register contains the previous value of data A or an independent measurement.

The status/control register has a read-only bit reflecting the status of the MDASM pin as well as read/write bits related to its control and configuration.

The MDASM status/control register (MDASMSCR, address 0x30605E) initializes the first MDASM channel. Table 5 defines the bits in this register.

Bit 0 is a read-only status pin that toggles according to the status of the incoming waveform. For input period measurement, let's write 0 to the unused bits (1, 3, 5, 6, 7, 8, and 11).

Because you don't want the MDASM to freeze in background debug mode, bit 2 is left as 0. Bit 4 is set to 0 to trigger the MDASM counter on the rising edge. Figure 2 shows the MDASM submodule for input period measurement.

Bit	Name	Function	Description
0	PIN	Pin input status	Reflects the status of the corresponding pin.
1	WOR	Wired-OR	Not used in DIS, IPWM, IPM and IC modes.
2	FREN	Freeze enable	This active high read/write control bit enables the MDASM to recognize the MIOB freeze signal.
3	—	0	
4	EDPOL	Polarity	In IPM and IC modes, the EDPOL bit selects the input capture-edge sensitivity of channel A. 0 = Channel A captures on a rising edge. 1 = Channel A captures on a falling edge.
5	FORCA	Force A	Not used in DIS, IPWM, IPM, and IC modes, and writing to it has no effect.
6	FORCB	Force B	Not used in DIS, IPWM, IPM, and IC modes. Writing to it has no effect. FORCA is cleared by reset and is always read as zero. Simultaneously writing a one to FORCA and FORCB resets output flip-flop.
7, 8	—	Reserved	
9, 10	BSL	Bus select	Selects which of the four possible 16-bit counter buses passing nearby is used by the MDASM.
11	—	0	
12–15	MOD	Mode select	Selects the MDASM's mode of operation. To avoid spurious interrupts, MDASM interrupts should be disabled before changing the operating mode. It's also imperative to go through the disable mode before changing the operating mode.

Table 5—*The MDASM status/control register (0x30605E) is programmed using these bits. To measure the waveform's period and frequency, program it to perform input period measurement by writing 0010 to bits 12–15.*

Bits 9 and 10 select which 16-bit counter bus the MDASM uses. Writing 00 to these two bits selects the default.

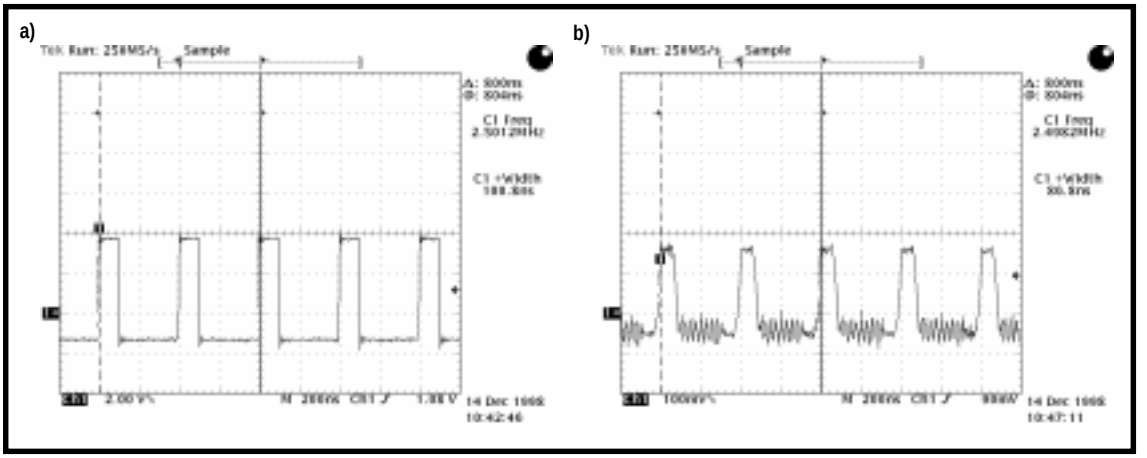


Photo 1a—*The scope capture of the MPC555 MIOS PWM output shows a 2.5-MHz 25% duty cycle signal with a fast slew rate. The pulse width is measured at 100 ns or 25% of the 400-ns period.* ***b—****This scope capture of the MPC555 MIOS PWM output shows a 2.5-MHz 25% duty cycle signal with a slow slew rate. Note the volts per division scale compared to Photo 1a.*

Other buses are available to the MDASM, depending on the frequency to be measured. To perform frequency measurement, you need to measure the input period (MOD 0010) and convert the time measurement into a frequency. It's also possible to measure the pulse width (IPWM) of the inputted signal by changing MOD 0010 to 0001.

To measure the input period, load `write -w 0x30605E = 0x0002` into the MDASM status/control register.

OBTAINING MEASUREMENTS

When the MDASM channel detects a rising edge, it writes the MMCSM counter value to its data A register. On detection of the following rising edge (one period later), the MDASM moves the first counter value into data B register and puts the new value in register A.

My MMCSM runs at 10 MHz, or counts at 100-ns intervals. Subtracting register B from register A gives the number of 100-ns counts during one period of the input waveform. To read the data in the data A and B registers, type `read -l 0x306058` into the command window of the background debug programmer.

To ensure that you read the contents of the two consecutive MDASM data registers at the same time, use a long read (`-l`). Otherwise, you might miscount because of the time it takes to read the data in two successive word reads.

The software tool returns eight hex characters in one long word. Listing 2 shows a measurement I made. Results from subsequent applications will differ from the example because the values are read from a free-running counter.

The long word `ABCDABC9` contains the contents of the data registers. Data A contains `ABCD`, and `ABC9` is in data B. By subtracting register B from register A (D – 9 in hex), you get 4. To calculate the frequency, multiply by 100 ns and take the reciprocal to get 2.5 MHz.

RESULTS

Photos 1a and 1b show the 2.5-MHz 25% duty-cycle waveform. An interesting feature of the MPC555 MIOS submodules is its ability to enable a slower slew rate on the generated waveforms. Photo 1a shows the faster setting and Photo 1b shows the slower one.

The slower transitions use less power because the keepers that pull the output high or low are turned off. This setup is useful in power-sensitive applications that require only slower waveforms to allow ample time for the gates to fully open.

A slower rise/fall time may also be useful in analog applications involving audio where low-pass filtering needs to be kept to a minimum. Notice that the volts per division are turned down to detect the slower rise times in Photo 1b.

This adjustment was needed because, at 2.5 MHz, the 200-ns rise time in the slow slew mode isn't enough time for the signal to reach its final level because the pulse is only 100-ns wide (see Photo 1a). For a slower signal and a greater duty cycle this wouldn't be an issue and you could take advantage of the gradual rise and fall times.

Now, you've seen how embedded microcontrollers with modular chip design can serve as capable signal generators and measurement tools for different types of applications. ■

At Motorola, Andrew Lillie has worked in the powertrain systems and advanced media platforms divisions. He is now involved with system-on-a-chip design technologies. You may reach him at ra8334@email.sps.mot.com.

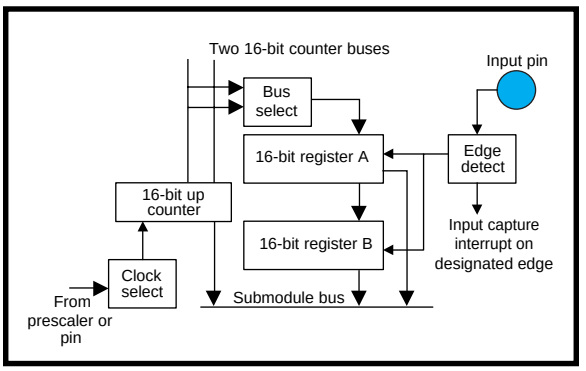


Figure 2—*Here's a MIOS double-action submodule configured for input period measurement mode. There are two 16-bit buses between the 16-bit up counter and data registers A and B.*

REFERENCES

A. Lillie, *Using the MIOS on the MPC555 Evaluation Board*, App note AN1778, Motorola, www.mcu.motsps.com.

Motorola, *MPC555 User's Manual*, www.mcu.motsps.com.

SOURCE

MPC555
Motorola
(800) 521-6274
Fax: (512) 895-4465
www.mot-sps.com

FEATURE ARTICLE

R.K. Kamat, G.M. Naik, & G.G. Tengshe

Thermistor-Based Conditional Output Sensor

Conventional sensors can cause bottlenecks in many applications, so enter the world of smart sensor technology. These authors created a thermistor-based temperature sensor with built-in hysteresis to minimize sensitivity to noise and glitches.

Over the past decade, we've seen a revolution in microelectronic circuits and devices. Today's microprocessors and microcontrollers are powerful and affordable, and they have revitalized the instrumentation world.

But, the modest suitability of conventional sensors used in microprocessor-based data-acquisition systems is becoming a bottleneck in diverse application fields.

This problem is being overcome by the rapid development of digital-output sensors that are compatible with microprocessors [1]. These so-called smart sensors have advantages like automatic calibration, automatic linearization, insensitivity to interference, elimination of cross-sensitivity, and improved frequency response.

There's no question that the hardware is becoming more complex inside these sensors but the external hardware is more simple. This new arrangement saves the cost of extra signal conditioning and conversion [2].

One application for the smart sensor is a flip-flop sensor [3]. A flip-flop sensor has a circuit that's sensitive to the measurand and, in order to sense, changes the flip-flop between a stable and an unstable state by counting the number of ones and zeros. Such sensors offer advantages like possible integration with ADCs and access by addressing a matrix of a sensor as in SRAM [1].

The most striking fault of these sensors is their sensitivity to glitches or noise impulses. In this article, we recommend a new type of thermistor-based temperature sensor with hysteresis that minimizes the drawbacks of a conventional flip-flop sensor.

IC SENSORS IN THERMAL DOMAIN

IC temperature sensors can be divided into two groups—on-chip signal conditioning with a built-in sensor and on-chip signal conditioning with an external sensor. Integrating the modules on one chip results in minimal pick-up noise, adaptive processing, the possibility of wireless interfaces, on-chip linearization, calibration, and cross-sensitivity compensation.

The drawbacks of single-chip integration include nonstandard processing steps, nonstandard initial wafer sizes, and difficulty in predicting the behavior of the material after fabrication. Choosing a package that permits optimal interaction between the sensor and measurand can be difficult, and because of the measurand, there's the possibility of damaging the sensing core and conditioning circuitry.

As early as 1966, Si and GE were predicted to be the leaders in IC temperature sensors [4]. GE's poor temperature characteristics enabled Si to take over the IC market.

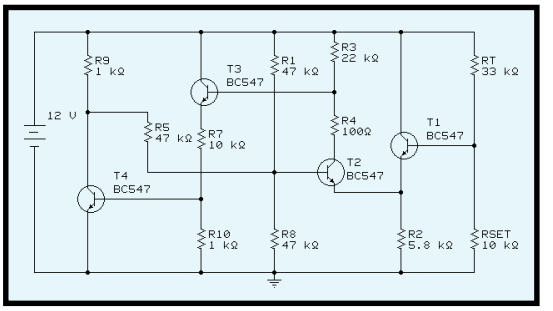


Figure 1—*Here's the full schematic of the Schmitt-trigger sensor which features high input impedance and full output swing along with less current burden on the thermistor.*

Drift in junction potential along with temperature sensing, in the case of P-N junction diodes, was the basis of first-generation IC temperature sensors and is still used today. A programmable temperature-monitoring chip with P-N junction as the temperature-sensing core was even used for tagging Atlantic salmon [5].

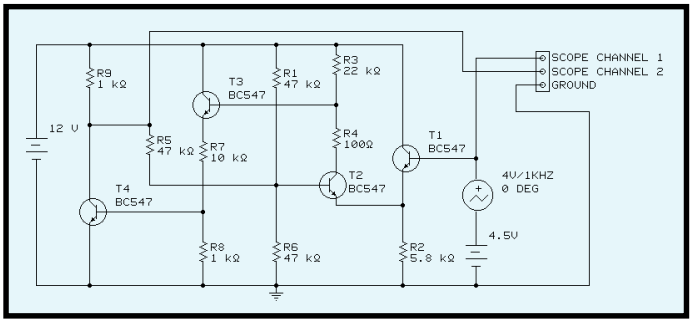


Figure 2—*The output response of the sensor was checked on Electronics Workbench. Here, the thermistor RT is replaced by a sine-wave generator to facilitate simulation.*

The transistor, which is the basic unit of the IC, exhibits good temperature sensing when connected in the negative feedback loop of an op amp. This performance seems to be the reason for the dual transistor structures found in many IC temperature sensors. These ICs use temperature-proportional delta VBE (also known as PTAT) that results from operating similar bipolar transistors at different current densities.

With the growing need for portable instruments, the focus has shifted toward low-power designs for IC temperature sensors. CMOS transistors are being used to reduce power consumption.

Other techniques, such as extensive switching of the circuit, on/off-keyed–type transmission to other modules, putting the possible number of submodules in standby mode, and hardware/software partitioning, are being implemented in the effort toward lower power. Techniques like multithreshold CMOS (MTCMOS), super cut-off CMOS (SCCMOS), and variable threshold CMOS (VTCMOS) are also being used to reduce power dissipation [6].

A CMOS monolithic temperature sensor based on compatible lateral bipolar transistors for the sensor and reference parts was developed with all CMOS circuits for ADC, control, and calibration [7]. However, the current gain of the lateral bipolar transistor and the leakage current to substrate depend on the IC processes, so the desired results are hardly feasible. That's why vertical bipolar substrate transistors are used in recent CMOS sensors [8].

The sensing is based on the PTAT mechanism with the output chopped suitably for offset reduction and then passed through a sigma-delta A/D module for digitization. The chip is 1.5 mm^2 with a temperature measurement range of –40° to +120°C, a supply voltage as low as 2.2 V, and power consumption as low as 7 µW.

IC-based temperature setpoint switches are also popular in control applications. Switching the output of an IC takes place when a sum of PTAT and its complementary CTAT pass through zero at a selected temperature set by external resistor [9]. This low-cost IC has an open-collector output and works in the range of –40° to +150°C with an accuracy of 1°C.

For some time, RTD has been the industry standard among temperature sensors. Its success is primarily due to its corrosion resistance, high melting point, and ease of purification. This sensor provides unequalled accuracy, sensitivity, stability, wide sensing range, and better nonlinearity characteristics (only of the second order).

With the progress of hybrid IC technology, it's possible to design RTD-based sensing cores on Si-based substrates. Honeywell has begun developing temperature sensors for the automobile and HVAC industries [10]. To reduce cost, they use a combination of nickel and iron in their TD series.

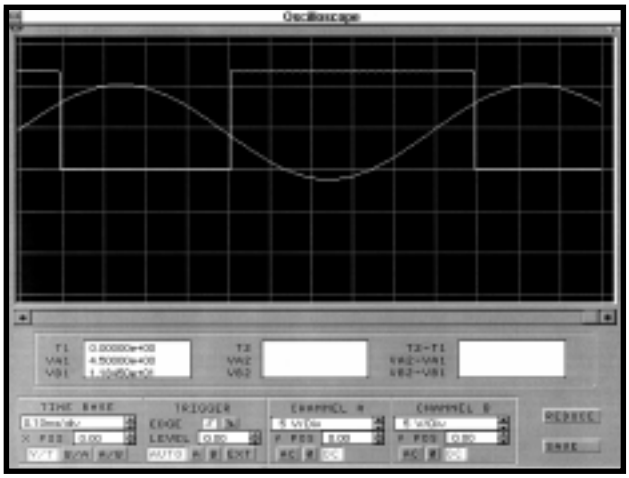


Photo 1—*This screenshot shows an enlarged version of the CRO module of Electronics Workbench. The square-wave output with sinusoidal excitation confirms the result.*

Linearization hardware is built into the chip and every care is taken to overcome the magnetoresistance of the Permalloy. A reference BIMOS chip for processing Pt-100 signals over the range of –200° to 850°C is included [11]. The output signal is time multiplexed and has frequency-modulated voltage that is suitable for microcontroller processing.

Microminiaturized amorphous GE thin-film thermistors (insulated by a thick PCVD silicon nitride layer) have been used as probes for measuring mass flow, heat conductivity, perfusion, and local temperature gradient [12]. These probes are useful for on-line blood-flow measurements in physiological and other medical applications.

Several researchers have worked out suitable methods of linearization, self-calibration, and compensation [13–15]. The latest activities in this field suggest that the present trend is to develop ICs based on RTDs and thermistors.

SENSOR DESIGN

Our sensor design is based on the emitter-coupled multivibrator [16]. The thermistor is embedded in the circuit as shown in Figure 1, and the UTP and LTP calculations are:

$$UTP = \frac{(R1 \| R5)}{(R5 + (R1 \| R5))} \times V_{CC}$$

$$LTP = \frac{(R5 \| R6)}{(R1 + (R5 \| R6))} \times V_{CC}$$

At low temperatures, the thermistor (RT) has high resistance. Because of Rset, the base of T1 is at lower voltage than the base of T2. The base of T2 is approximately at UTP.

With the increase in temperature, RT decreases, causing the base potential of T1 to rise above UTP level. In turn, this causes T1 and T5 to start, and T2 and T3 to turn off.

The circuit's advantages are high input impedance and full output swing from 0 to V_{CC}. It also reduces the current burden on thermistor in dormant mode resulting in less power dissipation.

SENSOR IN IC FORM

The Schmitt-trigger sensor circuit is easy to reproduce as an ASIC because it is based on all NPN transistors. This setup eliminates incorrect triggering (the result of mismatched VBEs on T1 and T2) by careful design of device layout in the photomasking stage.

To further minimize the effects of the tolerance and temperature coefficients on trigger-level definition, we recommend using a thin-film network (preferably of metal film resistors) of R1, R5, and R6. The components added externally to this proposed ASIC are a thermistor and Rset, which can be chosen to suit the application. A simulation of this circuit using Electronics Workbench is shown in Figure 2.

In some biomedical applications where the measurand temperature range is limited, the thermistor can be placed on the substrate using hybrid techniques. A hybrid IC design combines the advantages of silicon processing with exotic sensing principles.

The basic circuit can be arranged as cells in rows and columns. The output of this matrix sensor is accessed via addressing techniques similar to those used in SRAM. The matrix sensor can be duplicated using VLSI technology.

IC AND DISCRETE VERSION

Our sensor design is suitable as an ASIC or in a discrete version. The transistors presently available in the market are shown in Figure 2 so you can test the sensor in discrete form.

However, the IC version has some unique advantages—primarily, the elimination of incorrect triggering. This feature is possible because of the ease with which the well-matched transistors can be fabricated in ICs. A thin-film or laser-trimmed network of R1, R2, and R5 further reduces the effect of tolerance and temperature coefficient on trigger-level definition.

NOISE IMMUNITY

The noise immunity of our sensor is a function of the hysteresis introduced. The existing flip-flop sensor is sensitive to even a weak noise impulse produced due to temperature drift or supply variation.

But, the Schmitt-trigger–based sensor design can be made insensitive to noise. You do need to accurately estimate the noise, however [17].

After obtaining the noise estimate for a particular application, the resistors R1, R2 and R5 can be chosen so as to keep the hysteresis gap slightly more than the noise voltage. This setup helps to mask the noise.

Other techniques, like ensemble averaging, can be applied in software to reduce low-frequency noise. In the present case, the noise margin kept is:

$$UTP - LTP = 8\ V - 4\ V = 4\ V$$

However, there's a tradeoff between sensitivity/resolution and noise immunity. For better sensitivity, the amount of hysteresis should be minimal (as with a flip-flop sensor). In a Schmitt-trigger sensor, the noise immunity is greater, at the expense of lack of sensitivity in the deadband.

APPLICATIONS

The circuit in Figure 1 can be used as a low-cost high-performance temperature switch in applications such as over/under-temperature alarm, electric irons, over-temperature warnings in PC board applications, and more.

If the thermistor is biased in a self-heated region, the circuit can be used as vacuum alarm, CO_2 detection, or even as an ice indicator on highways. R1, R5, and R6 should be used to introduce enough hysteresis to overcome any expected glitches.

The circuit's hysteresis is user definable and can be set by choosing appropriate values of R1, R5, and R6. With a mechanical or solid-state relay (e.g., a triac) as a load to control the AC cycle, the circuit can work as a proportional controller. During the deadband, the final control element remains unaware of the measurand status.

Thermistor arrays are required in many medical applications, including measuring regional cerebral blood flow. The micrologic version of the circuit can be used for this purpose with an array of thin-film thermistors suitably arranged on the substrate.

This circuit can also be used to study temperature changes in relation to change in the composition of anesthetic gas. Or, it can record chemical reaction enthalpy in a microcalorimetric device.

APPLY NOW

Here, we emphasized the importance of microprocessor-compatible sensors and reviewed IC temperature sensors. The main drawback—sensitivity to noise—is eliminated by proposing hysteresis in the flip-flop sensor.

We designed a new kind of Schmitt-trigger sensor by using NPN transistors to suit integration as an ASIC.

The design team is currently upgrading the sensor by including modules to achieve lower power, auto-diagnostics, self-calibration, linearity check, over-temperature warning, and more. A similar design is in the process for RTD sensors.

R.K. Kamat is a lecturer in electronics at Goa University and is in charge of the university's Internet gateway. His interests include intelligent instrumentation, sensor design and fabrication, computer networks, and signal processing. You may reach him via rkkamat@unigoa.ernet.in.

G.M. Naik heads the Instrumentation Department at Goa University. His interests include electronic instrumentation, fiber-optic sensors, computer networks, and optical signal processing.

G.G. Tengshe heads the Research and Development Department of D. Y. Patil College of Engineering, Kolhapur, India. His interests include biomedical engineering, computer interfaces, thermal system design, and power electronics.

REFERENCES

[1] A.W. Van Herwaarden and R.F. Wolffenbuttel, "Introduction to sensors compatible with microprocessors," *Microprocessors & Microsystems*, **14**, 74–82, 1990.

