CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

DEBUGGING, EMULATORS & SIMULATORS

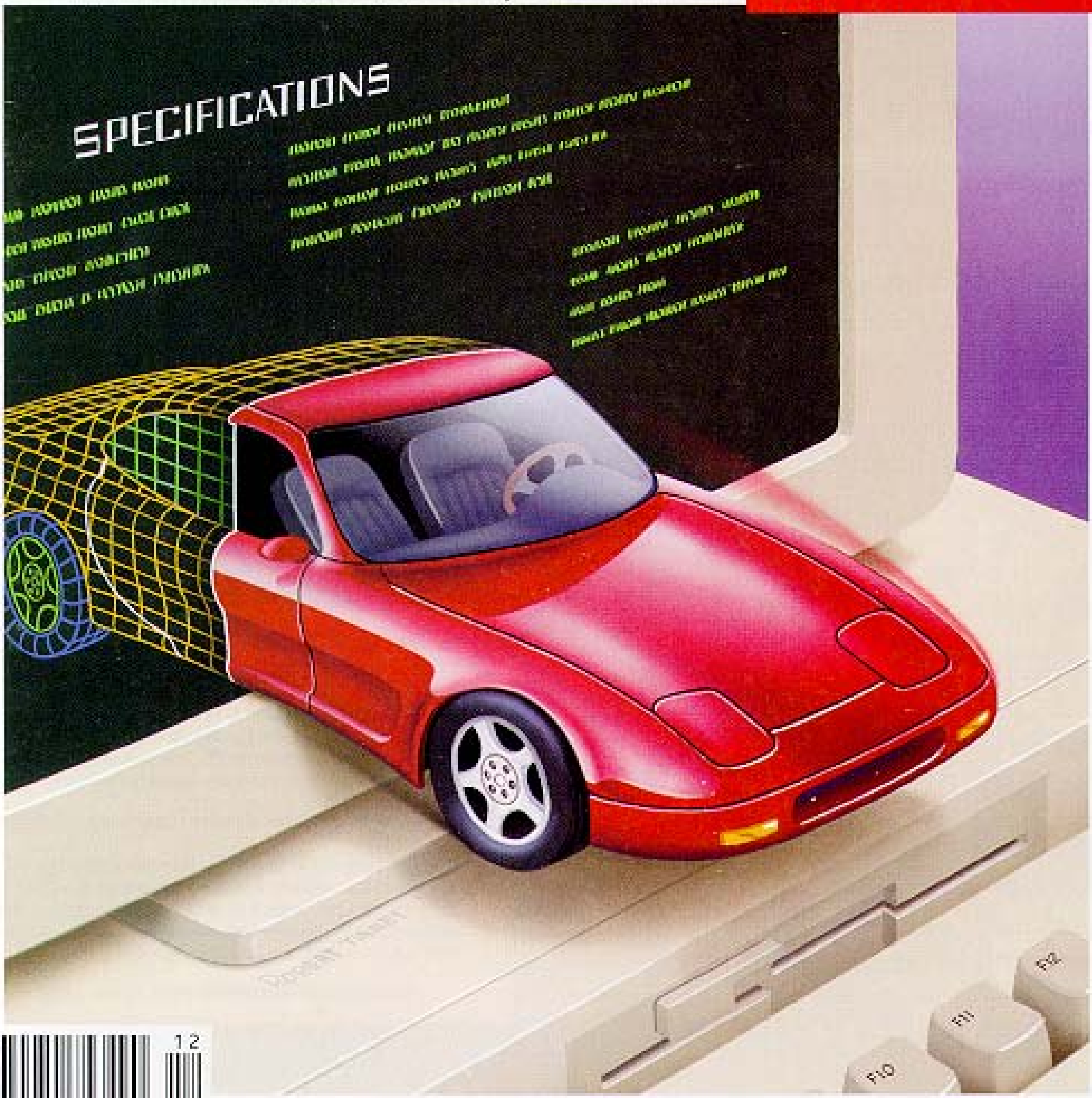
SPECIAL SECTION:
Embedded Control & Conversion

EPROM Emulator

Measuring Microvolts

Design Contest Winners

December '92/January '93 — Issue #30



$3.95 U.S.
$4.95 Canada

EDITOR'S INK

Same Great Articles, Twice as Often

have you heard the news? If you've already read Steve's editorial, or you were particularly observant when checking out the subscription cards in the last issue, then you know. If not, let me be the first to tell you: the Computer *Applications Journal* is going monthly. That's right. Now you can look forward to finding a new issue in your mailbox each and every month.

It was just about a year ago when we redesigned the look of the *Computer Applications Journal* and I promised that the quality of our editorial would remain as high as ever. Now, as we shift gears and go monthly, I continue the same pledge of high-quality editorial. All of our regular columnists will be in each issue, and we have some new features to go along with the change. The fun starts with the next issue, February 1993, and will continue each month thereafter.