[2] M.R. Haskard, "An experiment in smart sensor design," *Sensors and Actuators*, A24, 163–169, 1990.

[3] W. Lian and S. Middelhoek, "Flip-flop sensors: A new class of silicon sensors," *Sensors and Actuators*, **9**, 259–268, 1986.

[4] T.H. Herder, R.O. Olson, and J.S. Blackemore, *Revised Science Instruction*, **37**, 1301–1305, 1966.

[5] G. Fischer, C.W. Recksick, and K.D. Friedland, "A programmable temperature monitoring device for tagging small fish: A prototype chip development," *IEEE Transactions on VLSI*, **5:4**, 1997.

[6] K. Roy, A. Raghunathan, and S. Dey, "Low power design methodologies for systems-on-chips," Int'l Symposium on VLSI for information appliance, India, 1999.

[7] P. Krummenacher and H. Oguey, "Smart temperature sensors in CMOS technology," *Sensors and Actuators*, A21–23, 636–638, 1990.

[8] A. Bakker and J. Huijsing, "Micropower CMOS sensor with digital output," *IEEE Journal of Solid State Circuits*, **31:7**, 1996.

[9] A.P. Brokaw, "A temperature sensor with single resistor set point programming," *IEEE Journal of Solid State Circuits*, **31**, 1908–1915, 1996.

[10] H. Hencke, "The Design and application of Honeywell's laser trimmed temperature sensors," *Measurement & Control*, **22**, 233–236, 1989.

[11] G.C.M. Meijer and C.H. Voorwinden, "A novel BIMOS signal processor for Pt-100 temperature sensor with microcontroller interfacing," *Sensors and Actuators*, A25–27, 613–620, 1991.

[12] H. Kuttner et al., "Microminiaturised thermistor arrays for temperature gradient, flow and perfusion measurement," *Sensors and Actuators*, A25–27, 641–645, 1991.

[13] P.P.L. Regtein and P.J. Trimp, "Dynamic Calibration of sensors using EEPROMs," *Sensors and Actuators*, A21–23, 615–618, 1990.

[14] P. Hille, R. Hohler, and H. Strack, "A linearisation and compensation method for integrated sensors," *Sensors and Actuators*, A44, 95–102, 1994.

[15] P.T. Kolen, "Self calibration/compensation technique for microcontroller based sensor arrays," *IEEE Transactions on Instrumentation and Measurement*, **43**, 620–623, 1994.

[16] T.K. Hemingway, "Circuit consultants case book," *Business Book Limited*, 128–137, 1970.

[17] J.K. Atkinson, R.P. Sion, and S. Sizeland, "The characterization and compensation through sensor array signal processing techniques of drift and low frequency noise in thick film semiconductor sensors," *Sensors and Actuators*, A41–42, 607–611, 1994.

Video Switch

FEATURE ARTICLE

Cullen Jennings

Cullen needs a video multiplexer, but not just any multiplexer—one that receives input from multiple cameras, detects video-sync signals, and switches the multiple inputs on each field. He can't get this device in the local electronics store, so he builds it!

In a large computer-vision application I'm working on, I use multiple video cameras to grab stereo pairs of images. I needed a video multiplexer that takes inputs from multiple cameras, detects the video sync signals, and switches between the multiple inputs on each field. So, I constructed one.

My video multiplexer accepts six inputs and switches them to three outputs. It does sync detection and uses a Xilinx 9500-series PLD to implement the switching logic.

The PLD is in-system programmable, so it can be changed to implement switching schemes such as alternating between camera inputs every 3 s, every frame, on each field, or on a certain area of the image. This last scheme enables a banner from one video input to be superimposed over another video image, and other picture-in-picture–type applications are possible as well.

Although my video multiplexer is primarily designed for NTSC video applications, the bandwidth is adequate for switching video signals for high-resolution monitors. The switch also implements a simple DAC that outputs a video test signal that produces an image with a few simple gray bars. This test image demonstrates the simplicity of synthesizing a video signal using a relatively inexpensive PLD.

This project is interesting from several points of view. The PLD's programming is all done in VHDL. This powerful language is becoming much more available to engineers with small budgets.

The project is constructed using surface-mount components. Also, the PCB is only two layers, which keeps the price down. Video cameras have become so inexpensive, I'm sure their use will increase in both commercial and noncommercial projects.

This circuit is a useful tool for any project involving multiple video cameras. It is inexpensive (about $50), the components are widely available, and it is not particularly difficult to build.

CONCEPT

Figure 1 shows the circuit diagram. There are three video outputs, which can choose either three A or three B inputs. The logic block selects the inputs to use and receives information from the sync-detection circuit. It also synthesizes two output video signals that can be used as sync channels or other video test signals.

I connect the A and B channels to two RGB video cameras that are genlocked together, and I also connect the output to a video digitizer card in a computer. The PLD is programmed to switch on each field.

When I capture a frame, the even field is from one camera and the odd field is from the other. In software, I separate these into two images and process them through a stereo correlation algorithm to compute the distance from the cameras to the objects viewed.

In another application, I can take a certain region of the screen from in-

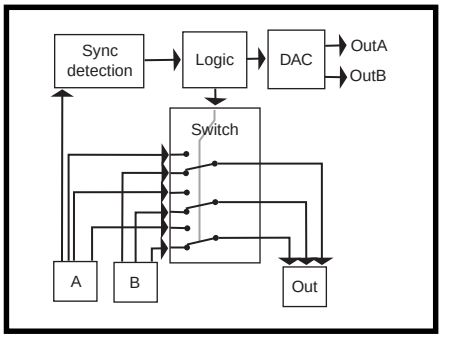


Figure 1—*In this diagram of the video switch, the signals flow from left to right. The DAC outputs can provide a regenerated sync signal or some other video test pattern.*

put A part of the time and from input B the rest of the time, enabling a banner or rectangle from the image A to be overlaid on the image B.

The system can be programmed to switch inputs every few seconds to allow viewing through two cameras or to put the view from the first camera in the top half of the image and the view from the second camera in the bottom half.

The ability to generate test patterns can be used to timestamp the video. Basically, you generate a test pattern that encodes the frame number as a binary pattern on the image. Then, use the switcher to overlay this image onto the left edge of the input video. This system is convenient for finding out exactly how many frames your video-capture card is dropping and whether it ever duplicates frames.

A slight modification of the circuit is required (diode and two resistors) to feed the AC signal into one of the PLD inputs so that it can synchronize the output sync signals with the AC line signals. Video cameras can be genlocked to this input signal, and the video-camera captures are synchronized with the flicker of the AC fluorescent lights. In some cases, this action significantly improves the image.

Something I have not yet looked into but am curious about is whether it's possible to regenerate portions of the sync signal that are corrupted by, say, a video copy-protection scheme. It may be easy just to switch over to the synthesized sync on the scan lines having a corrupted sync signal.

The project is also convenient if your scope doesn't have a TV trigger. Feed a video signal into this device and use one of the digital outputs from the PLD as a trigger for your scope.

RS-170 VIDEO SIGNALS

A video sequence is a series of still images called frames. Each frame consists of 525 horizontal lines called scan lines. The frame is split into two fields with the odd scan lines in the first field and the even in the second. The video signal transmits one frame after another.

For each frame, the first field is transmitted followed by the second. The fields are sent by transmitting each of the scan lines with appropriate delays and signals between each scan line, field, and frame so that the receiving system can tell what part of the image is being received. The gray-scale value of the image at a given location on the scan line is transmitted by encoding it as a voltage between 0.357 and 1.0 V.

Consider Figure 2. At the start of the scan line, the image is white, so the output voltage is 0.7 V. In the gray region, the output voltage drops to 0.5 V, and in the black area, it drops to 0.3 V. The scan lines occur at a rate of 15,734 Hz (about every 63.5555 ms).

At the start of each scan line, a horizontal sync is transmitted so the system knows where the scan line starts (see Figure 3a). Here, we have a back-porch section that is 1.18 ms long at 0.306 V, followed by the actual sync pulse that is 4.7 ms long at 0.020 V. Finally, there's the front porch for 3.14 ms at 0.306 V.

The first several scan lines in each field are not used for image data but are used for vertical sync and other purposes. The first three scan lines of the field contain equalizing pulses, followed by three lines of serration pulses and three more lines of equalizing pulses, as shown in Figure 3b.

The next 11 lines generally contain black scan lines but might have other

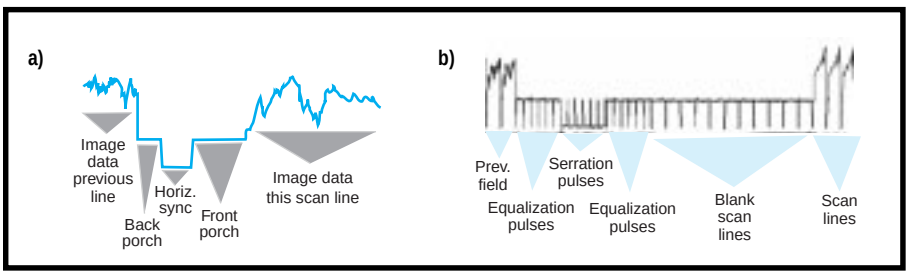


Figure 3a—*This figure depicts a horizontal sync signal between two scan lines. Time advances along the horizontal axis and the voltage is shown on the vertical axis.* ***b—****This figure depicts a vertical sync between two fields. Voltage is on the vertical axis and time advances along the horizontal axis.*

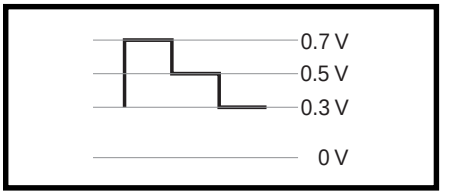


Figure 2—*This small graph depicts a video signal for a single scan line. Voltage is on the vertical axis, and time advances along the horizontal axis. The image is white for the first third of the line, gray for the second third, and black for the last third.*

information (e.g., closed caption). The equalizing pulses are 2.3 ms wide, and the serration pulses are 4.7 ms wide.

SCHEMATICS

The system's video output can be selected from the A or B video input. The logic block selects the input and uses the information provided by the sync-detection circuit to decide when to switch video inputs. The logic block also drives a simple DAC that generates a video output signal on the two sync out lines (see Figure 4).

All video signals coming into the system are terminated with a 75-Ω resistor. Depending on the jumper settings of JP1 and JP2, either the green A signal or the sync A is fed into the sync-detection circuit.

The signal is filtered in the passive RC circuit formed by R3, C7, and C9. The filtered signal is passed into the U5 chip, which is a National LM1881 chip that handles the sync detection.

I used Electronics Workbench to compute the frequency response curve for this filter (see Figure 5). This tool forms a convenient front end to SPICE. It has about a –18-db attenuation of the color burst signals but little effect on low-frequency sync signals.

This filter is only required if color signals are being fed into the system. It can be removed from the system by removing C7 and C9. There's no need to short R3 because it has little effect relative to the high input impedance of U5.

Various sync signals are fed into the PLD (U8 in Figure 4), which can be programmed to select the correct input.

The PLD is a Xilinx 9536. It receives a clock signal from the oscillator and controls two LEDs. JP6 enables the logic section to be easily connected up to another circuit or a logic analyzer.

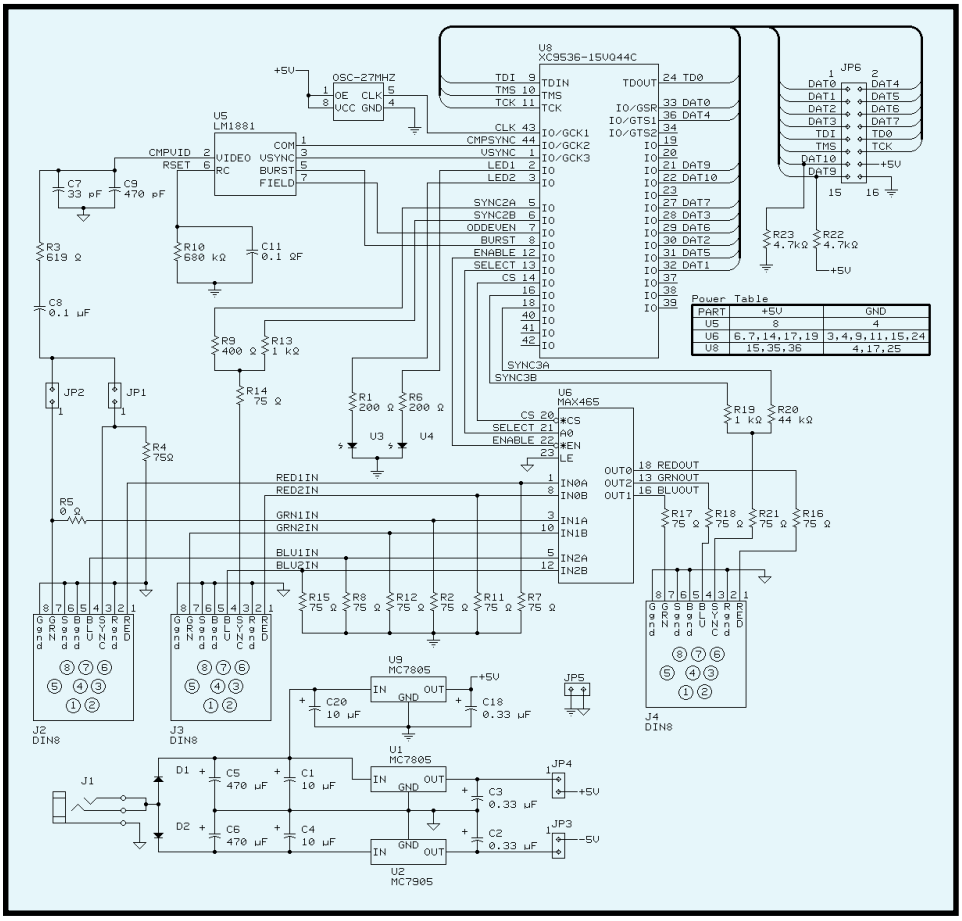


Figure 4—*In this schematic, signals generally flow left to right. The +5 V off of JP4 drives only the analog chips, whereas the other +5 V drives all of the digital chips.*

Data lines 0–7 can be used as general-purpose I/O. Data lines 9 and 10 are pulled to certain logic levels with R22 and R23. They can be pulled to the opposite levels by putting jumpers on JP6. Digital power is also available on the jumper to power some other circuit.

The PLD controls U6, a MAX-465 video switch that controls three channels and can select each channel from one of two inputs. It amplifies the signal by a factor of two, enabling you to divide the signal by two by running it through a 75-Ω resistor before driving the 75-Ω transmission line.

This setup makes it easy to match impedance and avoid over-driving the output amplifier, even if the output line is shorted to ground. It also double terminates the transmission line, greatly reducing reflections.

The logic block outputs a two-bit signal into the DAC built by a two-resistor network. R9 and R13 form the DAC for the B input sync, and R19 and R20 form the DAC for the output sync. Because the PLD outputs can be set to 0, 1, or Z (tristated), six output voltages are possible from this DAC.

The power-supply circuitry was designed to keep the noise on the power to the analog chips low. I considered using a switching supply because of concerns about heat dissipation, but I decided the linear power regulator caused less noise on the video output.

The digital circuits have their own supply because this was a simple way to reduce transient noise on the analog lines caused by high-speed digital switching. When substituting capacitors or transformers on the input side of the power regulators, keep in mind that a 12-VAC transformer means 12 V_{RMS}, resulting in a DC voltage of 17 V.

Figure 6a shows the input video signal and associated outputs of the

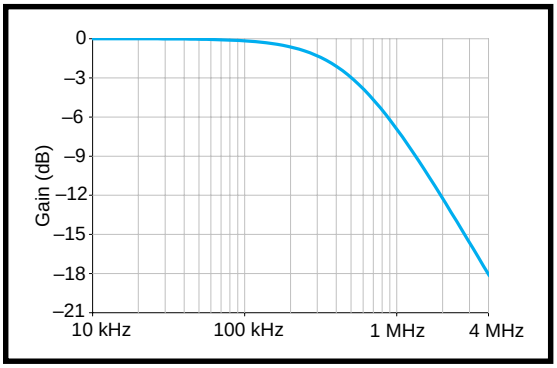


Figure 5—*The calculated filter response has a nearly –18-dB effect on the color burst signal and little effect on the sync signal, which is mostly under 500 kHz.*

sync-detection circuit. The top trace is the video input for one scan line with a horizontal sync at both ends. The second trace is the field, which only changes on a vertical sync. The next trace, the burst, indicates when the color burst signal is active. The last trace is the sync, which is active during the horizontal sync.

Figure 6b is a longer segment of the signal around a vertical sync. It shows the same signals as Figure 6a but over several lines and a vertical sync. The field changes at the start of the vertical pulse.

The board has a mini-DIN–style connector that's similar to an SVideo connector. It doesn't take up much space, but making cables is a pain. It might have been better to use BNC connectors.

I designed this circuit to work with NTSC, PAL, and SECAM–style video signals but only tested it with NTSC. The only values that deserve much consideration are C11 and R10, which form the time constant for vertical-sync detection. The voltage levels generated for output signals by R9, R13 and R19, R20 also need to be changed.

By the way, instead of using the Xilinx part, you can also do this project using a Cypress ISP PLD or AMD Mach ISP PLD.

LAYOUT AND CONSTRUCTION

The PCB layout was a bit of a challenge. I wanted a ground plane under the whole analog section, but I also wanted to keep costs low.

There are almost no breaks in the ground plane, and I got it on two layers. JP5 provides a simple way to control where the analog and digital grounds get connected. The digital lines are kept well separated from the analog lines.

With the exception of the filter formed by C7, C9 and R3, none of the values are critical. Even the filter values aren't that critical, but you should play with a SPICE model of the filter before you change them. The LM1881 datasheet explains the goals of this filter.

After you solder on all the SMT resistors and capacitors, check that there are no shorts between V_{CC}, +5, –5, and ground and add all the components in the power supply. Do the big capacitors last because they make it harder to reach other components. Check that all the voltages are correct before you add any expensive stuff.

After adding the chips, I applied power to the board and checked that none of the chips got hot. I then added the connectors and remaining components and programmed the PLD.

Next step: check that the input resistance on all the video inputs is 75 Ω. If everything looks good, connect up a bunch of video signals and go for it.

If it doesn't work, start tracing the signal through from the start. I powered up the board and fed a video signal into the sync input and checked a few points: the output of the filter on pin 2 of U5, that the field line (pin 7 of U5) is a 30-Hz square wave, and that the PLD output the same square wave to pin 2 of connector JP6. It should also cause the green LED to flicker at 30 Hz.

Check that a signal applied to the A input gets chopped up at 30 Hz, and similarly for B. Also check that this signal is going to U6. Now trace the signals for the video in and out of U6.

VHDL

VHDL is one way to describe the desired operation of a logic device. Although it's a standard, vendors tend to choose different parts to implement and add their own quirks.

I use Xilinx PLDs with the Xilinx foundation tool chain for place and route and Synopsys FPGA Express for

synthesis. This approach is great for large FPGAs, but it isn't cheap.

A less expensive system that works well for projects using small PLDs is the Cypress WARP2 system. For a few hundred dollars, you get a VHDL textbook, a VHDL synthesis tool, an ISP cable, and a few PLDs.

An even cheaper (i.e., free) solution is to use software from AMD and a Mach PLD. It won't do VHDL, but it uses a language at about the level of the (downloadable) PLD equations.

You can also download the complete code for the application to switch on each field. In the first part, I declare all the signals connected to the PLD and set the pin numbers.

If the line where field is assigned to `sel_out` is commented out and the next line after it is commented in, the system takes the first input for scan lines between 01111111 and 11000000 and takes the rest of the image from the other input.

Various signals are put out to the test connector so they can be monitored with a logic analyzer. `vsync` detects the vertical sync by looking at the sync signal's value during the color-burst phase of the sync.

`hsync` generates the horizontal sync signal. It's a flip-flop that is set whenever there is a sync that is not the vertical sync. It is reset on the start of the next burst event.

The line-count process sets the line to the count of the line in the field you're on. It counts the horizontal sync pulses and uses the vertical sync pulse to reset to zero.

Finally, the digital outs to the DAC are generated. The gray-scale value depends on the scan line you are at.

You can make simple modifications to the VHDL code. To switch every *n* frames, set up a counter to count even fields and switch on an appropriate value. To switch part way through a scan line, set up a counter that is reset by the horizontal pulse and that counts clock ticks from the oscillator, switching the output when it hits an appropriate value.

PROGRAM IT

The JTAG protocol was designed for testing the connection between chips.

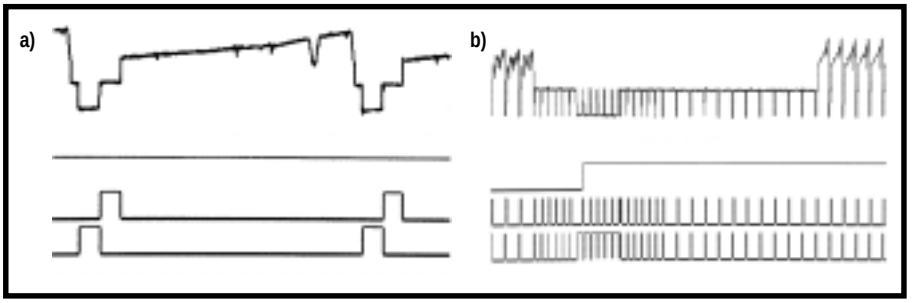


Figure 6—*The top lines show the video input signal, and the second lines show the field output of the PLD. The third lines show the color burst active signal, and the bottom lines show the horizontal sync signal.* ***a—****Here you see a complete scan line and the horizontal sync on either end of it.* ***b—****Here are many scan lines containing a vertical sync.*

It is useful for many things including debugging and programming devices.

Basically, it's a serial protocol. Each device has four wires—a TMS (test mode) signal, a TCK line that clocks the serial data into the system, and TDI (test data in) and TDO (test data out).

The TMS and TCK are connected together and controlled by the master device doing the testing. The TDI and TDO are daisy-chained together to form one long string with all the devices through which the serial data passes.

Different commands can be sent to each device, including commands to pass data on the serial line to the next device in the chain. Basic commands enable the chip to set the output level of any pin on the chip and read the logic level of pins on other chips. Thus, a JTAG test system can check that all the traces are correctly connected and there are no shorts.

Extended commands let you program the PLD via the JTAG interface. In-system programming is convenient because there's no need to own a programmer that has sockets for different SMT parts, pins don't get bent as parts go in and out of the programmer, and

there's no need to socket the part on the board so that it can be reprogrammed.

It's simple to program the PLD with Xilinx's XChecker cable and their software. Connect the JTAG pins on the cable and the ground but not the V_{CC}, power up the board, and program it.

If you don't have Xilinx software and are going to use my configuration files, get the XAPP058 app note and the associated software for PCs. Wire up a cable so you can use your parallel port data lines to drive the TDO, TMS, and TCK lines, and read the TDI line by connecting it to the ERROR line. With this circuit, there's no need to buffer the signals (see section 10.20 of *The Art of Electronics*).

Use the software in the app note to program the PLD. You need to modify the port access code to match the wiring of your parallel cable (see section 29.1.5 of *The Indispensable PC Hardware Book*).

I enjoyed building this project, and it works great. Let me know about new applications you find for it.

Thanks to Stewart Kingdon, Rod Barman, Lyndsay Campbell, and Alan Hawrylyshen.

Cullen Jennings works for Image Integration, a computer consulting company that develops software and network solutions for oil companies and software for air traffic control. You may reach him at c.jennings@ieee.org.

SOFTWARE

Project files are available via the Circuit Cellar web site, including the schematics in Protel, net lists, BOMs, Gerber and Postscript plots of the PCB, drill files, VHDL code for the PLD, reduced equations of the PLD that can be implemented in a language other than VHDL, and configuration images for programming the PLD.

REFERENCES

P. Horowitz and W. Hill, *The Art of Electronics*, Cambridge University Press, New York, NY, 1989.

K. Jack, *Video Demystified*, High Text, San Diego, CA, 1996.

H.W. Johnson and M. Graham, *High-Speed Digital Design: A Handbook of Black Magic*, Prentice Hall, Englewood Cliffs, NJ, 1993.

Maxim, MAX463–470 Two Channel, Triple/Quad RGB Video Switches and Buffers, www.maxim-ic.com/efp/AllParts.htm.

H.-P. Messmer, *The Indispensable PC Hardware Book: Your Hardware Questions Answered*, Addison-Wesley, Reading, MA, 1995.

National Semiconductor, *LM1881 Video Sync Separator Datasheet*, www.national.com/ds/LM/LM1881.pdf.

H.W. Ott, *Noise Reduction Techniques in Electronic Systems*, Wiley & Sons, New York, NY, 1988.

S. Sjoholm and L. Lindh, *VHDL for Designers*, Prentice Hall, Englewood Cliffs, NJ, 1997.

K. Skahill, *VHDL for Programmable Logic*, Addison-Wesley, Reading, MA, 1996.

Xilinx, *The Programmable Logic Book Databook*, San Jose, CA, 1998.

SOURCES

9500-series PLD
Xilinx Corp.
(408) 559-7778
Fax: (408) 559-7114
www.xilinx.com

LM1881
National Semiconductor
(408) 721-5000
Fax: (408) 739-9803
www.national.com

MAX465
Maxim Integrated Products
(408) 737-7600
Fax: (408) 737-7194
www.maxim-ic.com

Electronics Workbench
Interactive Image Technologies, Ltd.
(800) 263-5552
(416) 977-5550
Fax: (416) 977-1818
www.interactiv.com

WARP2
Cypress Semiconductor Corp.
(408) 943-2600
Fax: (408) 943-2741
www.cypress.com

TELEPHONY APPLICATION PROCESSOR

The **TAP-810** is a CompactPCI-based DSP resource card designed for PSTN to voice-over-IP connectivity applications. The board features quad T1/E1 line interfaces, a 100BaseT controller, and it is specified for 120 channels of G.723.1 and G.729a as well as other standard and proprietary algorithms.

This card lets voice and data entering the T1/E1 interface be compressed by the DSP resource and exit through the Ethernet interface without unnecessary processing by the host or bandwidth transfers over the bus. The host is free to manage call setup and teardown and to calculate and support custom calling features for packet-switched calls.

The TAP-810 is fully hot-swap compliant, permitting board installation and removal in a running system and management of available resources by application-level software. All connections to the TAP-810 are made through a rear-panel I/O transition module, which simplifies cabling and maintenance. This high-density combination of inputs, outputs, and DSP power provides the necessary resources to create large systems. With 120 channels-per-board gateways, a standard 19″ chassis can house up to 672 ports.

The TAP-810 is available at a per-port price of **$125** in OEM quantities.

Analogic Corp.
(978) 977-6817
Fax: (978) 977-6813
www.analogic.com/cti

RECONFIGURABLE DSP SYSTEM

MiroTech's **Aristotle** reconfigurable DSP board enables electrical and computer engineering instructors to teach application and system development on a DSP platform. The half-size, PCI add-on board features a Texas Instruments TMS320C44 DSP (running at 60 MHz) tightly coupled to a Xilinx XC4036 FPGA in the well-proven HPRC (reconfigurable) architecture. Both the FPGA and the DSP are independent and are accessible by users through designated ports. MiroTech's 16-bit high-speed banks of SRAM support the main processing elements. A test bus controller provides testability through a JTAG scan chain.

Aristotle can be configured with various forms of support software including RTOSs, compilers, place and route tools, and more. It also supports plug-and-play installation under Windows 95/NT. Aristotle's communications capabilities include a high-speed PCI interface, JTAG emulation, and four TIM-standard communications ports. An IndustryPack mezzanine site provides access to a wide variety of I/O modules.

MiroTech Microsystems, Inc.
www.mirotech.com

NouveauPC

edited by Harv Weiner

SOFTWARE DEVELOPMENT TOOLS FOR SHARC DSPs

Blue Wave Systems has combined Analog Devices' VisualDSP development tools with its own IDE6000 development environment to provide a powerful integrated software development environment for SHARC-based DSP systems. Although VisualDSP and IDE6000 remain distinct applications, Blue Wave has created a comprehensive set of guidelines to ensure that users receive maximum benefit from using the two in tandem.

The VisualDSP development toolkit consists of three main parts—a compiler tool chain, a simulator, and a project manager. The C compiler generates efficient code that is optimized for both code density and execution time. The user can include assembly-language statements in the C code for time-critical loops. Customers can also use pre-tested math, DSP, and C runtime library routines to help shorten time to market.

Combining the integrated compiler and project manager of VisualDSP with the system-level test and utility functionality provided by IDE6000 enables the user to concentrate on the DSP application rather than wasting time on hardware configuration or software management.

The combined VisualDSP and IDE6000 package is being offered at an introductory price of **$3000**.

Blue Wave Systems, Inc.
(972) 277-4600
Fax: (972) 277-4666
www.bluews.com

PASSIVE-BACKPLANE CPU BOARD

MantaRay is a highly integrated Pentium II passive-backplane CPU board. Using the Intel 440BX chipset, MantaRay supports processor speeds up to 450 MHz with a 100-MHz external clock and up to 333 MHz with a 66-MHz external clock. Optional features include onboard Ethernet that supports 10/100BaseT, a SCSI controller that supports speeds up to Ultra 2, and video support that includes 2 MB of SDRAM to work at resolutions up to 1280 × 1024 at 256 colors.

MantaRay supports up to 1-GB SDRAM and contains its own internal Level II cache embedded in the Pentium II CPU module. It supports ECC in the Level II cache as well as main memory, further enhancing data integrity. Additionally, MantaRay has a DiskOnChip socket to add flash-disk capability directly on the CPU board.

Other features include two RS-232–compatible serial ports, one ECP/EPP parallel port, a PCI IDE interface that supports up to four enhanced IDE devices with up to mode 4 PIO, a USB, and a floppy drive controller supporting up to two 5.25″ or 3.5″ floppy drives. Mini-DIN connectors are provided for a PS/2 mouse and a keyboard interface, and an onboard field-replaceable battery powers the real-time clock. The PCI bus features a 30-/33-MHz clock speed and a 32-bit data path, improving throughput capacity for high-speed peripherals.

MantaRay's price ranges from **$1775** to **$2275** with all the options.

I-Bus, Inc.
(619) 974-8426
Fax: (619) 268-7863
www.ibus.com

Nouveau PC

Peter Petersen
&
Tom Schotland

Win32 and Real Time

Tempted to use Windows NT or 95 as an RTOS? Peter and Tom have an alternative. They create a Win32-compatible environment with dedicated embedded-system development tools—less overhead, no real-time deficiencies.

There are many arguments for using Windows NT or 95/98 for embedded systems. Mass production has led to decreasing prices for PC hardware and software. Both Windows systems support true 32-bit flat-address programs and allow full utilization of current 32-bit CPUs.

Software development tools are available and have been tested by countless developers. And, CPU vendors offer a range of 32-bit Intel i80386-compatible microcontrollers like Intel's '386EX, AMD's Élan series, and National Semiconductor's NS486SXF.

But, there are problems, too. Windows NT and 95 need a lot of computer resources. Windows 95 needs at least 16 MB to run smoothly, and Windows NT needs 32 MB to perform well (not counting the application's needs).

Both systems require 100–150 MB of hard disk space. Even Microsoft's Windows NT Embedded variant requires 16 MB of RAM and 20 MB of ROM or other permanent storage.

Another important issue is real time. Windows was designed for home and office applications, which (except for games or multimedia applications) have weak or no real-time requirements.

The ability to assign real-time priorities to a process may mislead developers to believe that Windows 95 and NT support real-time processing. But, Windows dynamically changes priorities at runtime to achieve a more equal or fair CPU time distribution. Sharing the CPU equally is not the goal of a real-time system.

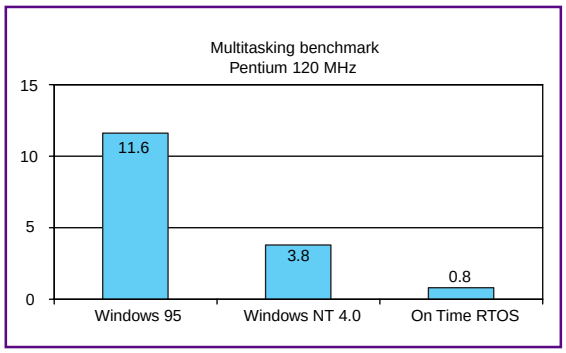


Figure 1—The benchmark performs 400,000 semaphore operations with 200,000 task switches. The times shown here are for a 120-MHz Pentium.

Win32's priority scheme implementation is also subject to priority inversion for application threads and deferred procedure calls of device-driver interrupt-service routines.

And, nonpreemptable system calls can use a nondeterministic amount of time. The internal timer resolution is fixed at 10 ms—too coarse for many systems. Besides, Windows does not guarantee that timer callbacks can be serviced within a maximum delay.

Application speed is another issue. Demon processes to implement various OS services such as virtual memory management can steal CPU time from a time-critical application.

Multitasking with Windows NT and 95 is not as efficient as in dedicated real-time systems. For example, running the code in Listing 1 under Windows NT, Windows 95, and an RTOS yields the results graphed in Figure 1.

Last but not least, licensing costs are a crucial factor for embedded systems development.

A SOLUTION?

Several products claim to add real-time capabilities to Windows NT. And, there are different approaches to solving the real-time problem.

Special ISA or PCI cards can be used to map normal interrupts to the PC's non-maskable interrupt to reduce interrupt latencies caused by NT device drivers. However, the real-time application has to run within a device driver, making access to and from regular applications difficult.

Software development requires a thorough knowledge of NT's device drivers. Device drivers run at the CPU's highest privilege level without protection for the NT kernel or system data structures. A simple bug can overwrite system memory and crash NT.

Replacing NT's hardware abstraction layer (HAL) is an approach that attempts to fix problems below the kernel by, for example, providing a higher timer-interrupt frequency. But, the fundamental real-time deficiencies of the kernel can't be fixed. A modified HAL may improve NT's soft real-time behavior but not meet the hard real-time requirements of application threads.

Another approach is to run NT as a single task of a true RTOS. Although such a system can yield deterministic time behavior, real-time tasks run completely isolated from the Windows world, requiring complex communications mechanisms.

Software development requires good knowledge of both the RTOS and Windows NT. This approach requires the use of Intel's hardware task switching, which is slow and increases interrupt latencies (but yields deterministic upper bounds).

Each of these approaches has its problems, and the high resource demands of NT are not reduced (and may even be increased). The performance achieved is either only soft real time or impeded by general performance bottlenecks. All solutions carry additional royalties and increase the OS licensing costs.

Products that need to be deeply integrated into the NT kernel (e.g., at the HAL level) will always lag behind the latest OS version. Also, they'll never be tested by such a wide user base as the desktop

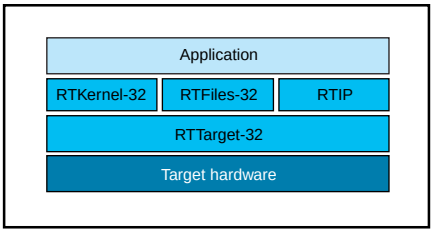


Figure 2—The On Time Win32/NT-compatible embedded RTOS has a scalable component architecture. Only the components needed are loaded on the target.

version of NT, so it's questionable whether they'll have the same degree of compatibility and stability.

BEST OF BOTH WORLDS

Windows NT and 95 just weren't designed for embedded systems. Why not use a system designed for such environments that has a Win32-compatible API?

This solution combines the advantages of Windows NT with those of real-time systems. And, being Win32 compatible doesn't compromise any of the features you'd expect from an embedded-systems OS. The system equirements would include:

- Win32-compatible API
- real-time extensions to the Win32 API
- support for mainstream Win32 development tools
- support for standard run-time systems
- scalability
- access to the computer hardware at the application level

Besides the Win32-specific features, the system should provide general features needed for real-time and embedded systems programming:

- low interrupt latencies
- memory protection
- deterministic real-time scheduling

Listing 1—Here's the source code for the benchmark program used to obtain the benchmark results. This program can run without source-code modifications under Windows 95, NT, and the On Time Win32-compatible embedded RTOS.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#define LOOPS 100000

HANDLE S1, S2;
DWORD WINAPI ThreadA(LPVOID lpdwParam)
{
  while (1){
    WaitForSingleObject(S1, INFINITE);
    ReleaseSemaphore(S2, 1, NULL);
 }
  return 0;
}
DWORD main(void)
{
  HANDLE H;
  DWORD ThreadID, T, i;
  S1 = CreateSemaphore(NULL, 0, 1, "BenchSema1");
  S2 = CreateSemaphore(NULL, 0, 1, "BenchSema2");
  H  = CreateThread(NULL, 0, ThreadA, NULL, 0, &ThreadID);
  Sleep(100);
  printf("switching between two tasks %i times...\n", LOOPS);
  T = GetTickCount();
  for (i=0; i<LOOPS; i++){
    ReleaseSemaphore(S1, 1, NULL);
    WaitForSingleObject(S2, INFINITE);
  }
  T = GetTickCount() - T;
  printf("Time for %i loops: %i milliseconds\n", LOOPS, T);
  TerminateThread(H, 0);
  CloseHandle(H);
  CloseHandle(S1);
  CloseHandle(S2);
  printf("Hit return to terminate...\n");
  getc(stdin);
  return 0;
}
```

- low resource requirements
- ROMability
- ease of use and powerful debug capabilities
- low or no run-time royalties

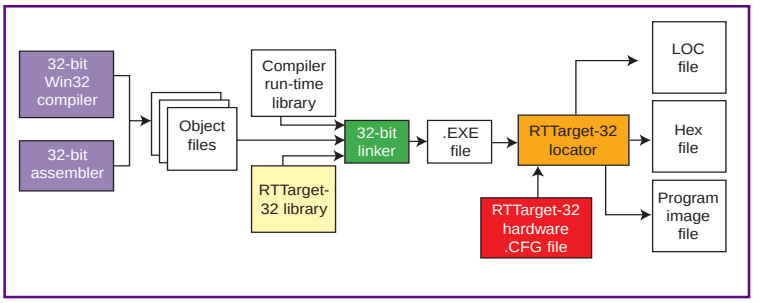


Figure 3—This diagram illustrates creating a binary program image file with standard Win32 compilers and the On Time Embedded RTOS.

TWO APPROACHES

There are two approaches to implementing a Win32-compatible RTOS. A Win32 API layer may be added as an extension to a preexisting traditional RTOS, or an RTOS may be designed from the ground up with implementation of a Win32 subset as a core design goal.

An example of the first approach is the Willows RT toolkit from Willows Software (also known as APIAccess from Award Software). The Win32 layer is supplied as a library that can be linked with an application built under one of several standard non-Win32 RTOSs, giving the application source-level compatibility with a Win32 subset.

However, binary compatibility is limited, since it is supplied through the use of virtual machine and other slow emulation techniques that simulate the native Win32 run-time environment for binary code. And, mainstream Win32 development tools can't be used with a non-Win32 RTOS.

The second approach does not suffer from the same problems. Both source and binary compatibility with desktop Win32 are possible without performance penalties. Standard Win32 development tools can be used. Because there are no extra implementation layers, applications have maximum performance and minimum size.

Commercial RTOSs using this approach include Phar Lap's Embedded Tool Suite and On Time's Win32/NT-compatible RTOS. Windows CE satisfies some of the requirements, but its nondeterministic performance, high interrupt latency, and large memory requirements (often above 2 MB) make it unsuitable for hard real-time use and small embedded systems.

Special versions of Microsoft development tools are required to create Windows CE applications, so standard desktop Win-32 tools can't be used. Finally, to boot Windows CE on standard embedded-PC hardware, MS-DOS must be present.

A WIN32-COMPATIBLE RTOS

Let's look at one RTOS designed to be Win32 compatible. The On Time Embedded RTOS runs Win32 console-mode applications (programs using `printf()`-style I/O) on any system with a '386 or higher CPU.

Its Win32 compatibility can support the complex run-time systems of current Win32 C++ compilers, including C++ language features like exception handling, global and local object construction/destruction, name spaces, and RTTI. It supports Win32 advanced features like uncommitted memory, structured exception handling, thread variables, and DLLs.

Its scalability permits booting and running a complete 32-bit application in just 16 KB of memory (see Figure 2). A supplementary native API supplies real-time functionality not available under Win32 alone.

Its foundation is the cross-development system RTTarget-32, which implements the core of the OS. It includes all the development tools needed to run 32-bit applications built with standard Win32 compilers on an embedded target.

The real-time kernel extends RTTarget-32's Win32 API support with multithread functions, semaphores, and critical sections. RTFiles-32 adds a file system, and RTIP adds a TCP/IP stack (see Figure 2).

DEVELOPMENT PROCESS

The software development process begins with RTTarget-32, and proceeds in

A Complete Example

Let's assume you have a host PC running DOS or Windows and a target PC that is supposed to boot the On Time Win32 RTOS and you want to run a test program compiled with Borland C++.

First, create the following test program in `HELLO.C`:

```
#include <stdio.h>
int main(void)
{
  printf("Hello, RTTarget-32!\n");
  return 0;
}
```

Compile and link the program with `bcc32 hello.c rtt32.lib`. To be able to run the program on the target, you must locate the program. For this purpose, a small configuration file must be created (`HELLO.CFG`):

```
// Define memory layout
Region NullPage      0    4k RAM
Region LowMem        4k 636k RAM
Region HighMem       1M   1M RAM

// Locate boot code and associated data
Locate BootCode   DISKBOOT.EXE LowMem
Locate BootData   SystemData   LowMem
Locate DiskBuffer DiskBuffer   LowMem
Locate Header     Hello        LowMem