In the last issue, I listed the themes for the upcoming year, but couldn't give any dates because we hadn't released the news of going monthly, yet. Now, I want to repeat those themes, but put some dates with them so you know the time frame with which you're working should you have expertise in a particular field and want to write an article. The date following each theme is the deadline for proposals, so be sure to get your ideas in early.

February, issue #31:	Home and Building Automation	(1011192)
March, issue #32:	Embedded Interfacing	(1 1/2/92)
April, issue #33:	Data Acquisition	(12/1/92)
May, issue #34:	Graphics and Video	(1/1/93)
June, issue #35:	Communications	(2/1 193)
July, issue #36:	Real-Time Programming	(3/1/93)
August, issue #37:	Measurement and Control	(4/1/93)
September, issue #38:	Signal Processing	(5/3/93)
October, issue #39:	Power Control and Conversion	(6/1 193)
November, issue #40:	Programmable Devices	(7/1/93)
December, issue #41:	Embedded Control	(8/2/93)

The big news for this issue is the results of the Fourth Annual Circuit Cellar Design Contest. We get some really creative entries each year, and this year was no exception. We hope to have full-length articles describing each of the winning projects sometime in the upcoming year.

The first feature article in this issue, "Build the SmartROM EPROM Emulator," was a winner in last year's contest and has been the source of questions from more than a few of our readers. Other articles include more hardware and software development tools, data collection and control projects, and a nifty application by Ed that teaches both basic physics and some tricky computer design tricks at the same time.

The theme for our next issue is Home and Building Automation, which is always one of the most popular issues of the year. We have some great articles lined up, so it's one not to be missed.

Ken

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

FOUNDER/EDITORIAL DIRECTOR
Steve Ciarcia

EDITOR-IN-CHIEF
Ken Davidson

ASSOCIATE EDITOR
Lisa Nadile

TECHNICAL EDITOR
Dan Woods

ENGINEERING STAFF
Jeff Bachiochi & Ed Nisley

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITOR
John Dybowski

NEW PRODUCTS EDITOR
Harv Weiner

ART DIRECTOR
Lisa Ferry

STAFF RESEARCHERS:
Northeast
John Dybowski
Midwest
Jon Elson &Tim McDonough
West Coast
Frank Kuechmann

Cover Illustration by Robert Tinney

PUBLISHER
Daniel Rodrigues

PUBLISHER'S ASSISTANT
Susan McGill

CIRCULATION COORDINATOR
Rose Mansella

CIRCULATION ASSISTANT
Barbara Maleski

CIRCULATION CONSULTANT
Gregory Spitzfaden

BUSINESS MANAGER
Jeannette Walters

ADVERTISING COORDINATOR
Dan Gorsky

CIRCUIT CELLAR INK, THE COMPUTER APPLICATIONS JOURNAL (ISSN 0896.8985) is published monthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon. CT 08066 (203) 875-2751. Second class postage paid at Vernon, CT and additional offices. One year(l2 issues) subscription rate U.S.A. and possessions $21.95. Canada/Mexico $31.95, all other countries $49.95 All subscription orders payable ın U.S. funds only, via international postal money order or check drawn on U.S. bank. Direct subscription orders to The Computer Applications Journal Subscriptions, P.O. Box 3050, Southeastern, PA19398 or call (215) 630-1914.

POSTMASTER: Please send address changes to The Computer Applications Journal, Circulation Dept., P.O. Box 3050, Southeastern, PA 19398.

HAJAR ASSOCIATES NATIONAL ADVERTISING REPRESENTATIVES

NORTHEAST
Debra Andersen
(617) 769-8950
Fax: (617) 769-8982

MID-ATLANTIC
Barbara Best
(908) 741-7744
Fax: (908) 741-6823

SOUTHEAST
Christa Collins
(305) 966-3939
Fax: (305) 985-8457

MIDWEST
Nanette Traetow
(708) 789-3080
Fax: (708) 789-3082

WEST COAST
Barbara Jones
& Shelley Rainey
(714) 540-3554
Fax: (714) 540-7103

Circuit Cellar BBS—24 Hrs. 3001120012400 bps, 8 bits, no parity, 1 stop bit, (203) 871-1988; 9600bps Courier HST, (203) 871-0549

All programs and schematics in Circuit Cellar INK have been carefully reviewed to ensure their performance is in accordance with the specifications described, and programs are posted on the Circuit Cellar BBS for electronic transfer by subscribers.

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

INSIDE ISSUE 30

SPECIAL SECTION Embedded Control & Conversion

READER'S INK

TIME DOMAIN QUESTIONS

Thanks for the interesting article about the time-domain reflectometer. There were a few things that the article did not mention, however, and I wonder if you might clarify them.