// Locate program entities
Locate Section    CODE         HighMem
Locate Section    DATA         HighMem
Locate Stack      Stack        HighMem 16k
Locate Heap       Heap         HighMem
```

Now you can locate using `RTLoc hello`, which produces files `HELLO.RTB` (the relocated program image) and `HELLO.LOC` (a detailed map file).

To create a bootable disk, insert an empty formatted disk in drive A and type `bootdisk hello a:`. Place the disk in the drive of the target computer and reboot it. RTTarget-32's boot code initializes the PC, reads the program from the diskette, switches to 32-bit protected mode, and executes the program.

a manner that doesn't intrude into the compiler's normal development cycle. The compiler with its linker and run-time system libraries is used in the same way as for native Win32 development (see Figure 3).

Both the compiler's command-line tools and IDE can be used. A library is supplied to make RTTarget-32's Win32 and non-Win32 API available to the application. Other RTOS components are supplied as additional libraries.

RTTarget-32's locator processes a standard Win32 .EXE file and a target hardware configuration file. The various components of the .EXE are located (and possibly separated into RAM and ROM areas).

The locator includes boot code and required DLLs into the program image and generates a single application image file that can be burned into an EPROM or placed on a boot disk. During development, the application is located by downloading over a serial link under the control of the cross debugger or a download utility.

RTTarget-32 consists of:

- configurable target boot code
- bootdisk utility
- locator
- target-resident debug monitor for remote debugging
- download utility and debugger
- Win32 emulation library providing approximately 160 Win32 API functions
- serial I/O library

RTTarget-32 supports Borland C/C++, C++ Builder, and Delphi as well as Visual C++ and Watcom C/C++. The cross debugger is based on Borland's TD32 and supports source-level remote debugging of Borland, Microsoft, and Watcom programs. Integration into the Microsoft Visual Studio and Borland IDEs permits use of those environments' debuggers as well.

The debugger supports embedded-systems development (interrupt handling, port I/O, etc.). State-of-the-art data compression and caching allow fast downloading.

Another feature is the efficient use of the memory management and debugging facilities of '386 and higher CPUs. RTTarget-32 can use paging for memory protection, RAM remapping, and uncommitted memory.

When memory protection is enabled, critical system data structures such as descriptor tables are inaccessible to the application code. Writing to read-only data areas is not permitted and triggers an exception.

RAM remapping can be used to combine fragmented memory regions to larger consecutive regions (e.g., conventional and extended memory on PC-compatible systems) or consecutive virtual regions consisting of both RAM and ROM can be created.

Uncommitted memory enables an application to differentiate between reserving and using (committing) address space. Most C/C++ run-time systems rely heavily on uncommitted memory so it must be supported by the OS for efficient memory management.

Of course, access to physical memory for DMA or memory-mapped devices is supported by assigning appropriate access rights for these memory regions (either statically in the locate process or dynamically at runtime).

The CPU's debug registers, in combination with paging, enable the implementation of powerful debugging features and can remove the need to use ICEs. Any invalid memory reference triggers an exception. Within the debugger, the offending instruction is highlighted and the cause of the problem can be investigated.

Debug registers are used to implement hardware breakpoints (e.g., break on a write cycle at a specific address). Hardware breakpoints can be set in ROM or RAM and don't change the program's run-time behavior.

SCALABILITY

Because the different parts of the On Time RTOS are modules linked from a library, only those parts used by the application are included automatically. The RTTarget-32 RTOS core and all extensions are supplied as linkable libraries.

A minimal RTTarget-32 program runs in about 12 KB of ROM and 4 KB of RAM (or 16 KB of RAM in systems booted from disk). An application linked with all of the available extensions (RTKernel-32, RTFiles-32, RTIP, floating-point emulator, and DLL loader) requires about 128 KB of ROM and 128 KB of RAM.

The On Time RTOS isn't a Windows NT clone. Only a subset of NT's API is supported to keep the resource requirements small. For example, only console mode apps without a GUI are supported although graphics programming is possible using an add-on graphics library.

The On Time RTOS does not support Windows NT or Windows 95 device drivers, but real-time embedded systems often have to deal with proprietary hardware. Here, RTTarget-32's support for port I/O, interrupt handling, and access to the physical address space from within the application code makes life easier for developers.

With so many options, finding the right OS for real-time embedded systems can be challenging. By using a Win32-compatible RTOS, you can leverage the technology base of standard PC hardware and software and put it to use in the world of embedded systems. EPC

Peter Petersen has done research in the field of massively parallel real-time data acquisition at DESY (German Electron Synchrotron). He contributed to the development of an Ada compiler for a multi-processor computer system before founding On Time in 1989. You may reach him at pp@on-time.de.

Tom Schotland studied mathematics and computer science and worked as a real-time programmer in neuroscience laboratories before joining On Time in 1993. You may reach him at tom@on-time.com.

SOFTWARE
Source code for this article may be downloaded via the Circuit Cellar web site.

SOURCES

APIAccess
Award Software International
(415) 968-4433
Fax: (415) 526-2392
www.award.com

Embedded Tool Suite
Phar Lap Software
(617) 661-1510
Fax: (617) 876-2972
www.pharlap.com

On Time Embedded RTOS
On Time Software
(516) 689-6654
Fax: (516) 689-1172
www.on-time.com

Windows CE, NT, 95/98, Visual C++
Microsoft Corp.
(206) 882-8080
Fax: (206) 936-7329
www.microsoft.com

Borland C/C++, C++ Builder, Delphi
Inprise
(800) 457-9527
(408) 431-1000
Fax: (831) 431-4122
www.inprise.com

Watcom C/C++
Sybase, Inc.
(510) 922-3555
(800) 879-2273
www.powersoft.com

Real-Time PC

Ingo Cyliax

Astronomical Issues

Part 1: Introduction to Embedded Astronomy

Ingo isn't just talking about large ideas. His focus is on actual issues relating to astronomy, like GPS and digital radio astronomy. The result is a stellar project that deals with high-speed digital signal processing.

It just wouldn't be an EPC section without a space application and Fred's not launching anything this month, so I guess it's my turn. Actually, I've been interested in the field of astronomy for some time because many of the problems encountered in astronomy can be solved with computers.

In this series, I want to show you some real-time applications in astronomy. This month, I cover positioning and time references and introduce the topic of radio astronomy as well as a project I've been working on.

POSITIONING

A major challenge in astronomy is locating the things you want to look at or measure. Locating things would be easy if the earth didn't rotate around its own axis and the sun. Well, that arrangement won't change anytime soon. However, locating objects is not impossible.

To solve the rotation problem, astronomers use the idea of a celestial sphere. The celestial sphere is a fixed reference that enables you to locate stars via fixed coordinates.

These fixed coordinates are known as declination (Dec) and right ascension (RA). Dec is the angle between the object and the celestial equator. RA is the angle between the object and a fixed reference point (expressed in hours, minutes, and seconds) on the equator.

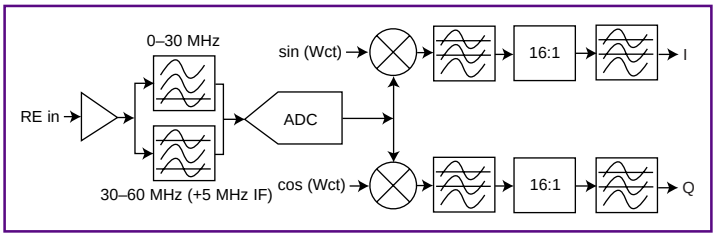


Figure 1—The components to the left of the ADC in this digital receiver are the preamplifier and the antialias or band-pass filter used to select an input frequency band. The receiver uses the cos(Wct) and sin(Wct) to tune the digital input signals and down-convert it into signals I and Q, which can be processed in software by a DSP or regular CPU.

The celestial sphere has a north pole, which is on the earth's axis of rotation. The plane of the celestial equator is inclined (by approximately 23.5°) compared to the plane of Earth's orbit around the sun.

Even though stars appear to always have the same Dec above the celestial equator, the sun does not. It varies from 23.5° above the celestial equator to 23.5° below. These points, called the solstices, take place around June 21 (northern) and December 22 (southern), which are the longest and shortest days in the respective hemispheres.

The times when the sun is directly on the celestial equator are called equinoxes. There are vernal (March 21) and autumnal (September 23) equinoxes. The reference point on the celestial equator for our RA is the vernal equinox.

But, because the earth's orbit precesses slightly (about

50″ per year), there's a slight discrepancy. So, star catalogs are calibrated to specific years, usually every 50 years. If you use a star catalog that's calibrated for the year 2000, you have to adjust the true vernal equinox by subtracting 50″ from the position.

The easiest way to use this star catalog with a telescope is to use an equatorial mount. An equatorial mount is aligned with the polar axis of earth's rotation. The declination angle is fixed, and once aligned, it doesn't need to be changed.

Next, calibrate the RA by finding the sidereal time. The local sidereal time is defined as the position on the celestial equator overhead at the observer's location. At the vernal equinox, the local sidereal time is zero.

Photo 1—Here's the sky view screen from xephem. xephem not only gives you the graphical view of the sky but can also calculate various astronomical data for objects in its database.

Like terrestrial time, sidereal time is measured in hours, minutes, and seconds. But, sidereal time is slightly faster than terrestrial time. One sidereal day (i.e., 24 sidereal hours) is equal to 23:56:04 in terrestrial time. That works out to one extra sidereal day every year.

Think of it this way. If the earth didn't rotate with respect to the sun (i.e., the same side of earth always faced the sun), there would be one sidereal day per year because the earth would rotate one time with respect to the celestial sphere, which is fixed at the vernal equinox.

Because the earth rotates about 365 times with respect to the sun, we have to add an extra sidereal day. That means the sidereal day is a bit shorter than the terrestrial day and explains why the stars rise about 4 min. earlier each day.

To compute the sidereal time, you need to know when the RA is at a specific terrestrial time. For example, the January issue of *Sky & Telescope* lists the RA of the sun on February 1 to be 20:56:30 at midday when the sun is at its zenith.

Let's calculate the current local sidereal time. I'm writing this at 18:48:00 on January 4th, local time. There's a difference of 27 days and 17:12:00 between right now and the time when I'll know the exact sidereal time (February 1).

When you're working with time, it's best to convert everything to seconds. The RA then becomes 75,390 s and the time difference is 2,332,800 + 61,820 s. Convert the terrestrial time difference to sidereal time by multiplying by 1.003 and you get (2,332,800 + 61,820) × 1.003 = 2401803.86, which you subtract from the RA. Converting that (–2326413.86) back to standard time format, you get 1:46:01, which is the local sidereal time.

Luckily, other people have figured all this stuff out and written the software. Also, the reference, or epoch, is taken from an astronomical almanac that's more precise than the one I used. xephem is my choice because it runs under Linux and X.

This package computes and displays the location of stars and nebulae as well as planets and their moons (see Photo 1). Its catalog can be extended and updated using databases downloaded from the 'Net. It can also be used to drive a telescope.

One feature I like is the ability to compute satellite locations based on orbital data that you can download from the 'Net. You can be like NASA and plot the track of shuttle missions in real time, assuming you get the correct orbital data.

Plugging in my local sidereal time and my longitude, I computed the local sidereal time to be 0:59:55. At first I thought I made a significant mistake in my earlier calculation, but as it turns out, I didn't. My longitude is roughly 89° W, which implies that a true local time has an offset of 5:55:59 from the meridian, while the timezone I live in has a 5:00:00 offset.

This offset means that the wall-clock time will be almost 56 min. later than the sidereal time I calculated, assuming that the sun transits (passes overhead) at exactly 12:00. If I subtract the offset (0:55:59) from my calculated time, I get 0:50:02, which is much closer to the correct time. Because xephem uses a more accurate epoch to compute the sidereal time, I'll trust the tool rather than my hand calculations.

The older version of this software is called ephem. ephem doesn't have an X Window GUI so it can be embedded in a telescope controller. It's fairly small and doesn't require many OS services to run. It can even be built to run under DOS.

Now that you know how to find things in the sky with software, or by hand if you were stranded on an island, let's take a survey to find out how much accuracy is involved.

The moon is 0.5°, or 1800″, in diameter. To find out how long it will take to transit, simply divide the size by the earth's rotation. This turns out to be:

So, if your time reference is off by more than a minute, the telescope will miss the moon—the largest object in the sky.

Jupiter, on the other hand, is the largest planet and it is about 30″ in size. The transit time for Jupiter works out to:

Even more demanding. That means your telescope is going to miss Jupiter unless you can compute the RA to within 0.9 s.

If you want to track an object smoothly, you have to continuously recompute its position at the resolving power of the telescope. A terrestrial optical telescope has a maximum resolving power of ~0.5″, so you want a positioner that can position at that accuracy (or less, if your telescope has less resolving power).

Now you're clearly in the realm of real-time control.

TIME REFERENCE

Time is an important variable when locating objects. You need an accurate time reference to compute the sidereal time, which is necessary to position the telescope and track the object.

In the example, the 32 ms required to track a high-resolution telescope is probably on the upper end that is necessary for positioning. A 0.5″ telescope is expensive, and you'd need perfect viewing conditions to take advantage of the resolution.

But, accurate time is also needed to correlate findings between different observation sites. Several experiments in astronomy depend on time. One example is occultation measurements. An occultation is when a planetary object or the moon moves in front of a star.

Astronomers measure occultations to calibrate orbital components of planets, the moon, and the earth. To measure an occultation, record when the star disappears and reappears. It's a discrete event in which the star blinks out and then blinks back on.

When more than one observer records these events accurately, the results are precise. You need to coordinate time between observers. Most astronomers can't afford an atomic clock, so they use radio time broadcasts to synchronize their clocks.

In North America, this can be done with WWV(B), which is a time signal broadcast by shortwave radio on standard frequencies. Another time standard is broadcast by CHU. Other parts of the world have similar time services.

WWV time signals can be decoded and used to calibrate clocks. These clocks are commonly referred to as radio clocks. Some are available with computer interfaces so they can be used to synchronize computers and networks.

Today, you can use a global positioning system (GPS) to synchronize your clocks. Most GPS receivers have serial (RS-232)-based computer interfaces that augment existing navigation equipment in boats and airplanes. GPS receivers have also become popular for time synchronization of computers and networks.

All you need is a GPS receiver with a NMEA-0183–compatible interface and the software to read it. NMEA-formatted messages are easy to parse and look a lot like Listing 1. The GPS receiver sends these messages over the serial line and the interface software on the computer reads them, parses them, and adjusts the clock to the time indicated in the messages.

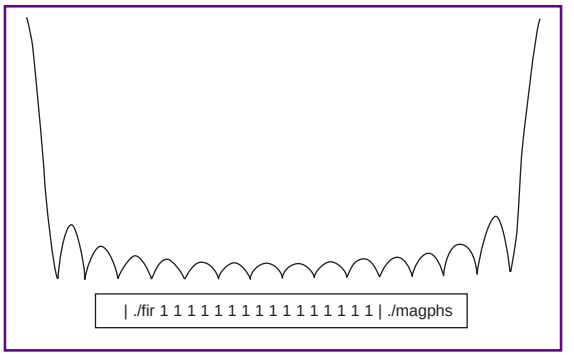


Figure 2—This graph shows the spectral response of a 16:1 decimation filter. The filter is crude but can be implemented efficiently. A sharper filter can be implemented at a lower sampling rate if necessary.

One simple program that accomplishes this is gps-linux. This program runs under Linux and reads and converts the messages into a format that the date program under Linux/Unix can use.

Using just the NMEA interface gives a certainty of 10–100 ms from the GPS receiver. This range is because of the unpredictable latency from the time the message is sent until the time it is received and decoded. You can enhance the accuracy by timestamping the messages when they are received. Timestamping eliminates some of the uncertainty in the buffering.

Professional-quality GPS receivers that can be used for time synchronization output a pulse per second (PPS) signal that is synchronized to the internal clock of the GPS receiver, and thus to the GPS network. Using the PPS signal to synchronize the computer clock permits time-synchronization accuracy of better than 10 µs.

There are various schemes for interfacing the PPS signals with modem control signals via the RS-232 interface. The data from the NMEA message coarsely synchronizes the clock, and the PPS signal precisely times elapsed seconds. In a way, the computer is used to phase-lock the computers' idea of time to the GPS network time.

xntp, an implementation of the network time protocol (NTP) system, has PPS and non-PPS–based GPS interface routines to synchronize the network time to a GPS receiver.

For astronomy purposes, location information is also useful. After all, longitude information helps you precisely calculate the sidereal time based on the UTC time and the offset from the meridian. The latitude information can be used to indicate the elevation of the polar axis needed for an equatorial mount.

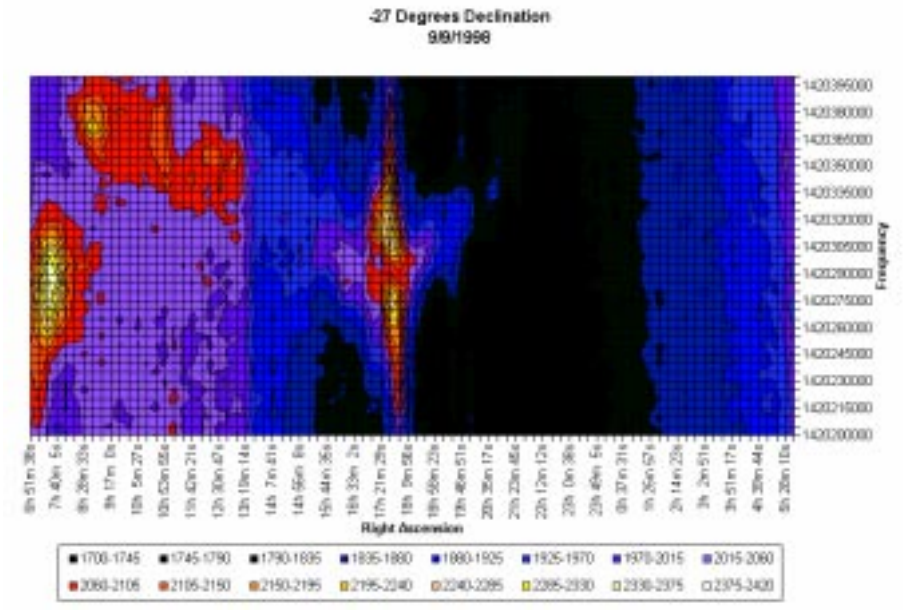


Photo 2—This radio map of the sky was created by Daniel Boyd Fox (a local amateur astronomer) with a 16-foot dish antenna and an ICOM R7000 communication receiver. He wrote a program (SETIFOX) to calculate the spectral data. The emission in the middle (around RA 18h) is caused by our galactic center, and the dentrils above and below are the spectra of the Doppler-shifted hydrogen emission of stuff moving away (lower frequency) and approaching (higher frequency).

RADIO ASTRONOMY

One area of astronomy that fascinates me is radio astronomy. Most people automatically think of big radio dishes and exotic equipment, but you can do some radio astronomy on a smaller scale.

In particular, you can use satellite TV dishes along with general-purpose scanners and receivers to make hydrogen-emission radio maps of the sky. Photo 2 shows a map that was done by a local amateur astronomer with a 16-foot dish antenna and an ICOM R7000 communication receiver. The different colors indicate power. The vertical axis is the spectrum about the hydrogen emission line, and the horizontal axis is the RA as scanned by the earth's rotation.

Also, Jupiter is a strong radio source below 38 MHz. You can receive Jupiter emissions with a shortwave receiver and an antenna with selectivity at the specific frequency.

Simply point the antenna at roughly the declination you expect to find Jupiter (approximately 0°57′). As earth sweeps close to the RA of Jupiter (0:02:30), the emissions from Jupiter should maximize. The phenomenon should sound like a whooshing sound on the radio and isn't present all the time. The radio emissions are caused by the dynamic interactions of Jupiter's magnetosphere and its moon Io.

If radio emissions from Jupiter don't interest you, there's another astronomical phenomenon that can be measured by the shortwave band and other frequencies. As meteors travel through the atmosphere, they leave an ionized trail that radio signals bounce off of. Radio-station signals that can't normally be picked up will bounce off the meteor trails and be heard.

If the setup is right, you can count how many meteors enter the atmosphere. Even small meteors that can't be seen will bounce back radio signals.

What does this have to do with real-time PC? Well, I've always wanted to do a project with high-speed DSP, but I didn't want to build a radar or sonar system. Because I'm interested in radio astronomy, I thought this project would be interesting and challenging. Also, the timing is right.

Building a totally digital shortwave receiver with a tuning range of DC–25 MHz requires an ADC that samples at 50 MS/s. Although the flash ADCs used in video and other signal-acquisition applications can handle the speed, they only do it at 8 bits, which doesn't provide enough dynamic response to do direct sampling.

But, both Analog Devices and Burr-Brown recently introduced 12-bit ADCs that can sample at over 50 MS/s. They also have the analog input bandwidth capable of dealing with radio applications.

Let's see what a digital radio system would have to look like. For this example, I used the AM receiver shown in Figure 1.

An all-digital radio is a system where the digitization is done before the first mixing stage. The only analog components are the preamplifier (if necessary) and the anti-aliasing filter.

If the ADC is used to mix the signal down, the antialiasing filter is a band-pass filter. More precisely, it's a low-pass filter to select the baseband. ADCs that are suited for digital radio applications typically have a wide analog bandwidth so signals at 4× the sampling rate can be mixed down.

Once the RF signal is digitized, it's channelized by selecting a subband of the signal and reducing the sampling rate

Listing 1—In the data from an NMEA output of a GPS receiver, everything is separated by commas. NMEA defines the format and components of each line (i.e., sentences). The current time (054449.813) can be found in several different sentences in different positions. gps-time (one of the programs I mentioned) looks for a sentence that starts with $GPGGA.

```
$GPGLL,3909.360,N,08625.143,W,054449.813,A*2D
$GPGGA,054449.81,3909.360,N,08625.143,W,1,04,2.0,00033,M,,,*3D
$GPRMB,A,0.01,L,SIM001,SIM002,3911.410,N,08625.153,W,002.1,000.,021.7,V*10
$GPRMC,054449.81,A,3909.360,N,08625.143,W,21.7,001.3,050199,02.,W*60
$GPAPB,A,A,0.0,L,N,V,V,2.4,M,SIM002,2.2,M,0.9,M*61
$GPGSA,A,3,01,02,03,04,,,,,,,,,2.0,2.0,2.0*34
$GPGSV,3,1,10,10,78,049,,24,51,058,,06,42,311,,13,36,084,*7E
$GPGSV,3,2,10,30,33,259,,05,22,214,,04,14,083,,26,12,172,*7F
$GPGSV,3,3,10,17,06,274,,18,03,032,,,,,,,,*72
```

for further processing. Theoretically, you can filter the digital signal to the bandwidth you want to look at.

In the AM receiver example, that means filtering a 50-MS/s signal directly down to a 6-kS/s bandpass filter—computationally expensive and not exactly practical. Channelizing lets you pick larger blocks of bandwidth to reduce the sampling rate so you can do the final filtering or signal processing.

Reducing the signal rate, or decimation, is like throwing away the samples you don't need or subsampling the input samples. In itself, this task would cause signals to be aliased so you must apply a low-pass filter before you toss out the samples.

A simple way to handle this process is to use a barn-door filter. This crude digital filter is inexpensive to implement, especially when used as a decimation filter. A 16:1 decimation filter reduces the sampling rate by 16 and only lets one-sixteenth of the spectrum pass through. Figure 2 shows the spectrum of a 16:1 decimation process.

Because the decimation process is a low-pass filter, you may wonder how to get at the frequencies outside the low band. Just use the same idea an analog radio uses. A mixer is used to multiply the input signal with a carrier. The mixer's output is the sum and the difference of the carrier and the signal of interest.

Adjust the carrier so the difference shifts the signal of interest into the passband of the decimation filter and you're in business. You can tune any signal in the input.

To make the radio general, mix the input signal with both the sin() and the cos() of the carrier to produce two products (I [in-phase] and Q [quadrature] signal) that are then filtered separately.

With an AM receiver, you're interested in the magnitude of the signal, which is:

But, with an FM receiver, the key is the phase difference of the signal or:

Other modulation techniques use I and Q to demodulate into the final signal as needed. Once the input signals are mixed into the I and Q signal and decimated, the signal is low-pass filtered until you have a 6-kS/s signal that carries the final 3-kHz bandwidth AM signal.

You may wonder if multiplying a synthesized sin() and cos() can be done more cheaply at the sampling rate than by band-pass filtering the signal first. I'll cover the topic next month when I describe how these signal-processing components can be realized in an FPGA. RPC.EPC

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.*

SOFTWARE

xephem—ftp://iraf.noao.edu/contrib/xephem/xephem_3.1/xephem-3.1.tar.g
gps-time—http://mix.hive.no/~tommy/linux/gps-time/gps-time.tg
xntp—ftp://louie.udel.edu/pub/ntp/

REFERENCES

R. Beebe, *Jupiter: The Giant Planet*, Smithsonian Institution Press, Washington, DC, 1997.
B.F. Burke and F. Graham-Smith, *Introduction to Radio Astronomy*, Cambridge University Press, New York, NY, 1997.
G. North, *Advanced Amateur Astronomy*, Cambridge University Press, New York, NY, 1997.
J.B. Sidgwick, *Amateur Astronomer's Handbook*, Dover Publications, New York, NY, 1971.
R.R. Bate, D.D. Mueller, and J.E. White, *Fundamentals of Astrodynamics*, Dover Publications, New York, NY, 1971.
Sky Publishing Corp., *Sky & Telescope*, Cambridge, MA.

Applied PCs

Fred Eady

ICE on Tap

Part 2: Emulating over Ethernet

It doesn't matter whether you're a seasoned design engineer or brand new to embedded systems. Fred presents the bare bones of the SuperTAP setup, so even if you've never used an emulator before, you will be ready to now!

It would be great if I could start up the SuperTap and run a demo simulation right here on the printed page. It would also be great if everyone reading this column knew embedded systems in and out. Unfortunately, that isn't the way it is. Besides, a two-word column wouldn't go over real well.

There are plenty of you who I consider to be top-notch engineers. Your discipline may be chemicals, electricity, or even information technology. The point is, there are professionals reading this column who aren't necessarily "embedded."

With that thought, I think it's good to put some emphasis on just what the SuperTap does and how to get it to the point of actually "doing." Describing the running of emulation is useless if the reader has never used an emulator. So, let's open the SuperTap Emulator installation guide to page 4-1 and put SuperTap on an Ethernet network.

SUPERTAP ON THE NETWORK

The first order of business after the network hardware (a 3Com EtherLink III Ethernet interface card) is installed in the host PC is to define all of the players. This step is done via the host database, TCP/IP definitions, and our imaginations. Setting up the 3Com card was a snap. I'm hosting Windows 95 on the SuperTap host PC, and all the necessary drivers for the 3Com card were there.

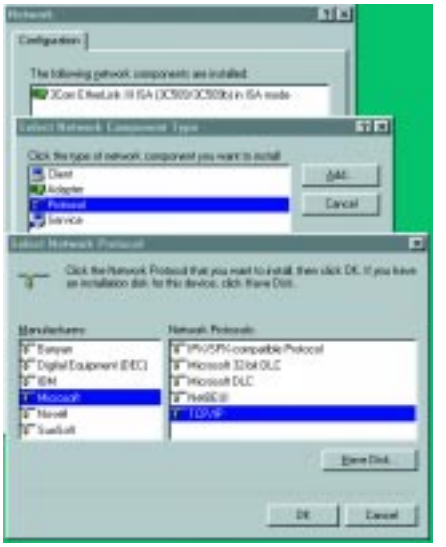


Photo 1—Every time I do this, I always seem to forget something. Not this time!

I tested initial Ethernet connectivity by using NetBEUI to communicate with the computer I'm using to write this article. Of course, I'll be using TCP/IP. For the networking folks out there, Photo 1 is a familiar sight.

Photo 2 is where the rubber meets the road as far as the host PC's TCP/IP is concerned. This is a private network so we can use any address. To avoid the problem of choosing addresses that have special uses, such as the loopback address, I'll play it safe and use the recommended addresses and subnet masks.

This is all jolly good stuff that's pure Windows 95 and to most of you, routine. Routine isn't always bad. Remember that the real gold is getting the equipment on the network.

The next step towards the end of the rainbow is to set up the hosts database. This relatively simple maneuver involves

Photo 2—It may be routine, but it had better be correct.

finding the hosts file and adding our player's definitions. Under Windows 95, the hosts file is found in the Windows directory. It's a little more hidden in NT but not hard to find. The network is defined in Photo 3.

Here's where we take a slightly different path. The initial setup of SuperTap suggests using a serial connection. If that's too mundane for you, the next suggestion is Ethernet via BNC or thin-wire 10Base2. Guess what we're gonna use? Ethernet, of course, but with twisted-pair 10BaseT.

The only downside to this method is that if you don't own an Ethernet hub, this method is a more pricey way to go. The upside is that the cable is easier to work with and there are no 50-Ω terminators to deal with. Heck, this is the Circuit Cellar Florida Room. How many hubs do we need anyway? Looks like 10BaseT to me.

Another good thing about the Applied Microsystems development kit is that everything you need to get this puppy going is in the box. If you weren't around last time when I unpacked SuperTap, I should mention that there were a couple of media access units (MAUs) that didn't appear to be necessary.

Well, here we are and the 10BaseT MAU is now connected to the Ethernet communications Adapter's Ethernet port and an RJ-45 twisted pair cable is leaving the port and terminating at a port on the Ethernet hub.

I have the SuperTap and Ethernet communications adapter HSS (high-speed serial) port tied together via an RJ-45 cable. A 10BaseT MAU is connected to the 15-pin AUI (10Base5) port on the Ethernet communications adapter, and a standard RJ-45 twisted pair Ethernet cable is connecting the MAU and a standard dumb Ethernet hub. But, installation of the Ethernet adapter is not quite complete.

There's a matter of downloading some network parameters to the Ethernet adapter's flash. To accomplish this task, we must power up the SuperTap/Ethernet adapter combination. This is done with a well-thought-out DIN cable arrangement. The cable is foolproof—it can only be plugged into the power-supply brick one way.

The longer of the two male cables plugs into the DIN connector on the Ethernet adapter and the short DIN is connected to the power receptacle of the SuperTap. The reason for this arrangement is so the Ethernet adapter lead is power-limited.

Before we power up the SuperTap, let's make the physical and logical connections to the host PC that will enable us to communicate with the Ethernet communications adapter.

First, connect a serial cable between the host PC's async port and the serial port on the Ethernet communications adapter. Although this procedure can be performed with any serial async emitting device, we'll use Bill's Win95 HyperTerminal.

After setting up for 9600 bps, 8N1, and hardware handshaking, I set the Ethernet communications adapter rotary switch to 0. Photo 4 represents another milestone in bringing SuperTap to a productive state.

Let's take a closer look at the information contained in the HyperTerminal window in Photo 4, starting at the Ethernet address line. The Ethernet address consists of six hexadecimal numbers each separated by a colon. Just so happens that this hexadecimal address matches the Ethernet communications adapter address printed on a label on the 15-pin AUI connector.

Photo 3—Don't let the example lead you astray. The addresses must start in column 1.

Note too that the static IP address selected line looks a lot like the IP address we entered in the hosts database file: the IP address for the Ethernet communications adapter. A close look at the static netmask selected entry reveals a match of the subnet mask value we input in Photo 2.

For the TCP/IP-challenged out there, the subnet mask values must match between the communicating parties. Conveniently enough, our netmask defines a Class B network.

The broadcast-enabled line signifies that the Ethernet communications adapter won't ignore network broadcast of routing information. This has to do with routing tables and the magic that goes on behind them. Our network uses static values so this parameter won't help or hurt us.

"Route daemon started" tells us that the default configuration that automatically runs the daemon on Ethernet communications adapter startup is active and the daemon was started. The last line is pretty

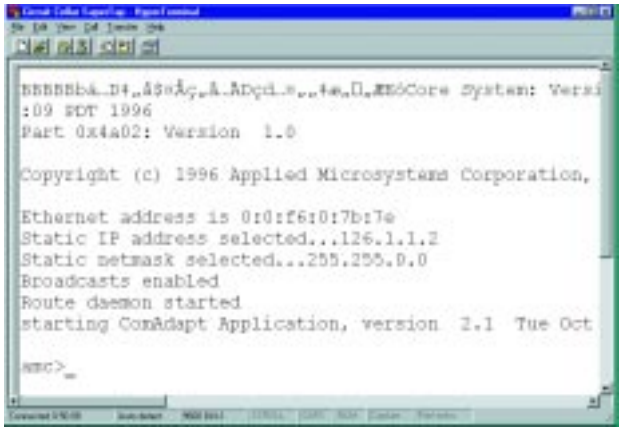


Photo 4—It's always nice to see stuff come up, but it's doubly nice when it comes up with the data you entered!

straightforward—the ComAdapt App started.

In a nutshell, our network is small and doesn't access hosts across network gateways and routers. This also implies that TCP/IP reverse address resolution protocol (RARP) or boot resolution protocol (BOOTP) won't be used in our self-contained network.

Here's what's going on in our network world. We entered the IP address manually using HyperTerminal communicating with the Ethernet communications adapter via the Ethernet adapter's serial port. The IP information that we entered is associated with the Ethernet address (the six hex digits separated by colons) that's stored in the Ethernet communications adapter's nonvolatile RAM.

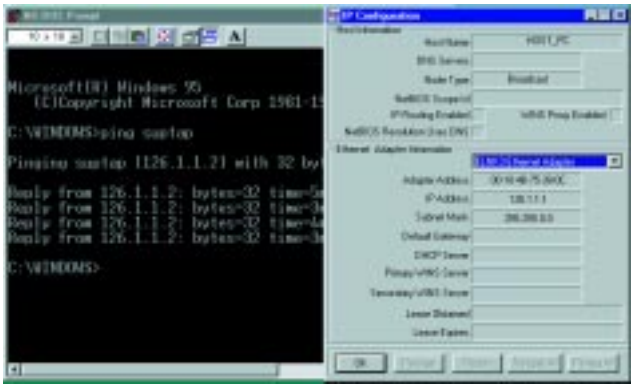


Photo 5—Did you know that PING actually stands for Packet InterNet Groper?

If our SuperTap was on a larger network, any TCP/IP address resolution request (ARP) transmitted by the network OS would get a response from our Ethernet communications adapter. The idea is to use the IP address to get the hardware address of the addressed device.

Those requests don't exist, but we still need to IP-identify the Ethernet communications adapter. The static IP address and static netmask you see reported by the Ethernet communications adapter in Photo 4 were entered using `netparam`.

As you've probably ascertained, the Ethernet communications adapter is loaded with firmware that enables communication with the outside world using asynchronous serial protocols. It also contains code that starts the daemon and lets the Ethernet communications adapter play on larger more sophisticated networks.

I won't get any deeper into networking, but before we move on, here's the command syntax for storing the static IP address and static netmask on the adapter's flash:

```
netparam -static_ip_address
126.1.1.2 -resolve_ip_address
static netparam -static_netmask
255.255.0.0 -resolve_netmask
static
```

Remember when I used NetBEUI to test the Ethernet connection between the soldier and host_pc machines? Photo 5 is the result of pinging SuperTap from the host PC side.

IT'S ALIVE

Now that we can access SuperTap from anywhere in the Florida Room (or the world, for that matter), let's load up the debugger, flip the appropriate switches, and look at some of the code in action.

Looking back to Part 1, I recall unpacking an embedded PC with no processor and no instructions. I scoffed at this being just another embedded platform that surely

Photo 6—I've never been so happy to see so many buttons!

I could figure out. Obviously, it was to be the target for the SuperTap, but the only clue was that the SuperTap plugged into the empty processor socket.

Since then, I've read a few more pages of the documentation. Seems that this insignificant target board is the tool I'll use to open your eyes to SuperTap operation.

After looking over the Adastra embedded PC target, I decided it was time to act. After all, how much smoke and bright light could I possibly generate if something went wrong? With that mindset, I went off in search of a power supply.

It's hard to believe that Applied Microsystems didn't include a power supply. They certainly thought of everything else. At this point, I thought you might want to know what the lashup looked like before I added the Adastra.

The Ethernet communications adapter was alive with flashing lights indicating that TCP/IP was busy passing data between the Ethernet adapter and the host PC. The 10BaseT MAU is all green and go. (Did I say "go"? That's what happens when you live near the Cape.)

SuperTap's internal cooling fan is running, and the power indicator is green. I'm a little concerned that the run/pause LED is red, but everything else seems to be cooking right along.

OK, it's time. Off with all power to the SuperTap and Ethernet communications adapter. I carefully mate an extender for the '486 socket to the SuperTap and insert the same in the empty Adastra '486 socket. SuperTap and the Adastra are mated and awaiting power.

I'm thinking that I need to finish the article with working hardware and all the worst scenarios are running through my mind, but here goes. Power to the SuperTap and Ethernet communications adapter.

No problems yet. No blinding light or smoke. Things are looking good.

Now, power to the Adastra. The Adastra power-supply fan is running and the power didn't shut down. Things are looking better. All that stands between us and a functional emulation system is a few keystrokes on the CAD-UL XDB startup screen to define working directories, boot options, and connection type.

Behold the portal to the promised land in Photo 6. The first half of the command uses the XDB startup screen parms to locate working directories, determine CPU type, and figure out the TCP/IP stuff. Listing 1 is the `.XDB` file that sets up the memory map and user-defined buttons (download, start, reload, and exit) that are directly above the command screen.

As you can see, we can talk all day just about the buttons across the top of the command XDB window. I went ahead and downloaded the `CDEMON` hex file and started it.

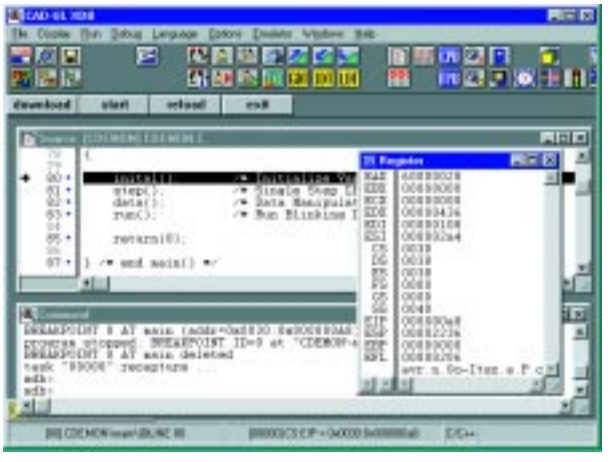


Photo 7—Beauty in the hexadecimal eye of the beholder.

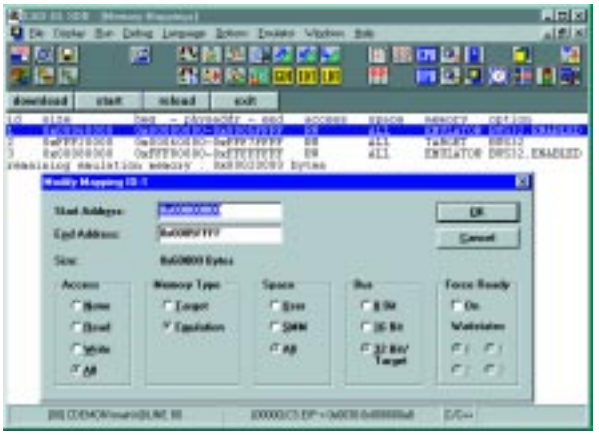


Photo 8—Memory where you need it, when you need it, and how you need it.

Photo 7 shows the C source code in the upper window and the command status in the lower window. I pulled up a CPU register window to give you a taste of how that looks on XDB. We can't visit every button, so I'll show you the really juicy ones.

First, let's take a closer look at memory mapping. Memory mapping enables you to allocate overlay memory in the areas that concern your programs. Overlay memory is a chunk of memory (usually RAM) that can be placed in various address spaces.

SuperTap comes with 1 MB of overlay standard and can be upgraded to multiple megabytes of overlay. A typical memory map is shown in Photo 8. I selected the modify option to illustrate just how overlay memory can be used.

Starting from the left, note the access frame. Selecting an option within this frame sets the user access privileges for this block of overlay memory. If your program should not access this area of memory, select None. Using this option, SuperTap is designed to alert you if this area is stepped on.

By selecting Read, you can also use this area to emulate a ROM data area. Again, if any program writes to a read-only area, SuperTap informs you of the transgression.

The remaining frames determine where the memory resides (Memory Type), who can use the memory area (Space), how wide the bus will be (Bus), and whether or not to force a Ready (Force Ready).

SuperTap does all the things a good emulator does. Memory dumps, register displays, breakpoints, and traces are all standard.

One feature that caught my eye was the manner in which SuperTap described and allowed the manipulation of descriptor tables. Photo 9 is a typical GDT display.

Notice that all of the bits are broken out for easy reading. When a bit is selected, its function and state are explained in plain English. Everything about that descriptor is right there, and as you can see, the segment can be modified from here, too.

Listing 1–Sorta C, sorta DOS batch file, sorta works.