The review of the theory of TDR and the description of the user interface were excellent, but you never mentioned in what form the results of the test are presented to the user. I sort of gather that the instrument looks for the (first? largest? last?) reflection and calculates the distance to that point and the impedance (resistance) there. How are these results displayed, and is it possible to search for additional reflections?

It seems that the hardware can answer the question, "Was there a low-to-high and/or a high-to-low transition across voltage V after time T!" Can you give a brief explanation how the software derives from this the delay, direction, and magnitude of reflected pulses?

I'm curious about any theoretical and/or practical reasons that you chose to use a pulse waveform instead of a step waveform for the measurement. Does the software calibrate itself by measuring the amplitude of the outgoing pulse, or is it assumed to be constant?

I like the idea of using a latching relay to save power when selecting the cable termination, but it appears in Figure 4 that R31 (100 ohms) is in parallel with the resistor (R8/R9) selected by K1/S2. Does the software somehow take this into account!

Dave Tweed
Littleton, Mass.

John Wettroth responds:

The TDR displays a distance to biggest (largest absolute value reflection coefficient) discontinuity in the cable and a termination resistance value. For large and small reflection coefficients, it displays open or shorted; otherwise, a resistance (in ohms).

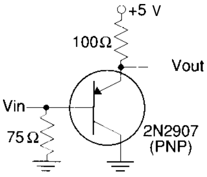
The TDR measures the amount of time required to get a pulse back and adjusts a DAC to a level corresponding to its level. Distance is cable velocity/time with a velocity factor for the cable type. Resistance is calculated by reflection coefficient equation solution after a correction is made for loss in the cable for distance traveled.

A pulse waveform was used for practical reasons. Because of the specifics of how I search for the reflected waveform using the DAC and a comparator, it was easier. Step waveforms create return waveforms with steps and were more complicated to find in software. Looking at the waveforms on a scope is more complicated than a normal TDR scope display, but since the unit doesn't provide a display, I took the easy way out. In response to your question about assuming the pulse height is constant, this software uses two pulse heights: one for 50 and one for 75 ohms. This area of the software could be improved by measuring the outgoing pulse. I played with this somewhat, but it's difficult; assuming it is constant doesn't usually produce significant errors.

The value of R31 is supposed to be 1000 ohms. I noticed that myself. Its only purpose is to keep the transistor tied when switching resistances. It could even be eliminated with little effect.

IMPROVED BUFFERS

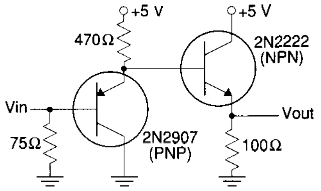
There appears to be a considerable amount of interest in driving multiple VGA monitors from a single display adapter [judging from magazine ads and recent Circuit Cellar BBS discussions). I was recently hired to design just such an eight-way VGA splitter, mainly because the customer wanted to avoid the $900 price of a commercial unit. My design was similar to Michael Swartzendruber's splitter in the October/November '92 issue of the *Computer Applications Journal* ("Driving Multiple VGA Monitors"), but with, perhaps, a very important addition. Mr. Swartzendruber's basic PNP emitter follower (buffer] design is shown below:



The transfer characteristic of this buffer is Vout = Vin + *0.6* volts.

Considering the fact that the typical maximum analog RGB output is about 1 volt, this 0.6-volt offset may be undesirable (it does seem to work satisfactorily, however). The horizontal and vertical sync inputs are also driven in this manner and the 0.6-volt offset becomes a main concern here because many monitors specify the sync inputs as TTL type, suggesting a maximum low input of 0.8 volts. With the input already at 0.6 volts [I actually measure over 0.7 volts!), trouble could be close. One monitor I tried wouldn't sync at all due to the offset. Fortunately, the fix is no more complicated or expensive than the original circuit. If we add an NPN emitter follower onto the output of the PNP follower, the signal if faithfully reproduced. Consider the following:

The transfer characteristic of the NPN follower is Vout = Vin – 0.6 volts. Combining both equations yields Vout = Vin, and we have a true buffer. A complete circuit is shown below:



Note the increase in the PNP emitter resistor value due to the reduced load of the NPN base. Caution: The PNP transistor must be first in line because an NPN obliterates the lower 0.6 volts of the desired signal.

To produce a nice steady DAC ramp on the analog color lines for oscilloscope inspection, draw a single horizontal line across the graphics screen, increasing in intensity (0% to 100%) from left to right. Such an image lets you easily observe offset and scaling of your analog driving circuitry.

Dale Nassar
Amite, La.

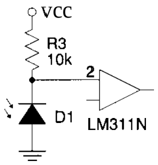
CORRECTIONS

Issue #28, Page 36, Figure 4a
Pin 1 of U6 (MAX7224) should be grounded.

Issue #28, Page 37, Figure 4b
The zener diode (D3) should be labeled 5.1 V.

Issue #29, Page 24, Figure 2
The NPN transistor, Q12, should be labeled 2N4401.

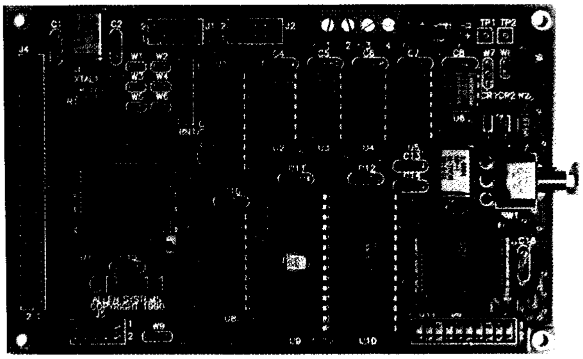
Issue #29, Page 70, Figure 4
The photodiode and resistor should be connected as follows:

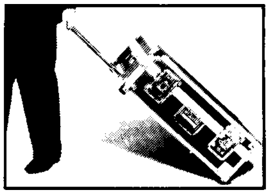


We Want to Hear from You

We encourage our readers to write letters of praise, condemnation, or suggestion to the editors of the Computer Applications Journal. Send them to:

The Computer Applications Journal
letters to the Editor
4 Park Street
Vernon, CT 06066

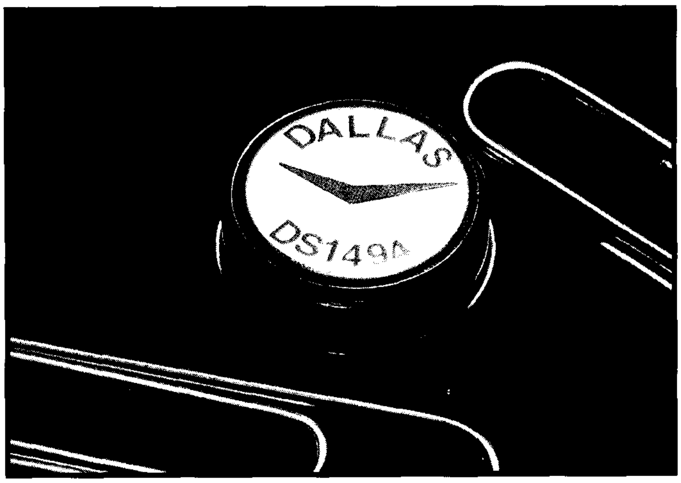




NEW PRODUCT NEWS

Edited by Harv Weiner

MINIATURE TIMER



The DS1494 Time-in-a-Can from Dallas Semiconductor serves as a real-time clock for a computer or as an add-on, run-time meter that tracks the time a system is in use.

You can install the DS1494 in a matter of minutes. The user need only make one connection to logic voltage (+5 volts) and the other to ground. System software can remain unchanged because you can add the device as a fully independent observer of the system's activity.

Another feature available in this add-on mode is a cycle counter that counts the number of times the system has been turned on and off. The DS1494 measures the amount of time the system is off by subtracting the on-time from the real time. A unique 48-bit serial number identifies the system. The chip also contains 4,096 bits of nonvolatile SRAM .

With Dallas Semiconductor's new single-wire protocol (plus ground), communication takes place through the lid of the sealed can. Data transfer occurs at 16 kilobits per second in the same manner as Morse Code, with long and short pulses representing 1s and 0s. This single-wire protocol also simplifies attachment to the printed circuit board and facilitates surface mounting.

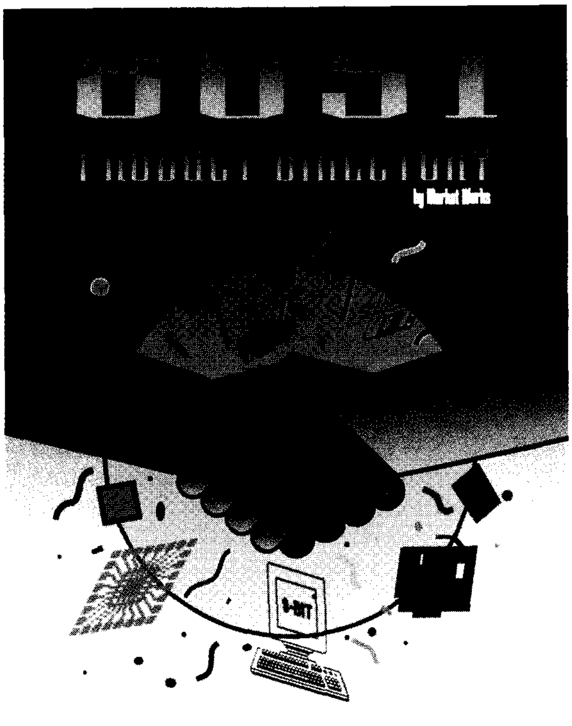
You can also design the DS1494 into the system software to provide a full range of timekeeping features, including a real-time clock with +2 minutes per month accuracy, an elapsed time meter, and a cycle counter.