```
stop
dele map /all
set map /access=all /space=all /type=emulator \
 0xfff80000 until 0xffffffff
set map /access=all /space=all /type=emulator \
 0 until 0x40000
set option /asm = on
set control /display=0
define button download "\
load /hex of cdemon.hx;\
load /debug=0/segment/noload of cdemon.bd;\n"
define button start "\
restart;\
set break at main hard;run;dele break at main;set task /step;\n"
define button reload "\
load /hex of cdemon.hx;\
restart;\
set break at main hard;run;dele break at main;set task /step;\n"
define button exit "\
exit\n"
```

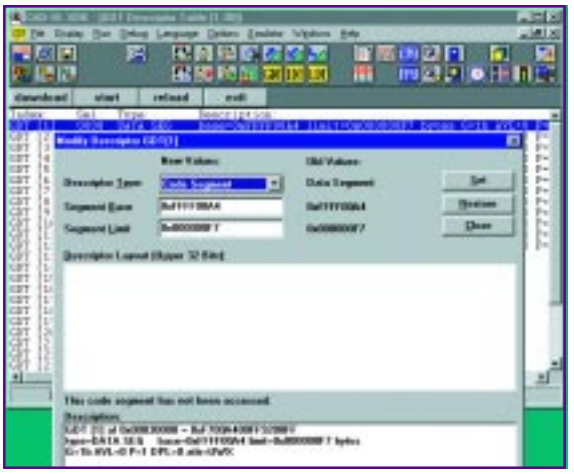


Photo 9—You can trash the manual. All of your bits are right here.

TAPPING OUT

You've just experienced emulation through emulation. Via the printed page, you've taken a SuperTap emulator from box to production.

SuperTap is by no means perfect. The folks at Applied Microsystems know this and have stepped forward with the shortcomings.

Some of the bugs relate directly to how the '486 works, but none are showstoppers. For instance, attempting to force wait states on overlay memory mapped to active target memory may cause unpredictable results. I agree with that. To use the old adage, if it hurts, don't do it.

Most of the other bugs are simple omissions. Things like the Save Settings command not saving the state of open windows. As I said, not showstoppers, but bugs nevertheless.

Otherwise, the SuperTap is a great learning tool. In fact, while I was reading through the documentation, I found the answer to an Intel hex question I'd been researching on the Internet for days with no avail.

Photo 10—Strap on a booster or two, add an external tank....

My kit is obviously a marketing demo setup, but it has enough real stuff to make it worthwhile. Whether you're a seasoned embedded engineer or one of the professionals I spoke of earlier, you need a tool like this one.

A little bit of everything that's important to embedded is here. There's some networking, some Windows NT/95, some honest hardware, and plenty of hexadecimal.

I can't show you the pretty LED displays on the Ethernet communications adapter or the rolling LED display on the Adastra board, but I'll leave you with Photo 10—SuperTap in control. It may have come from a big box, but it's not complicated. It's embedded. APC.EPC

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

SOURCES

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

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

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

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

DEPARTMENTS

MICRO SERIES

Joe DiBartolomeo

TPU

Scheduler and Microcoding

Part 4 of 4

In this final installment of the TPU series, Joe looks in detail at the scheduler and the CHAN sub-instruction. Once everything is up and running, you'll find that the TPU enables the CPU to use its time more effectively.

In the first half of this series, I looked at the most common implementation of the basic timer/counter function found on most microprocessors.

Next, I introduced the TPU, a timing unit that's completely different from the common implementation. Remember, the TPU is a semiautonomous microengine that can run preprogrammed timing/counting functions out of CPU ROM or run user microcode out of onboard RAM or flash memory.

Last month, I looked at programming the TPU using microcode. Because it's important to understand the hardware when programming in microcode, I started by looking at the TPU's hardware structure. The main hardware block is the execution unit (basically, a microengine), which runs the microcode.

The execution unit is a shared resource among the 16 TPU channels on the '68332. Sharing the execution unit means that only one channel's microcode can be run at a time. Access

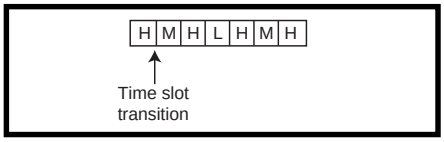


Figure 1—*The scheduler's round-robin priority scheme ensures high-priority channels get the majority of execution time and lower-priority channels don't get locked out.*

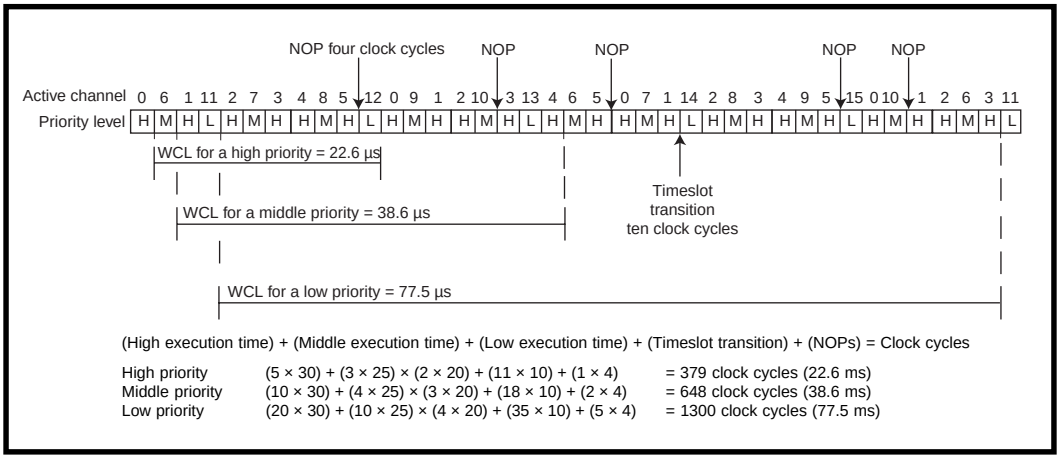


Figure 2—*In this example of latency time for high-, middle-, and low-priority channels, note that because each channel is constantly requesting service, the latency time is equal to the worst-case latency time (WCL).*

to the execution unit is determined by the scheduler, which is what I want to look at this time.

When I presented the microcode instruction formats and subinstruction set last month, I purposely left out one subinstruction—CHAN—because I wanted to cover the scheduler first. So, in the final installment of this series, I want to discuss CHAN and present an example of TPU microcoding. Decoding a matrix keypad is a good example of how the TPU can save both CPU time and money.

SCHEDULER

A TPU channel gains access to the execution unit by issuing a service request. Once the scheduler grants access, the execution unit runs that channel's microcode.

Microcode for all TPU channels is made up of states and is organized as a function. There are 16 possible states per function. Which channel state is executed depends on which of the 16 possible entry points causes a service request. Later, I'll look at entry points and event service requests.

Each TPU channel's microcode is completely independent of all other channels. The only constraint is program memory size. The address of the state to be run is obtained from the entry-point segment and is loaded into the execution unit. When the state ends, the scheduler determines access to the execution unit.

The scheduler doesn't arbitrate the individual channel events (i.e., the states). It only arbitrates between channels. A schematic of the scheduler and the 16 TPU channels was presented in Part 1.

The scheduler places the channel service requests into a round-robin priority scheme which ensures that high-priority channels get the majority of the execution unit's time and lower priority channels don't get locked out.

In the priority scheme illustrated in Figure 1, note that there are seven time-slot partitions that reserve four timeslots for high-priority channels, two for middle-priority, and one for low-priority channels. Also, priority passing occurs when a timeslot for a priority level occurs and no channel with that priority level is requesting service. This arrangement ensures no idle time as well as orderly passing to the next priority level.

REQUEST LATENCY

In a round-robin priority scheme, the length of time between a service request and the granting of service is referred to as latency time. Latency time depends on how many TPU channels are active and how long each channel's service takes. So, predicting latency time is quite difficult, if not impossible.

In most systems (particularly real-time systems), this lack of predictability is unacceptable. However, there's a way to determine the worst-case latency of any channel. It's a simple formula based on the number of active TPU channels, the time each channel function requires, and the channel's priority levels.

For example, let's look at the worst-case latency of a high-, middle-, and low-priority channel. In this example, all channels are continuously requesting service. As you can see in Figure 2, TPU channels 0–5 are high priority, 6–10 are middle priority, and 11–15 are low priority.

	Entry points	Host request (HSR)	Link request (LSR)	Match/transition service request (M/TSR)	Pin state	Channel flag 0
Host control states	0	01	X	X	0	X
	1	01	X	X	1	X
	2	10	X	X	X	X
	3	11	X	X	X	X
Operational states	4	00	0	1	0	0
	5	00	0	1	0	1
	6	00	0	1	1	0
	7	00	0	1	1	1
	8	00	1	0	0	0
	9	00	1	0	0	1
	10	00	1	0	1	0
	11	00	1	0	1	1
	12	00	1	1	0	0
	13	00	1	1	0	1
	14	00	1	1	1	0
	15	00	1	1	1	1

Table 1—*The entry points for TPU states include four states that are entered via CPU control and 12 states that are entered via pin action.*

To simplify this example, each priority class requires the same amount of execution time. High-priority channels require 30 clock cycles and middle- and low-priority channels require 25 and 20 clock cycles, respectively.

It takes 10 clock cycles for a time-slot's transition to be completed. Also, a four clock-cycle NOP executes when a priority level's round robin ends.

There's also a RAM collision rate (RCR) to be considered. The RCR is a two-cycle wait that occurs when the CPU and the TPU both access the TPU RAM at the same time and a RAM collision occurs. I've buried the RCR in the cycle times of each priority level.

Assuming a clock of 59.6 ns, you can determine from Figure 2 that the latency time would be 22.6 µs for a high-priority channel, 38.6 µs for a middle-priority channel, and 77.5 µs for a low-priority channel. In this example, all channels are constantly requesting service so the latency time is the worst case.

When the system is running in emulation mode, the priority for each channel is set the same way as it was when the system was running the preprogrammed functions (i.e., by setting the respective two bits of the channel priority registers CPR0 and CPR1).

Table 2—*As you can see from the syntax, with four valid formats, CHAN is the most complex subinstruction.*

Format 2
chan {flags} {,pac} {,psc} {write_mer}
{,neg_TDL} {,neg_MER} {neg_LSL} {,cir}
Format 3
chan {flags} {,tbs} |{ {,neg_TDL}
{,neg_MER} } {{,pac} {,psc}} | {, config:=p}
{,enable_mtsr|disable_mtsr}
Format 4
chan {flags} {,cir}
Format 5
chan {flags} {,cir}, {match_gte|match_equal}

{flags} is one of the following keywords:
set flag0, set flag1, set flag2, clear flag0,
clear flag1, clear flag2

PROGRAM STORAGE AND ENTRY POINTS

The user microcode resides in onboard RAM or flash memory, depending on the microprocessor. In the '68332, microcode resides in 2 KB of onboard RAM which is divided into two segments—a 1-KB microcode segment and a 1-KB entry point segment, as shown in Figure 3.

The microcode segment holds the 32-bit instructions that make up the state routines and functions. A state routine is a noninterruptable sequence of micro-instructions. A function can have up to 16 state routines that are accessed by the state address. The entry-point segment holds the address of the states.

Table 1 shows the possible entry points. There are four host-service request (HSR) entry points that are exactly the same HSR bits as for the preprogrammed function. The HSR entry points let the CPU start a TPU channel function. The other 12 entry points are based on channel activity.

The structure to define a state in TPUASM is:

```
%entry name= NAME; start ad-
  dress = ADDRESS;
  disable_match(enable_match);
cond hsr1= x, hsr0=x, lsr=x,
  m/tsr=x, pin=x, flag0=x;
(Insert microcode here)
end_of_phase or end_of_link
```

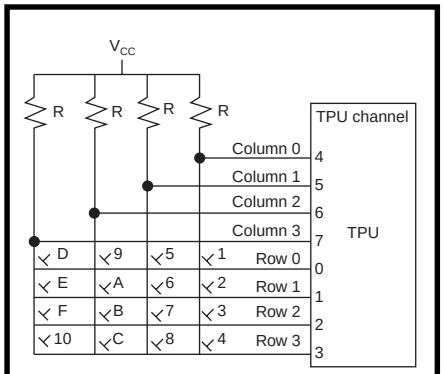
Each state in each TPU channel function must have this structure. Therefore, if you want to enter STATE at address START through a host service request when hsr1 and hrs0 are both one, use:

```
%entry name= STATE; start ad-
  dress = START;
  disable_match;
cond hsr1= 1, hsr0=1, lsr=x,
  m/tsr=x, pin=x, flag0=x;
```

All 16 states must be terminated properly with an end_of_state or end_of_phase even if they are not used.

When a request for service occurs on a channel, the scheduler determines when that channel gets access. Once the scheduler grants access to the requesting channel, control is passed to the appropriate state routine. The microcode for that state is executed and control is then returned to the scheduler.

From Table 1 you can see that the channel hardware can generate 12 different service requests. Each TPU channel has several registers that are used to capture and match based on events that occur on the output pin or on the two internal timers TCR0 and TCR1. The subinstruction CHAN sets the events that can produce a service request.



Key-pressed value	Key	Key-pressed value	Key
0 0000	0	0 1001	9
0 0001	1	0 1010	A
0 0010	2	0 1011	B
0 0011	3	0 1100	C
0 0100	4	0 1101	D
0 0101	5	0 1110	E
0 0110	6	0 1111	F
0 0111	7	1 0000	10
0 1000	8		

Figure 4—*Here's the hardware setup for 4 × 4 matrix keypad decoding. As you see, it requires eight TPU channels—four input channels for the rows and four output channels for the columns.*

CHAN SUBINSTRUCTION

The CHAN subinstruction controls the actions of the channel and the output pin. Its syntax is shown in Table 2. Note that there are four valid instruction formats—2, 3, 4, and 5—making CHAN the most complex subinstruction.

The pcs sets the channel pin state and can be PIN:=HIGH, PIN:=LOW, or PIN:=PAC. The pac controls the pin's action once a match occurs.

If the pin is programmed as an output, the action on the pin is determined by:

- pac:=high—on match event forces pin high
- pac:=low—on match event forces pin low
- pac:=no_change—on match event doesn't change pin state
- pac:=toggle—on match event forces pin state to toggle

If the pin is programmed as an input, the transition that is detected is determined by:

- pac:=high_low—high to low transitions detected
- pac:=low_high—low to high transition detected
- pac:=no_detect—no transitions detected
- pac:=any_trans—any transitions detected

The TBS controls the channel configuration, setting the channel as input or output, which TCR is matched, and which TCR is captured. These are the commands that are defined in Table 3.

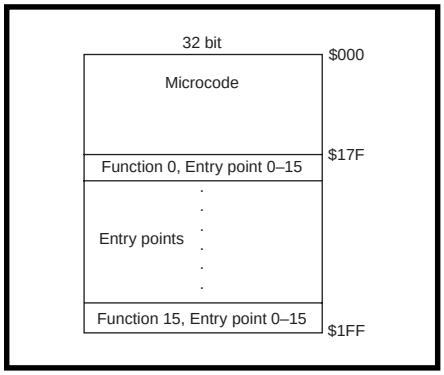


Figure 3—*In the '68332, the 2 KB of onboard RAM is divided into 1 KB for microcode and 1 KB for program entry points. The microcode instructions are 32 bits long. The formatting details were covered in Part 3.*

As you can see, CHAN has many options so the pin can be finely configured. chan tbs:=out_m2_c1, pac:=toggle is an example of setting up a channel. The channel pin is set up as an output on a match of TCR2, capture TCR1, and toggle the pin state.

Now that you've seen the basics of microcoding, it's time to look at an example. I realize that all of this entry point, service request latency, and states information can be complicated and a little intimidating, but you need to be informed if you want to use the full capabilities of the TPU.

My examples don't involve any of these issues, except for one entry point. In fact, many applications don't require entering and exiting between TPU functions.

Of course, there is a price to pay—normally, larger code size—but this disadvantage is offset by the fact that RAM not used for entry points can be used for program storage, thereby allowing programs greater than 1 KB. Thus, using an entry point from function 15 leaves all other entry point RAM available for program storage.

MATRIX KEYPADS

Matrix keypads are commonly included in embedded systems to permit access and control of a system or instrument. The technique for decoding a matrix keypad is fairly well known.

Each key is represented by a row and a column, as illustrated in Figure 4. When a key is pressed, the column and row lines are connected by the

key switch. Then, it's just a matter of determining which row and column are active.

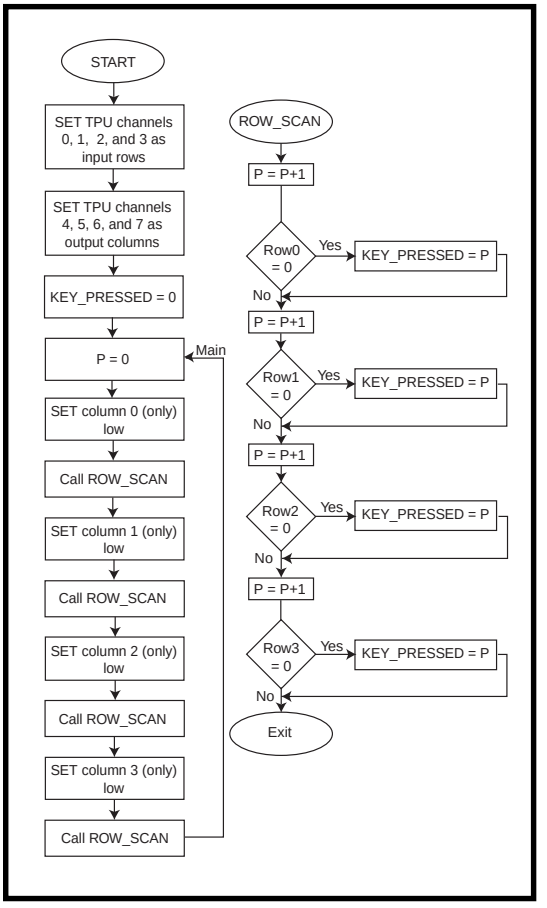
To determine the active rows and columns, pull all the column lines high and then systematically drive the column lines low one at a time. When a low appears on a row, it identifies the row that was selected.

The column is then known, and thus the key pressed is identified. Of course, you can do the reverse by driving the rows low and monitoring the column lines.

In this example, I use a 4 × 4 matrix keypad. I'll use four TPU channels as input and four TPU channels as outputs. The key pressed is sent to the CPU via parameter RAM location 7 of channel 15.

At this RAM location, a value of 0000 means that no key was pressed. A value other than 0000 indicates to the CPU which key was pressed (see Figure 4). It's a simple matter for the CPU to read this location for keypad information. The flowchart for the matrix keypad decode is shown in Figure 5, and Listing 1 shows you the microcode. (The `RUN_SCAN` function is not shown and is available via the Circuit Cellar web site.)

Figure 5—*This flowchart illustrates the TPU code for the 4 × 4 matrix keypad decode.*



The first thing I do in the microcode is set up the TPU channels. TPU channels 0, 1, 2, and 3 are set up as inputs and are the four rows using `chan TBS:=in_m1_c1`. Because I don't use the match and compare features, `m1` and `c1` are not relevant.

The next thing is to set TPU channels 4, 5, 6, and 7 as outputs and initialize their pins high using the two subinstructions `chan TBS:=out_m1_c1` and `PIN:=high`. As with the

input channels, the capture and compare features are not relevant. The chan_reg instruction is used to change channels.

The last thing I do before getting into the main loop of the program is to set KEY_PRESSED to zero. This indicates to the CPU that no key has been pressed.

Now the column scanning begins. I set the p register equal to zero and take the column 0 pin (line) low and check each row one at a time. The subroutine ROW_SCAN is used to scan the rows. I increment the p register by one prior to checking each row, but I only store p in the KEY_PRESSED location when a low is detected on a row.

On return from ROW_SCAN, set the column 0 pin high and the column 1 pin low. Then, call ROW_SCAN again and repeat the process for columns 2 and 3.

At this point, if a key had been pressed, KEY_PRESSED would have a nonzero value representing the key that had been pressed. If no key was pressed, then KEY_PRESSED would still be zero. That completes a scan of the matrix keypad, and the loop starts again at MAIN.

In this example, you can clearly see the benefit of being able to program the TPU, both in terms of CPU load sharing and overall system cost. If the TPU wasn't present, the CPU would have to perform the keypad decoding using valuable CPU time. If CPU time couldn't be spared, external circuitry would be required and system cost would be increased.

There are many additions such as a key debouncing, CPU interrupt, a wait to allow the CPU more time to read the KEY_PRESSED, or a keypad buffer that can be easily added to this program because most of the parameter RAM is unused. The next example takes advantage of the parameter RAM.

Listing 1—*In the TPU code for decoding a 4 × 4 matrix keypad, notice that the TPU runs autonomously, passing the keypressed value to the CPU via parameter RAM. ROW_SCAN is on the Circuit Cellar web site.*

```
%macro KEY_PRESSED 'prm7'. (Key pressed stored in parameter RAM location 7)
(Set TPU channels 0-3 as inputs rows, 4-7 as outputs, column, decoding, see
 Figure 4)
  au chan_reg:= #$00.       (Set TPU channel 0 as input)
  nop.
  chan TBS:=in_m1_c1.
  au chan_reg:= #$10.       (Set TPU channel 1 as input)
  nop.
  chan TBS:=in_m1_c1.
  au chan_reg:= #$20.       (Set TPU channel 2 as input)
  nop.
  chan TBS:=in_m1_c1.
  au chan_reg:= #$30.       (Set TPU channel 3 as input)
  nop.
  chan TBS:=in_m1_c1.
  au chan_reg:= #$40.       (Set TPU channel 4 as output, pin set high)
  nop.
  chan TBS:=out_m1_c1,
  PIN:=high.
  au chan_reg:= #$50.       (Set TPU channel 5 as output, pin set high)
  nop.
  chan TBS:=out_m1_c1,
  PIN:=high.
  au chan_reg:= #$60.       (Set TPU channel 6 as output, pin set high)
  nop.
  chan TBS:=in_m1_c1,
  PIN:=high.
  au chan_reg:= #$70.       (Set TPU channel 7 as output, pin set high)
  nop.
  chan TBS:=in_m1_c1,
  PIN:=high.
  au p:=#$00.
  ram -> @KEY_PRESSED.      (Start with KEY_PRESSED = 0, no key pressed).
MAIN:
  au p:=#$00.               (Reset p)
  ram -> @KEY_PRESSED.      (Start with KEY_PRESSED = 0, no key pressed).
  au chan_reg:= #$70.       (Change to TPU channel 7, column 3)
  PIN:=high.                (Take column high low)
  au chan_reg:= #$40.       (Change to TPU channel 4, column 0)
  PIN:=low.                 (Take column line low)
  Call ROW_SCAN, flush.     (Scan row to see if a key has been pressed)
  au chan_reg:= #$40.       (Change to TPU channel 4, column 0)
  PIN:=high.                (Return pin high)
  au chan_reg:= #$50.       (Change to TPU channel 5, column 1)
  PIN:=low.                 (Take column line low)
  Call ROW_SCAN, flush.     (Scan row to see if a key has been pressed)
  au chan_reg:= #$50.       (Change to TPU channel 5, column 1)
  PIN:=high.                (Return pin high)
  au chan_reg:= #$60.       (Change to TPU channel 6, column 2)
  PIN:=low.                 (Take column line low)
  Call ROW_SCAN, flush.     (Scan row to see if a key has been pressed)
  au chan_reg:= #$60.       (Change to TPU channel 6, column 2)
  PIN:=high.                (Return pin high)
  au chan_reg:= #$70.       (Change to TPU channel 7, column 3)
  PIN:=low.                 (Take column line low)
  Call ROW_SCAN, flush.     (Scan row to see if a key has been pressed)
                            (A Wait could be placed here to give CPU time to
                                read KEY_PRESSED )
  goto MAIN, flush.         (All rows and columns scanned, start again)
```

DATA COLLECTION

Many embedded applications require the acquisition and manipulation of data from external sources. Normally, an embedded CPU requests an ADC to convert an external analog signal to digital form, reads the ADC, and processes the data.

These functions are normally interlaced to maximize CPU throughput. However, it would be better if all the CPU had to do was read the data. This situation is where the TPU comes in handy.

Figure 6a shows a setup where the TPU controls the ADC and stores the data in TPU parameter RAM. The TPU can initiate the conversion, wait for the conversion to be complete, and store the data in parameter RAM for

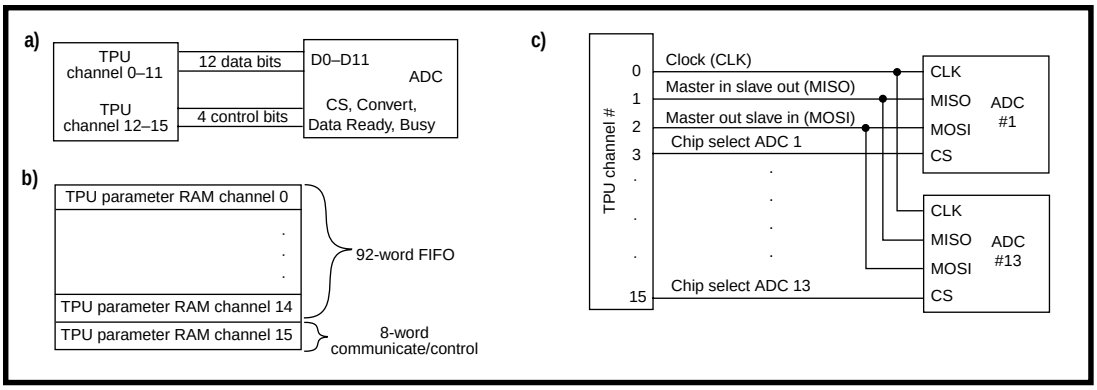


Figure 6—*Diagrams (**a**) and (**c**) show how the TPU can be used to control ADCs and store the data in TPU parameter RAM, which acts as a FIFO (**b**).*

the CPU to read it. The TPU could do this under CPU control.

A better approach is to have the TPU collect the data in a free-running mode and use the 100 words of parameter RAM as a FIFO, as shown in Figure 6b. The TPU parameter RAM is divided into a 92-word FIFO, and 8 words are used for communications and control between the TPU and the CPU.

The TPU collects the data as fast as it is generated by the ADC and stores it in parameter RAM where the CPU can take it off as needed. The CPU and TPU simply maintain FIFO pointers. The control portion of the parameter RAM can be used for things such as flagging FIFO full, half-full, and empty.

This setup gives the CPU many options when it comes to data processing because most processing algorithms require an array of data. These processing techniques vary from simple averaging to slightly more complex techniques, such as boxcar averaging, to full DSP techniques.

Another example of the TPU collection data for the CPU is shown in Figure 6c. The TPU reads 13 serial ADCs and stores the data in parameter RAM. Because the converters are serial, they require more time to transfer the data, which makes using the TPU even more appealing.

The example in Figure 6c follows the Motorola SPI serial bus. Three TPU channels pins are used as data clock (CLK), master in slave out (MISO), and master out slave in (MOSI). The remaining 13 TPU channel pins are used as chip selects.

Again, a FIFO in TPU RAM can be set up giving the CPU a great deal of

Command	Definition
in_m1_c1	Input, match TCR1, capture TCR1
in_m2_c1	Input, match TCR2, capture TCR1
in_m1_c2	Input, match TCR1, capture TCR2
in_m2_c2	Input, match TCR2, capture TCR2
out_m1_c1	Output, match TCR1, capture TCR1
out_m2_c1	Output, match TCR2, capture TCR1
out_m1_c2	Output, match TCR1, capture TCR2
out_m2_c2	Output, match TCR2, capture TCR2
write_mer	Writes match event register
neg_TDL	Negates transition detect latch
neg_MRL	Negates match recognition latch
neg_LSL	Negates link service latch
enable_mtrs	Enable service request
disable_mtrs	Disable service request
config :=p	Allows setup via CPU
cir	Sets host interrupt request for the current channel

Table 3—*This table expands two short parms for the CHAN formats 2, 3, 4, 5, and 9 from Table 2.*

flexibility. I used the Motorola SPI interface as a base, but you can use other serial buses such as I^2C.

In Figure 6a, external hardware can be used to increase the number of ADCs, data bits, or both. The TPU and CPU FIFO can just as easily be used with a DAC to generate arbitrary waveforms.

PUT IT TO WORK

It's important to realize the flexibility that the TPU provides for the systems, hardware, and software designers. The few examples I've given barely scratch the surface of uses for the TPU.

Even a company as large as Motorola couldn't anticipate or service every application with their preprogrammed functions in ROM. That's why the TPU is user-programmable.

Although this concludes the series on the TPU, no series of this length could possibly cover every detail and every application of the TPU. My goal was to present the TPU in general terms and still provide enough details so you could see how versatile the TPU is and how it can significantly increase CPU throughput.

Whether you use the many preprogrammed functions or write your own microcode, think of the TPU as a coprocessor and not just a timing unit.

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

REFERENCES

T. Harman, *The Motorola MC68332 Microcontroller*, Prentice-Hall, Englewood Cliffs, NJ, 1991.

Motorola, *Central Processor Unit*, 1990.

Motorola, *Time Processor Unit Macro Assemble Reference Manual*, TPUASM.

Motorola, *Time Processor Unit Reference Manual*, TPURM/AD, 1993.

Motorola, *TPU Microcode Technical Training Course*, MTT39/CN, 1996.

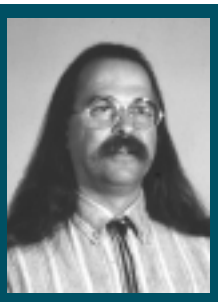
SOURCE

Microcontroller
Motorola
(512) 328-2268
Fax: (512) 891-4465
www.mot.com

Jeff Bachiochi

Dallas 1-Wire

Part 1: One on One



Hey, it's not just one time only! Jeff opens this new series on the Dallas 1-wire bus protocol by discussing 1-wire devices and how they are accessed when you use them alone on a single bidirectional I/O pin.

Those of you who work with small embedded micros know that there are always tradeoffs to be made between cost and function. You pay more, you get more. Whether it's code space, internal RAM, peripherals, or I/O, the more you need the more it costs.

It's no wonder that when designs change (via managerial enhancements), designers often go nuts trying to cram in the necessary bells and whistles. Sure, great designs maximize the use of available assets. Just don't expect tomorrow's bright idea to be easily implemented with today's minimalist designs. Maybe we engineers have to start outsmarting management by overcompensating for the anticipated deluge of last minute must-haves.

There are a number of successful approaches to adding function through external devices without demanding more and more I/O from a processor. To keep the required I/O to a minimum, these approaches tend to use some kind of serial protocol. SPI and I²C are two of the most popular.

SPI uses up to four signal lines (SCL, SDI, SDO, and CS), whereas I²C uses two signal lines (CLK and DATA). SPI is a shorter protocol because each device has its own chip select. I²C requires address information to be sent along with the data so it's a longer protocol. But, it only requires two signal lines, even when multiple devices share the same I/O.

THE CHALLENGE

Is it possible to reduce the necessary interface to a single I/O pin, yet still let multiple external devices communicate with a processor? Dallas Semiconductor accepted the challenge years ago by creating a line of 1-wire data devices.

To fit a 1-wire protocol, the device needs to communicate using a half-duplex protocol. This arrangement achieves bidirectional communication over the same wire.

The Dallas design is based on an open-collector type drive with a pull-up resistor to V_{CC}. Any device connected to the 1-wire bus can pull the idle state (bus held high by the pullup) down to ground with its open-collector output.

The computer or microcomputer that the external 1-wire device is connected to is considered the master. All external devices are considered slaves. Generally, the slave devices won't clamp the bus low unless they are responding to the master's request.

Most 1-wire devices are powered parasitically, which means that they derive their power from the bus (while it's pulled up). This setup gives their timed circuits energy to respond to a falling edge on the bus.

Communication between the master and slave devices is handled via read and write timeslots. A timeslot is a predetermined period of time that begins when an active state (low) is

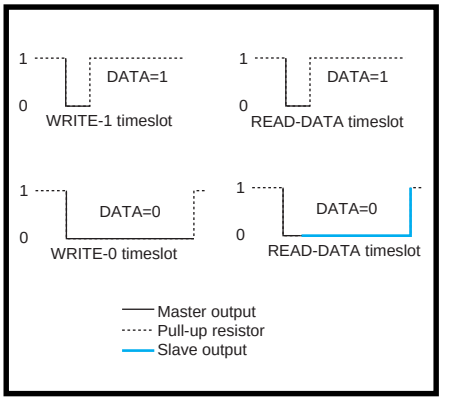


Figure 1—*After an initial low output, both write and read timeslots can hold the output low indicating a data bit of 0. When writing, the master controls the data, and when reading, the slave controls the data.*

applied to the 1-wire bus. The falling edge of this short pulse lets the other devices know when to read the data bus or write to the data bus for the remainder of the time slot.

There are two different slot timings—reset and data. The reset slot contains a low for at least 480 µs and a response time of not more than 300 µs. These long times are easily recognizable from the shorter data-slot times.

Data slots are no more than 120 µs. These consist of a maximum 15 µs low followed by the actual data, which is a high or low on the bus. The master's initial low gets all external devices listening to the bus.

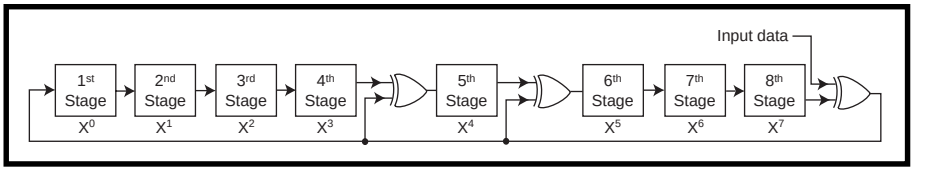


Figure 2—*A cyclic redundancy check (CRC) provides some assurance that the received data is good. The last byte of a transmission holds the CRC value calculated from the previous data bytes.*

If the external device is receiving data, it samples the bus after a minimum of 15 µs from the initial drop of the bus. If the external device must send data, the data is placed on the bus immediately following the initial drop (see Figure 1).

Remember that a slave device won't initiate data transfer on the bus. There must be a way for the master to tell an external device to respond. When a single device is part of the system, either as a permanent component or a socketed temporary touch device, a reset pulse should precede any commands.

Although devices that are permanently attached will always be present, temporary devices may or may not be there. The reset pulse is a request for any connected devices to respond with a presence pulse to indicate that at least one device is connected.

Listing 1—*With these assembly-language routines, you can use a bidirectional bit to talk to any Dallas 1-wire device.*

```
ASM
_RESET_P   bcf     PORTB,0   ;Force 1-wire I/O bit low
           bsf     STATUS,5  ;Point to Bank1 (direction control)
           bcf     TRISB,0   ;I/O directin=output, 1-wire forced low
           movlw   167       ;167 counts (500 µs)
           movwf   _cntr     ;into count register
RESET_P1   decfsz  _cntr     ;decrement count and skip next if=0
           goto    RESET_P1  ;else go back and decrement again
RESET_P2   bsf     TRISB,0   ;I/O direction=input, 1-wire pulled up
           bcf     STATUS,5  ;point back to Bank0
           movlw   20        ;20 count (60 µs)
           movwf   _cntr     ;into count register
RESET_P3   decfsz  _cntr     ;decrement count and skip next if=0
           goto    RESET_P3  ;else go back and decrement again
           clrf    _temp     ;initialize samples register=0
RESET_P4   movlw   36        ;36 counts (180 µs)
           movwf   _cntr     ;into count register
RESET_P5   btfss   PORTB,0   ;skip next if 1-wire is high
           incf    _temp     ;low so increment samples register
           decfsz  _cntr     ;decrement count and skip next if=0
           goto    RESET_P5  ;else go back, sample, decrement again
RESET_P6   movlw   80        ;count 80 (240 µs)
           movf    _cntr     ;into count register
RESET_P7   decfsz  _cntr     ;decrement count and skip next if=0
           goto    RESET_P7  ;else go back and decrement again
           return            ;done (temp returns sampled low count)
_W_B       rrf     _temp     ;rotate LSBit data into carry
           btfss   STATUS,0  ;skip next if carry (data)=1
           goto    W_B2      ;else jump to data=0 routine
W_B1       call    W1S       ;call the write-1 time slot
           goto    W_B3      ;go on
W_B2       call    W0S       ;call the write-0 time slot
W_B3       decfsz  _cntr     ;decrement count and skip next if=0
           goto    _W_B      ;else go back and do another bit
           return            ;done sending all bits
_R_B       call    R_S       ;call the read time slot
           movf    _rtemp    ;get the sampled low count
           bcf     STATUS,0  ;clear carry (read data=0 setup)
           btfsc   STATUS,2  ;skip next if sampled count <>0
           bsf     STATUS,0  ;set carry (read data=1)
           rrf     _temp     ;rotate data into the MSBit
           decfsz  _cntr     ;decrement count and skip next if=0
           goto    _R_B      ;else go back and do another bit
           return            ;done sending all bits
W1S        movlw   1         ;1 count (3 µs)
           movwf   _ctemp    ;into count register
```

(continued)

ONE ON ONE

Each device has a unique serial number associated with it, but when only one device is connected on the 1-wire bus, communication is simplified. The SKIP_ROM command (CCH) is the only command necessary to activate the device. SKIP_ROM commands the attached device to immediately wake up and take notice.

It's not necessary to know the ID number of the device. If you don't know the family of the device you're talking to (it may be any touch device), you can issue READ_ROM and the attached device will respond with its family code, ID number, and cyclic redundancy check (CRC) byte. From the family code byte you can determine what kind of device is connected.

The 8-byte ROM code contains the family code byte, 6 bytes of ID code, and a CRC byte. The CRC byte is based on the previous 7 bytes and can be used to authenticate the validity of the 8-byte transmission. Although a wrong CRC byte might be because of a bad data transfer, it could also mean that multiple devices are talking at the same time.

This time around, I'm limiting the discussion to single devices tied to an I/O pin. Figure 2 shows how the CRC byte is calculated. Each bit shifted into the CRC byte is XOR'd with bits 4, 5, and 8 of the CRC before shifting.

Listing 1—*continued*

```
            bcf     PORTB,0   ;force 1-wire I/O bit low
            bsf     STATUS,5  ;point to Bank1 (direction control)
            bcf     TRISB,0   ;I/O direction=output, 1-wire forced low
W1S1        decfsz  _ctemp    ;decrement count and skip next if=0
            goto    W1S1      ;else go back and decrement again
            bsf     TRISB,0   ;I/O direction=input, 1-wire pulled up
            bcf     STATUS,5  ;point to Bank0
            movlw   33        ;33 count (99 µs)
            movwf   _ctemp    ;into count register
W1S2        decfsz  _ctemp    ;decrement count and skip next if=0
            goto    W1S2      ;else go back and decrement again
            return            ;done sending bit
W0S         movlw   30        ;30 count (90 µs)
            movwf   _ctemp    ;into count register
            bcf     PORTB,0   ;force 1-wire I/O bit low
            bsf     STATUS,5  ;point to Bank1 (direction control)
            bcf     TRISB,0   ;I/O direction=output, 1-wire forced low
W0S1        decfsz  _ctemp    ;decrement count and skip next if=0
            goto    W0S1      ;else go back and decrement again
            bsf     TRISB,0   ;I/O direction=input, 1-wire pulled up
            bcf     STATUS,5  ;point to Bank0
            movlw   04        ;4 count (12 µs)
            movwf   _ctemp    ;into count register
W0S2        decfsz  _ctemp    ;decrement count and skip next if=0
            goto    W0S2      ;else go back and decrement again
            return            ;done sending bit
R_S         movlw   1         ;1 count (3 µs)
            movwf   _ctemp    ;into count register
            bcf     PORTB,0   ;force 1-wire I/O bit low
            bsf     STATUS,5  ;point to Bank1 (direction control)
            bcf     TRISB,0   ;I/O direction=output, 1-wire forced low
R_S1        decfsz  _ctemp    ;decrement count and skip next if=0
            goto    R_S1      ;else go back and decrement again
            bsf     TRISB,0   ;I/O direction=input, 1-wire pulled up
            bcf     STATUS,5  ;point to Bank0
            movlw   3         ;3 count (15 µs)
            movwf   _ctemp    ;into count register
            clrf    _rtemp    ;initialize sample count=0
R_S2        btfss   PORTB,0   ;skip next if 1-wire input=1
            incf    _rtemp    ;low so increment samples register
            decfsz  _ctemp    ;decrement count and skip next if=0
            goto    R_S2      ;else go back and decrement again
R_S3        movlw   16        ;16 count (48 µs)
            movwf   _ctemp    ;into count register
R_S4        decfsz  _ctemp    ;decrement count and skip next if=0
            goto    R_S4      ;else go back and decrement again
            return            ;done receiving bit
ENDASM
END
```

Although six assembly-language routines are shown in Listing 1—reset, write-1, write-0, read, write-byte, and read-byte—only the first four of these routines are necessary. With these four routines, you can make contact with almost any 1-wire device.

My first program was written as a generic program enabling you to experiment with various 1-wire devices. It was written for the PicStic using

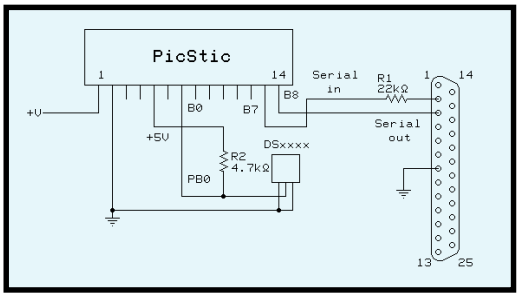


Figure 3—*Here, I used a PicStic as a 1-wire interface controller. Calls to assembly-language routines from BASIC enable me to quickly write simple applications using 1-wire sensors. Output messages go to either a PC or a serial LCD.*

```
RESET   <presence>

Command (hex)? CC (SKIP_ROM)
Response (bits-dec)? 0

Command (hex)? BE (READ_SCRATCHPAD)
Response (bits-dec)? 72
<temp> <pos/neg> <user1> <user2> <res1>  <res2>  <cr>  <cpc> <crc>
  48      0        0       0     255     255     41    81    51
```

Figure 4—*My first program was flexible enough to enable me to enter any of the 1-wire command bytes and receive responses from any of the devices connected. Here you can a* `SKIP_ROM` *command with no response and a* `READ_SCRATCHPAD` *command with a 9-byte (72-bit) response.*

PicBasic but it can be used with almost any PIC microprocessor.

I like to use the PicStic because it contains all the essentials you need every time you build a PIC circuit—a 5-V regulator, crystal, and supporting capacitors. And because the PicStic is electrically reprogrammable (it uses a '16F84), my debugging time is reduced. PicBasic lets me embed assembly language where necessary and still use the higher level commands to quickly build the framework.

You'll also need to know the specifics for each 1-wire device (i.e., which commands to use for which devices

Listing 2—*This BASIC program (using assembly routines from Listing 1) communicates with a DS1820 1-wire digital thermometer and displays the temperature (in Celsius and Fahrenheit) on a PC or LCD.*

```
'Written in PicBasic
'variable definitions
'b0 used in bit tests

symbol first=b5             ;Celsius value
symbol second=w3            ;b6 & b7 Fahrenheit value
symbol temp=b9              ;data being passed
symbol cntr=b10             ;bit counter
symbol ctemp=b11            ;temp counter
symbol rtemp=b12            ;temp zero counter

START: call RESET_P         ;1-wire reset (and presence)
  if temp=0 then NO_GOT_P   ;check for no device
  temp=$CC : cntr=8         ;SKIP_ROM command (it's 8 bits long)
  call W_B                  ;1-wire send
  temp=$44 : cntr=8         ;Convert Temperature command (it's 8 bits
                              long)
  call W_B                  ;1-wire send
  pause 500                 ;wait for conversion
  call RESET_P              ;1-wire reset (and presence)
  if temp=0 then NO_GOT_P   ;check for no device
  temp=$CC : cntr=8         ;SKIP_ROM command (it's 8 bits long)
  call W_B                  ;1-wire send
  temp=$BE : cntr=8         ;READ_SCRATCHPAD (it's 8 bits long)
  call W_B                  ;1-wire send
  cntr=8                    ;8 bits
  call R_B                  ;1-wire read
  first=temp/2              ;first=value read (divide by 2 for nearest
                              whole degree)
  second=temp*9             ;here's where the conversion
  second=second/10          ;to degrees F
  second=second+32          ;is done
  serout 7, n2400,(12)      ;form feed
  serout 7, n2400,("The Temperature is...",13,10)
  serout 7, n2400,(13,10,#first," Celsius",13,10,#second," Fahrenheit")
  goto START                ;do it all again...