Typical uses include a stopwatch, an alarm clock, a logbook, a time and date stamp, an hour meter, a calendar, a cycle timer, and an event scheduler. The DS1494K Time-in-a-Can Starter Kit includes the device, a PC COM port adapter, hour meter retrofit accessories, and demonstration software.

The DS1494 sells for $7.50 in quantities of 1,000. The DS1494K Time-in-a-Can Starter Kit sells for $25.

Dallas Semiconductor
4401 S. Beltwood Pkwy.
Dallas, TX 75244-3219
(214) 450-0448

#500



8051 PRODUCT DIRECTORY

The new 8051 Product Directory published by Market Works lists more than 65 suppliers of hundreds of 8051 products, such as chips, boards, emulators, compilers, debuggers, and real-time kernels. The Directory contains datasheets for each product and provides company names, addresses, phone numbers, fax numbers, product performance information, prices, and ordering information. It also features cross-reference guides, sources of information, and distribution locations useful to 8051 developers, design engineers, and purchasing managers.

Market Works sells the 8051 Directory for $24.

Market Works
4040 Moorpark Ave., Ste. 203
San Jose, CA 95117
(408) 261-3333
Fax: (408) 261-3336

#501

NEW PRODUCT NEWS

ROYALTY-FREE EMBEDDED BIOS

General Software is offering an Embedded **BIOS Adaptation Kit** that enables embedded system developers to manufacture their own **ROM BIOS,** and to customize it to meet the special needs of their embedded hardware. It is licensed royalty-free for increased flexibility and reduced product cost.

Embedded BIOS comes with over 15,000 lines of assembly language source code in modular units. More than 30 configuration options can tailor the BIOS to your hardware while retaining IBM PC BIOS compatibility. A complete set of ROM building utilities, a ROM disk BIOS extension module, remote disk software, and General Software's BIOS-aware debugger are also included.

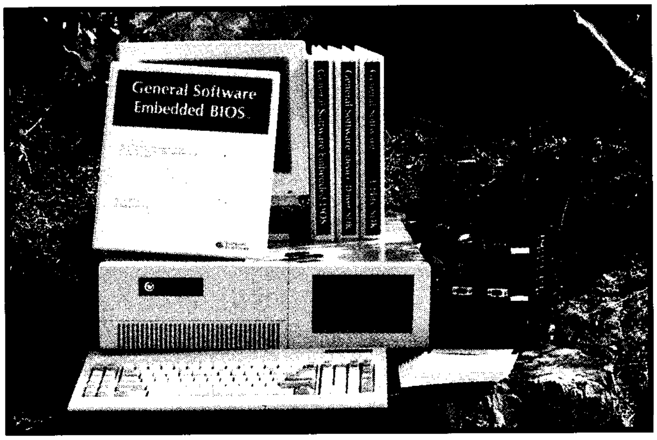
The new BIOS provides special support for on-chip peripherals of the 80186 family and includes programming chip selects, on-chip timers, an on-chip interrupt controller, and other functions. You can also configure it to a standard AT or a hybrid platform employing both on-chip and external peripherals. The BIOS supports the same basic software interrupts as a PC or XT and retains the same hard-coded entry points, so ROM extensions on video adapters, which call those entry points directly, continue to function correctly.

The primary focus of Embedded BIOS is the support of AT-compatible BIOS functions in a real-time environment with a low interrupt latency of less than ten instructions. The BIOS also offers full reentry when used in conjunction with the company's Embedded DOS operating system.

The Embedded BIOS Adaptation Kit sells for $350 plus shipping.

General Software
P.O. Box 2571
Redmond, WA 98073
(206) 391-4285
Fax: (206) 746-4655

#502



RS-232 DATA ACQUISITION SOFTWARE

T.A.L. Enterprises' new software reads real-time data from any RS-232 device directly into any PC application. The Software Wedge captures data from your PC's serial port, custom tailors it to your specifications, and then transfers the data to any application you specify either by sending keystrokes to the application or by dynamic data exchange. The data from your serial device appears to any application as if you had typed it in on the keyboard.