NO_GOT_P:
  serout 7, n2400,("No device",13,10)
  goto                      START
```

and how many bytes a command will return, if any). Refer to the individual 1-wire datasheets for this information.

After wiring up the circuit in Figure 3, programming the PicStic, and applying power, you are presented with three choices—reset the 1-wire bus, send a command, and request data.

Commands are broken down into function and memory commands. A reset is required before any function command, and a memory command may follow a function command. The function and memory commands are write only, but either command may offer a response.

Let's try using the program on a DS1820 connected to the 1-wire bus. Following these steps, we receive the responses displayed in Figure 4.

If the device's V_{CC} lead is connected to ground (true 1-wire connection) you must not use the 1-wire bus for 500 ms after sending a convert temp command (44h). Because the device uses parasitic power in this configuration, it must see a high for this duration.

After that, you can proceed with reading the conversion. However, if you connect its V_{CC} lead to V_{CC}, you can poll the device for end of conversion status with `Response (bits-dec)? 8 <busy>`.

The response to `READ_SCRATCHPAD` has the temperature in the first byte. If you know it's above 0°, you can stop reading after the first byte.

The conversation the master has with a 1-wire device can be terminated at any time. The temperature reply is in 0.5°C increments. To get a true temperature in Celsius degrees, divide this by 2 (ditch the 0.5 bit).

To convert this temperature to Fahrenheit, perform the conversion calculation:

$$F = \left(\frac{9C}{5}\right) + 32$$

or, in our case (because Celsius is in 0.5° increments):

$$F = \left(\frac{9C}{10}\right) + 32$$

This circuitry and software can easily be massaged into displaying the present temperature in both Celsius and Fahrenheit with a bit more software.

Using a serial LCD can eliminate the serial connection to your PC that would be used in the experimental and debugging process. Listing 2 shows you what my BASIC program looks like (minus the same assembly-language routines that are used in Listing 1).

Next month, I'm going to look at the 1-wire bus with multiple sensors. I'll show you how to get these devices to act civilly with one another.

You might want to check out Dallas's web site for documentation on 1-wire devices. They have serial and parallel port dongles for reading external touch memory devices.

Software for reading these on the PC is also available, if you're into that. I prefer to provide my micros with the ability to use these devices directly. I hope that's your direction, too.

Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on Circuit Cellar INK*'s engineering staff. His background includes product design and manufacturing. He may be reached at jeff.bachiochi@circuitcellar.com.*

SOFTWARE

Source code for this article, as well as a Dallas 1-wire parts list, may be found via the Circuit Cellar web site.

SOURCES

1-wire devices
Dallas Semiconductor
(800) 336-6933
(972) 371-4000
Fax: (972) 371-3715
www.dalsemi.com

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

PicBasic
microEngineering Labs
(719) 520-5323
Fax: (719) 520-1867
www.melabs.com

SILICON UPDATE

Tom Cantrell

Maximicro

Tom says that the MAX1460 has lots of talent but its processor will never be a star. Why? Because it doesn't aspire to be. That's the innovative twist he applauds on Maxim's first step into the micro market.

As in a movie, the big star chips (like micros and memories) garner all the attention and adulation. But although they're given little more than a few seconds of scrolling fine print in the credits, others are just as important when it comes to getting the film in the can.

The headliners rely on a supporting cast of chips to handle glue logic and real-world interfacing. Regulators, voltage comparators, op-amps, filters, analog converters, and communications transceivers all have roles to play.

In the quest to cram everything on a single chip, the next step is to integrate these nondigital support functions onto the MCU, creating what's best described as a mixed-signal micro.

Examples of the trend abound. Microchip made a deal with Seattle Silicon and introduced the PIC16HV540, which includes a built-in voltage regulator (use any 6–12-V battery or wall-mount supply) and high-voltage (15 V) I/O for direct connect to sensors and relays.

The trend also presents an opportunity for new suppliers to enter the MCU market. For instance, Analog Devices leveraged their mixed-signal know-how with the AduC812, which combines a fast (200 kHz) eight-channel 12-bit ADC, two-channel 12-bit DAC, and flash-based 8051 MCU core.

It's not hard to imagine other mixed-signal leaders making a move. So, I was excited, but not surprised, when I heard Maxim was working on a processor.

For those (presumably few) of you not familiar with the company, Maxim has done well by offering a huge selection of mixed-signal problem solvers including ADCs, DACs, op-amps, supervisors, transceivers, and the like.

This being their first showing in the processing theatre, the MAX1460 is best described as a smart ADC because the on-chip 16-bit processor isn't user programmable. The processor isn't the star; it plays a supporting role.

Figure 1 gives you a backstage look. The on-chip processor is one link in a signal chain that includes a programmable gain amp, 16-bit ADC, temperature sensor, 12-bit DAC, and op-amp.

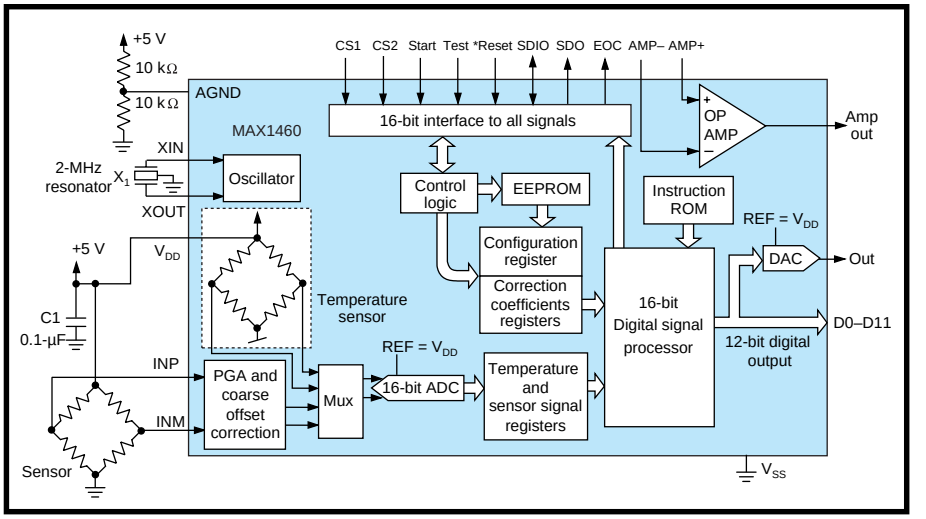


Figure 1—*The MAX1460 combines everything needed to accommodate a variety of bridge-type sensors. Major integrated functions include a programmable gain amp (PGA), 16-bit ADC, temp sensor, and 16-bit processor.*

The micro is described as a RISC or DSP, presumably combining aspects of both (not that there's much difference anymore). At this point, the details of the architecture and on-chip ROM code haven't been disclosed, although much can be inferred by the functions they perform.

PIEZO DE RESISTANCE

To get a better idea of what the MAX-1460 is, let's look at where it came from and what it's supposed to do.

The MAX1460 is designed to act as a signal-conditioning front end for a variety of sensors, especially those based on the classic Wheatstone bridge. The Wheatstone-bridge concept exploits the fact that physical energy (tension or compression) causes a change in resistance detected as a voltage differential between the two legs of the bridge.

The Wheatstone bridge is versatile. Use gravity as the physical input and you've got a scale. Turn it sideways and it's an accelerometer. Mechanically connect it, and voilà, a strain gauge. Use the bridge as a diaphragm and it detects pressure, level, and flow.

That's the good news. The bad news is extremely low-level output that is hard to detect and easily swamped by ambient electrical interference.

A typical piezo-resistive bridge sensor may be able to generate only 20 mV or so per volt of excitation. In a 5-V system, that means the peak output is 100 mV, which even at meager 8-bit resolution requires the ability to detect low-level signals down to hundreds of microvolts per least-significant bit. Push for 12 bits and you're talking nanovolts!

Resistors also react to temperature (i.e., thermistor), which is nice when that's what you want to do. Otherwise, it's a pain.

An uncompensated sensor may exhibit thermal errors on the order of 10, 20, or even 30% of full-scale output (FSO) including significant (e.g., 5% FSO) nonlinearity. Without temperature compensation, there's no need to worry about A/D resolution because even a few bits of accuracy will be tough to obtain.

You know the drill. Get some op-amps, a temp sensor (to compensate), ADC, MCU, and a grab bag of discretes, then have at it. Amplify the input and remove any offset, compensate (both gain and offset) for temperature, and output the digital result.

Earlier parts like the MAX1457 and '58 (see Figure 2), put together a few parts of the puzzle. The main difference is that the '57 uses a relatively large (128 × 16) external EEPROM to store temperature compensation coefficients and the '58 has a smaller EEPROM (8 × 16) on chip.

Photo 1—*They call it a smart ADC, and the MAX1460 arguably represents Maxim's first entry into the micro market.*

The different size of the EEPROM reflects a fundamental difference in the '57's and '58's approach to compensation. It starts by realizing that the goal of compensation is to come up with curves that cancel the sensor's output and offset thermal error curves.

Every programmer knows there are two ways to generate a curve. Either come up with an equation that defines the curve, or use a table lookup. That's the difference between the '57 and '58.

The '57 relies on its larger EEPROM to store a table of output and offset compensations corresponding with up to 120 temperature points. This setup enables the output to be fine-tuned across the entire temperature range, yielding excellent 0.1% accuracy.

Meanwhile, the '58 (essentially an analog computer hardwired to execute one calculation) uses an equation incorporating error coefficients. An equation of only a few terms takes up a lot less room in memory than a table, hence the '58's smaller EEPROM.

But, a simple equation usually does not perfectly fit a real-world curve, so the '58's accuracy (1%) isn't as good as the '57's. Still, 1% is good compared to the 10–20% or greater error of an uncompensated sensor.

Although a big improvement over rolling your own, the '57 and '58 are still analog from the system designer's point of view. Not only do you have to come up with an ADC and the software to babysit it, you're likely to encounter problems getting the analog signal from here to there, especially in a high-temperature or otherwise harsh environments where the sensor and controller are physically separated.

LITTLE SIGNAL, BIG BITS

Enter the '1460. By adding the ADC and micro, the '1460 puts everything under one roof, inputting a tiny signal from the bridge and outputting accurate (0.1%, 12 bit) 1s and 0s. The operating principle is exactly the same as before, but now most of the details and sensitive analog signals are handled on chip.

The design is ratiometric (i.e., relative to supply voltage) with input centered around a virtual ground, obtained by connecting 10-kΩ pull-up and pull-down resistors to AGND. Maxim recommends avoiding the extremes of the input range for best signal-to-noise ratio (SNR). So, it's best to try to keep the input between 1 and 4 V rather than the full-scale 0–5 V.

The first step is coarse amplification (×46, ×61, ×77, or ×93) and offset correction (eight selections) to get dynamic range into the ballpark. The output is a differential voltage between $-V_{DD}$ and $+V_{DD}$ that, post-A/D conversion, is interpreted as an integer value between –32768 and +32767 or, for correction coefficients, a fraction between –1.0 and +0.99997.

Once digitized by the 16-bit ADC, the coarse-adjusted input and on-chip temperature-sensor reading are fed to the processor. Although the output is only 12 bits, 16 bits of resolution are required here to prevent quantization and rounding errors from polluting the final output.

The other set of processor inputs comes from the correction-coefficient registers in Table 1 that reflect the offset

and output specs for a particular sensor, including thermal characteristics.

With digitized input and temp and the correction coefficients in hand, the processor executes Output = Gain (1+ G_1T + G_2T2) (Signal + Of_0 + Of_1T + Of_2T2) + D_{OFF} to generate an output that compensates the sensor's offset and gain (see Figure 3).

Gain and D_{OFF} deal with the basic gain and offset of the sensor and G_1, G_2, Of_1, and Of_2 handle linear and nonlinear components of thermal compensation. Essentially, the MAX1460 processor does in software what the earlier parts did with analog circuits.

Once the calculation is complete, the 12-bit result is output in parallel on the D0–D11 pins. It's simultaneously delivered as a bitstream on the OUT pin. Feeding OUT back through the spare on-chip op-amp and adding a few discretes permits a high-level (rail-to-rail) analog voltage or 4–20-mA output to be generated.

It may seem odd for a sensor to go both ways (i.e., analog and digital outputs), but often, industrial designers require legacy analog capability to help smooth the migration of their customers and installed equipment base to fully digital designs.

From the system designer's perspective, MAX1460 operation is simple. Just power up and the signal is acquired and processed, and 67 ms later the outputs (D0–D11, OUT) are updated and the EOC pin signals completion. Two chip-select pins (CS1 and CS2) condition all digital I/Os, so multiple '1460s can reside on a common bus.

Conversion can also be initiated with the START pin. This feature is useful if the default 35-ms sensor warm-up time is insufficient. There's a repeat mode that automatically reinitiates conversion (max. sample rate is 15 Hz).

SERIAL SETUP

Using the MAX1460 is easy. But, setting it up (i.e., determining the proper calibration coefficients and programming them into the on-chip EEPROM) is another story.

With the TEST pin asserted, two other pins (SDIO and SDO) provide access to the chip's inner workings. The first gotcha is that in Test mode, the host system is responsible for providing a 2-MHz 50% duty-cycle clock on XIN that not only clocks transfers on SDIO and SDO but also runs the chip. According to the designers, the

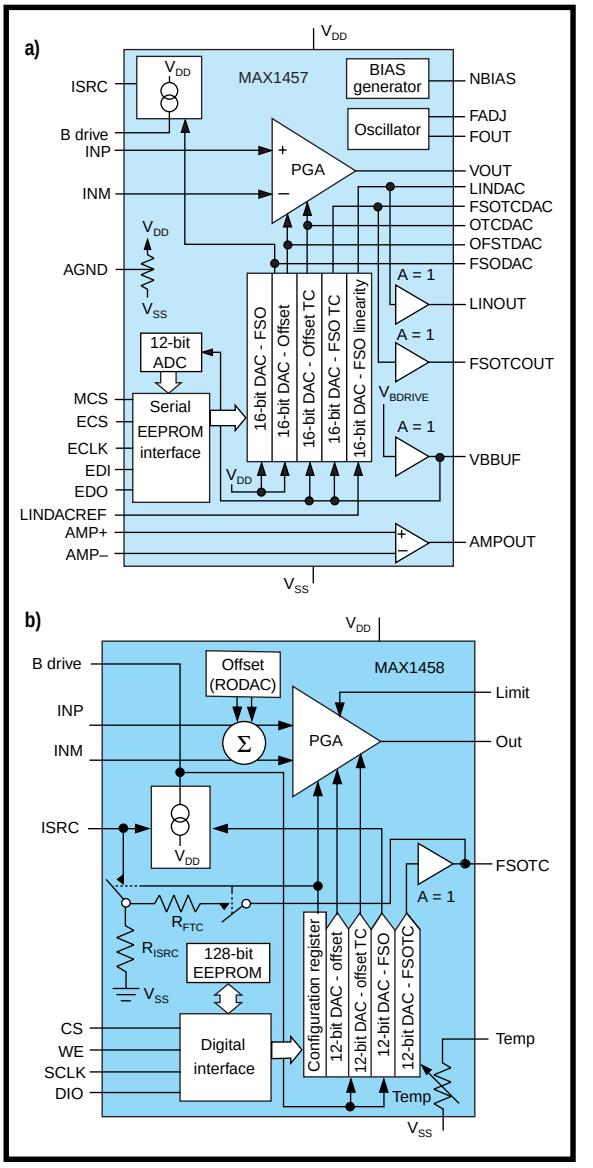


Figure 2—*Earlier Maxim parts, the MAX1457* ***(a)*** *and MAX1458* ***(b)****, targeted the same bridge sensor applications but required support from an external ADC and micro.*

Coefficient	Register address	Function	Range	Format
Gain	1	Gain correction	–32,768 to +32767	Integer
G_1	2	Linear TC gain	–1.0 to +0.99997	Fraction
G_2	3	Quadratic TC gain	–1.0 to +0.99997	Fraction
Of_0	4	Offset correction	–1.0 to +0.99997	Fraction
Of_1	5	Linear TC offset	–1.0 to +0.99997	Fraction
Of_2	6	Quadratic TC offset	–1.0 to +0.99997	Fraction
D_{OFF}	7	Output midscale pedastal	–32,768 to +32767	Integer

Table 1—*The key calculation performed by the on-chip processor compensates sensor offset and gain for temperature by using sensor-dependent correction coefficients.*

clock could probably range from about 1 to 3 MHz before it impacts conversion accuracy, but the datasheet says 2 MHz, period.

Calibration for a particular MAX-1460 and sensor pair involves determining the appropriate values for the configuration and correction-coefficient registers. The idea is to operate the sensor and MAX1460 pair under various temperature (low, medium, and high) and input-excitation (low and high) conditions to gather enough data points to fit a curve using the output equation and correction coefficients.

To communicate with the chip, the host system issues commands (see Table 2) and monitors processor activity and results via SDIO and SDO. The processor reveals the contents of the accumulator (S), the 8-bit program counter (P), and the 8-bit instruction it's currently executing (PS). A new instruction and corresponding 32 bits of status (S, P, and PS) are delivered every 16 clock cycles.

After issuing the start-conversion command, the host looks for the appearance of certain opcodes on SDIO to catch the conversion result on SDO exactly 130,586 XIN clock cycles later (i.e., 65.293 ms). Remember that all of this happens at 2 MHz.

Once the sensor-specific configuration and calibration is determined, it's time to burn it into the 128-bit (8×16) EEPROM. This exercise calls for a different (125 kHz) clock and multiple commands to program each bit individually.

I don't see any commands to read the EEPROM; only one to erase it. Not sure how comfortable I am with the concept of a write-only memory. On the other hand, it offers security to anyone worried about having their coefficients ripped off. I suspect there's a way to read the EEPROM, but it's not in the datasheet.

Once programmed and put back into service (TEST deasserted), things get simple again. After powerup, the MAX-1460 automatically loads the EEPROM contents into the configuration and correction-coefficient registers and does its thing.

EV WAY OUT

The fact that the setup procedure is nontrivial would seem to dampen aspirations for self-calibrating designs that could adapt over time, perhaps to compensate for sensor aging or environmental changes.

To their credit, Maxim appears to disclose enough details to enable you to support in-system self-reconfiguration. But, they clearly expect that

Command	Hex Code
Write a calibration coefficient into a DSP register	1 hex
Block-erase the entire EEPROM (write 0 to all 128 bits)	4 hex
Write 1 to a single EEPROM bit	2 hex
NOOP (no operation)	0 hex
Start conversion: registers are not updated with EEPROM values. SDIO and SDO are enabled as DSP outputs	8 hex
Start conversion: registers are updated with EEPROM values. SDIO and SDO are enabled as DSP outputs	A hex
Start conversion: registers are not updated with EEPROM values. SDIO and SDO are disabled	C hex
Start conversion: registers are updated with EEPROM values. SDIOand SDO are disabled	E hex
Reserved	3, 5, 6, 7, 9, B, D, F hex

Table 2—*Calibration and EEPROM programming is carried out by a host system that issues commands to the on-chip processor and monitors operation via the SDIO and SDO pins.*

reprogramming the EEPROM won't be done often and will be handled by a test system rather than logic built into the application.

To that end, they offer the MAX-1460 evaluation kit. It includes an EV board with the '1460 and sample pressure sensor, an interface board that connects to a PC parallel port, Windows software that walks the user through calibration, and EEPROM programming and documentation.

Keep in mind that calibration for a custom-matched MAX1460 and sensor pair is tedious. It may take a couple hours because the sensor has to soak for about 30 min. at each temperature. Thus, the software assist that Maxim provides to automate the process is a welcome addition.

Although it's targeted at a fairly narrow niche, I find the MAX1460 intriguing in its implications for the bigger picture. It provides more evidence of the trend towards mixed-signal micros and smart sensors.

The Maxim folks I spoke with indicated that opportunities related to IEEE 1451.2 (the digital sensor interface discussed in *INK* 104) are under review.

Finally, Maxim's entry into the micro marketplace, albeit rather indirectly, may portend exciting times for MCU customers and competitors.

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.

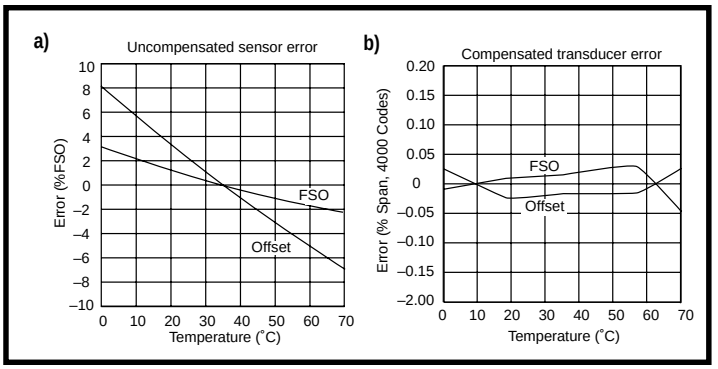


Figure 3—*Once calibrated, the output accuracy across temperature of the MAX1460+ sensor pair* ***(a)*** *is dramatically improved compared to the uncompensated sensor* ***(b)****.*

SOURCES

MAX1457, '1458, '1460
Maxim Integrated Products
(408) 737-7600
Fax: (408) 737-7194
www.maxim-ic.com

PIC16HV540
Microchip Technology, Inc.
(602) 786-7200
Fax: (602) 899-9210
www.microchip.com

AduC812
Analog Devices
(617) 329-4700
Fax: (617) 329-1241
www.analog.com

PRIORITY INTERRUPT

Sitting in the Dark

You get up one morning and get ready to jump in the shower. It smells funny. Obviously your local water treatment facility is having a little "control" problem. Instead, you get dressed and jog over to the local BART station for the ride to work. Unfortunately, it seems that the trains aren't running. You return home to get your car and drive to work but red lights are flashing at many intersections. Some idiot runs the flashing red light in front of you and crashes into another car. You call 911 on your cellphone, but no one answers. The 911 emergency system isn't working.

Three hours later you make it to the office. You would have been a bit earlier but the elevator was down and it took you 20 minutes to climb 16 floors. Fortunately, you're just in time for the meeting with the lawyers hired to carry out a company merger. The bad news is that apparently the state has lost your company's corporation records and all the lawyers can say is that it will probably be rather expensive to fix the problem. In disgust you say "I need a drink!" and suggest that everyone head (down the stairs) to the restaurant across the street.

After a few rounds you discover that you're the only one with enough cash to cover the check. No one else could find a working ATM this morning and the bar's credit card scanner rejects any card with a '00 expiration date. Work is a total loss so you leave and head over to "The Video Den" to check on the HDTV you put a $2000 deposit on last week. When you get there, they have no record of your order or the deposit. So, you call and ask a friend to drop by your apartment and get your receipt but the telephone line is dead. And on, and on….

As humorous as this sounds, it is certainly an unpleasant scenario. Of course, the media will have you believe that reports like this are going to be the norm after January 1st, 2000. So why am I talking about this now? Well, after July, I'm sure Y2K is all you're going to hear about. Unfortunately, because so many of the problems will undoubtedly involve embedded controls, I can't ignore the issue entirely—especially after recently running into a Y2K problem myself.

As many of you know, my house is controlled by a computerized home automation system (HCS-II) which was designed and documented here in the magazine. It has performed flawlessly and now includes many enhancements and peripherals. Like most HCS users, I only update the software when I change the system configuration (if it ain't broke, don't fix it).

Last year, users asked if the HCS was Y2K compatible. We set a PC for 11:59 PM on 12/31/99 and had it update the HCS real-time clock. After a minute, the HCS clicked over to 01/01/00 12:00 AM—and kept going! We announced that the HCS was Y2K compatible. Which meant I didn't have to rip into my own HCS to change the software.

Hold on there. A few weeks ago we got a call from a HCS user who ran the same test we performed and got the same results. Then he set the PC for the middle of January 2000 and tried to set the HCS clock. Disaster! Apparently, the HCS could rollover correctly from 1999 into the new millenium, but it couldn't be set correctly by a PC once the date was actually in the year 2000. We fixed the problem immediately and issued a new software release (V.3.62). But, without a concerned user we might not have discovered the problem until someone actually updated their clock next January. How many other situations like this are out there?

This bit of personal horror made me think harder about the Y2K issue in general. I'm not jumping on any Y2K bandwagon. I don't expect chaos. Unfortunately, regardless of its hype, the Y2K problem is real and embedded chips are the greatest risks. Embedded microcontrollers have become key components in complex commercial products such traffic lights, power generation equipment, water and sewer systems, airport runway lights, antenna aiming systems, and so on. In between are the products that average consumers encounter daily—elevators, fax machines, televisions, VCRs and microwave ovens, to mention just a few.

An elevator is a noteworthy example of the Y2K problem. If the interval between maintenance checks is exceeded (as calculated by the embedded chip), most elevators become inactive and go to the bottom of the shaft to await maintenance. If the chip reads 00 and calculates the maintenance interval from 1900, the shutdown procedures activate and the elevator becomes inoperable.

Fixing Y2K problems is going to be an exercise in rounding up enough programmers (manpower) who ask the right questions (intelligence). I had a couple thousand HCS owners checking to see if we had a problem. The real Y2K problem is in finding the manpower to check all of the world's elevators. With all the embedded controllers out there, ultimately some of them won't get the millenium message.

As you can see, embedded control is a big part of the Y2K situation. I'm not trying to add to the hype, I'm just endeavoring to balance things with a little personal experience. Unfortunately, talk can't fix it and I have eight months to update my HCS software—or else.

steve.ciarcia@circuitcellar.com