The Software Wedge is fully interrupt driven and operates at all standard data rates up to 19,200 bps. It supports two-way serial communications, input data parsing and filtering, intelligent keystroke macro insertion, date and time stamping, automatic receive data acknowledgment, and full input translation table. To avoid losing data, Software Wedge buffers all input data and transfers it to your application programs only when they are ready. The program supports all common flow control protocols.

Both DOS and Windows versions of the program are available. The Windows version provides full support for dynamic data exchange, including a powerful set of DDE commands that allow other applications to take complete control of all Software Wedge functions. The DOS version is a removable RAM program that occupies only 5K; it may be loaded into the upper memory area.

The Software Wedge will run on any PC compatible (DOS version 2.0 or higher or Windows 3.x) with any PC application program. You may use it with any RS-232–compatible device, including bar code readers, scales, modems, industrial, laboratory and measuring instruments, and so forth. Bar code readers must have internal decode and RS-232 output.

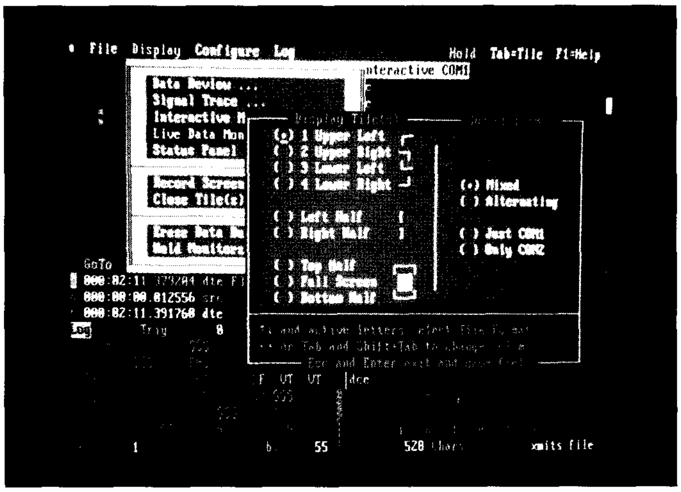
The Software Wedge DOS version sells for **$129,** and the Windows version sells for $199.

T.A.L. Enterprises
2022 Wallace St. • Philadelphia, PA 19130
(215) **763-2620** • Fax: (215) 763-9711

#503

NEW PRODUCT NEWS

COMMUNICATIONS DEBUGGING SOFTWARE



```
000:02:11 379204 dte F
000:00:00.012556 src
000:02:11.391760 dte
```

Paladin Software Inc. is shipping a new version of **MicroTAP** (formerly DataScope), a powerful communications, debugging, data capturing, and analytical tool. By allowing the user to apply powerful capture, display, and search tools to ordinarily invisible serial transmissions, MicroTAP is an alternative to expensive hardware line monitors.

MicroTAP is the only serial line monitor to include context-sensitive Hypertext with cross-links and an index, Hypersetup, up to four user-alterable multitasking window displays, and oscilloscopelike signal event tracing. All data and signal events are time-stamped to the microsecond and feature twin history cursors to provide absolute or relative information as well as the time difference.

Version 2.1 also adds CUA compliant file management, signal pattern triggering, and expanded data stream triggering capabilities, which include logical AND and OR functions, user-selectable source streams, and wild card bytes in binary trigger strings, alphanumeric trigger strings, or both.

Other features include an 8-MB log capacity, dual simultaneous data review displays, macro display recording, and snapshot disk logging. MicroTAP supports all COM ports at all possible baud rates and all combinations of word length, parity, stop bits, and output control lines.

MicroTAP sells for $299 and includes cable, connectors, and a manual.

Paladin Software, Inc.
3945 Kenosha Ave. • San Diego, CA 92117
(619) 490-0368 • Fax: (619) 490-0177

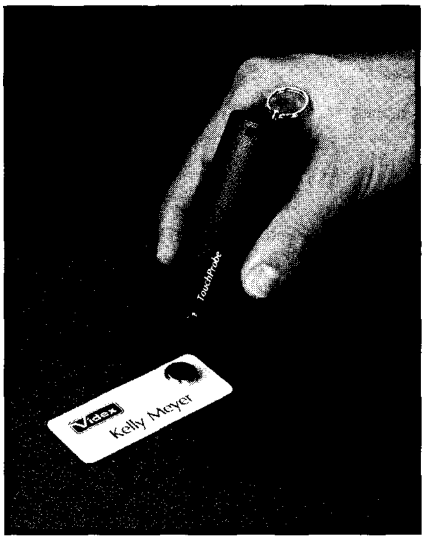
#504

PORTABLE TOUCH-BUTTON READER

The **TouchProbe** is about the size and weight of a small pocket flashlight and reads information from *touch buttons.* These buttons are memory chips housed in small, water-resistant, stainless steel cases and work like electronic labels. TouchProbe accesses these buttons with a simple contact to read information or store data. Each button contains a unique 48-bit serial number.

The standard TouchProbe model features a cast metal case, a real-time internal clock, and 128K of internal memory. Its lithium batteries require no recharging and last more than five years or 350,000 reads. The TouchProbe is useful in areas where environmental conditions prevent the use of bar codes.

The TouchProbe measures 4.5" x 1.5" x 0.8" and weighs less than 6 ounces. It features an LED flash after a successful read and tone patterns to give the user information about the reader's operations. A real-time internal clock provides a time- and date-stamped record of every transaction. The TouchProbe can store over 5,000 reads in its internal memory before downloading to a computer.



The communications software, available for MS-DOS and Macintosh, configures the TouchProbe and touch buttons and downloads data to the computer as an ASCII text file. You can easily import this file to existing applications and to most database and spreadsheet programs. The TouchProbe Recharger-Downloader Station uses a standard RS-232 serial port to communicate with a variety of computers.

The TouchProbe sells for $298 with 32K of internal memory and $395 with 128K of internal memory. Read-only buttons are $3.15 each and read/write buttons are $15 each. The Downloader sells for $149, and the Communications software sells for $100.

Videx
1105 N.E. Circle Blvd.
Corvallis, OR 97330-4285
(503) 758-0521
Fax: (503) 752-5285

#505

NEW PRODUCT NEWS

SIMULATOR FOR 80C196KC CONTROLLER

Lear Corn Company recently announced the release of SIM196KC, a new full-screen simulator-debugger for the Intel 80C196KC family of embedded controllers. The simulator is fully interactive, and the user has access to all the internal registers, timers, counters, data memory areas, control flags, and so forth presented in various display screens by typing unambiguous commands through the keyboard.

A quasi stand-alone full disassembler is standard, and the complete interrupt system common to the 80C196KB is fully supported as well as serial communications simulation. Function keys simulate incoming signals to the HSI and special interrupt pins and act as debounced switches when pressed. You can program incoming signals and serial data to occur at user-defined intervals in terms of CPU T-states.

The new and unique Vertical Windows scheme, which makes it possible to use the additional 256 bytes of internal RAM as registers, is also fully emulated. Another powerful fully supported new feature of the 80C196KC, known as the Peripheral Transaction Server (PTS), offers a DMA-like generation of block transfers, A/D conversions, and HSI and HSO operations under a faster more code-efficient alternative to the standard interrupt system.

The 80C196KC allows the user to test and debug programs written for the 80C96 and 80C196KB because it is a superset of the MCS-96 family. The simulator fully supports all the new instructions available only in the KC version.

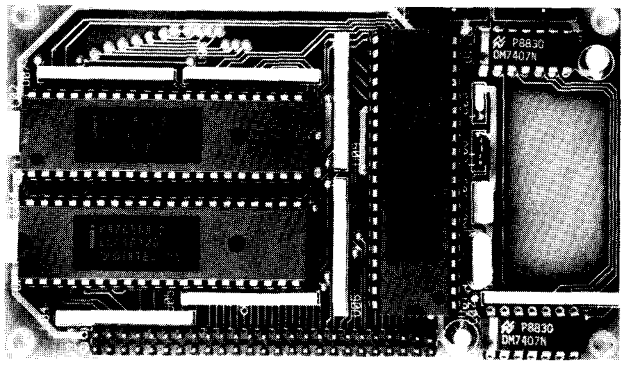
The SIM196KC sells for $400 by itself and for $450 if the MCS-96 cross-assembler is included as a package. The price includes the software with unlimited upgrade privileges and a comprehensive user's manual.

Lear Corn Company
2440 Kipling St., Ste. 206
Lakewood, CO 80215
(303) 232-2226
Fax: (303) 232-8721

#506

NEW PRODUCT NEWS

CUSTOM KEYBOARD AND KEYPAD ENCODERS



Vetra Systems Corporation announces a family of PC Keyboard Encoders that simplify the design of PC-based systems accepting operator input from custom switches and control panels. The Encoders accept up to 48 discrete switches or 128 matrix-connected switches and convert contact closures to scan codes which are compatible with the keyboard ports of IBM PC/XT/AT or compatibles.

In addition to standard key codes, designers can specify custom key codes, such as multiple-key combinations from one switch closure. This feature allows easy implementation of key sequences, such as Alt+key, Ctrl+key, and so forth. You can use a standard keyboard during development and replace it later with the custom switch plus Encoder combination, with complete transparency to the application software.

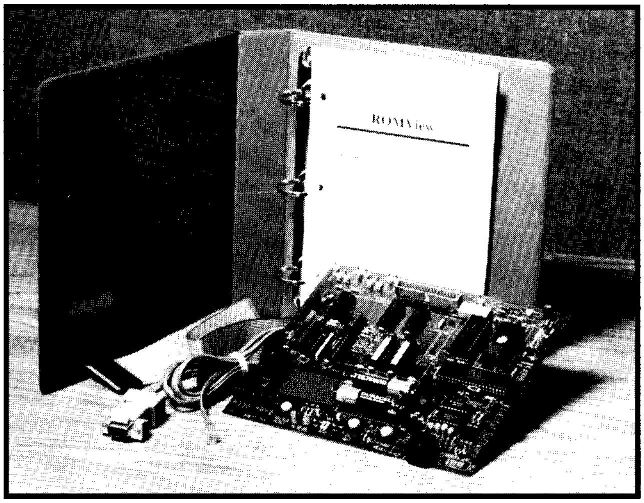
Vetra also offers a *Smart-Wye* model which accepts a standard PC keyboard in addition to the custom switches. The scan codes cleanly merge with the scan codes from the custom switches, permitting simultaneous use of a full keyboard. The Encoders are contained on a 2.6" x 4.6" printed circuit board and are powered from the PC keyboard port so require no external power supply.

You can use these Encoders in embedded and dedicated PC applications, such as intelligent systems, instruments controlled by a PC, industrial control, communication controllers, production control, and test systems.

The Encoder with 24 discrete switch capacity sells for less than $85 in quantities of 50 or more.

Vetra Systems Corp.
27 **Newtown** Rd. **.** Plainview, NY 11803
(516) 454-6469 • Fax: (516) 454-I 648 **#507**

NEW PRODUCT NEWS



REMOTE DEBUG SUPPORT TOOL

Datalight has released a new remote debugging tool, ROMView. ROMView enables developers to start remote debugging on a stand-alone target system quickly, without tying up target system serial ports or RAM.

ROMView has a preconfigured remote debugging kernel in ROM that works with either Borland's Turbo Debugger or Datalight's CodeView-like debugger. Connecting ROMView involves plugging a pod into the target system ROM socket and attaching ROMView to the memory Write line. It also connects to a PC via serial port.

ROMView maps ROM, RAM, and several I/O devices into the ROM socket space on a target system. It maps the RAM into the lower addresses and the ROM into the higher addresses, so it can hold the boot software. It also maps device control registers into the ROM area. These registers manage an 8250 UART, switches, and LEDs. The switches and LEDs allow low-level program I/O during testing.

ROMView comes with a ROM that includes the Turbo Debugger remote kernel and Datalight's RDEB remote kernel. The ROM also includes self-test software to verify that ROMView is functional. The source code for configuring the kernel is also included.

ROMView sells for $395 and comes with the ROMView circuit board, a disk containing source code, cables, and a complete user's manual.

Datalight
307 N. Olympic Ave., Ste. 201 • Arlington, WA 98223
(206) 435-8086 . Fax: (206) 435-0253 **#508**

FEATURES

DESIGN CONTEST WINNER

FEATURE ARTICLE

Sanjaya Vatuk

Build the SmartRom EPROM Emulator

If you feel like you're not swatting program bugs fast enough with erase-and-burn firmware development, try this EPROM emulator. A debug session is just a download away.

Grind! Burn! Crash! Zap! If these sounds are familiar, you are probably doing microcontroller development the traditional way: assembling source code, burning it into an EPROM, and trying to figure out why your gizmo turns left instead of right. Only after erasing the EPROM are you finally ready to start the cycle all over again.

The ideal solution to this problem is an in-circuit emulator, but its cost is usually beyond the means of the average hobbyist or independent developer. An inexpensive alternative is an EPROM emulator, which offers a way to download code to almost any microprocessor without the time penalties imposed by the burn and erase routines. Because changes can be tried rapidly, you are encouraged to make incremental firmware improvements, ultimately resulting in tighter, essentially bulletproof programs.

Although there are several commercial emulators on the market, none seemed to offer the combination of features and flexibility I wanted to implement. These criteria included speed, host independence, and a friendly user interface.

The result is SmartROM, a low-cost serial EPROM emulator with on-board intelligence. In addition to emulating a range of standard EPROM devices from the 2732 to the 27256, SmartROM can mimic the Intel 87C64, an EPROM featuring on-board address latches for use with micros having a multiplexed data and address bus. You may use virtually any system with an RS-232 port as a host to download object files. SmartROM accepts data at standard baud rates up to 115,200 bps, making downloads a quick and painless process. Currently supported 8-bit object file download formats include Intel Hex, Motorola S19, MOS Technology Hex, and Tektronix Hex. All configuration is done via front-panel push buttons, and LEDs provide visual confirmation of status.

There are four basic hardware requirements in any EPROM emulator design, foremost being some type of reprogrammable memory. With the price of 32K static RAM chips dropping well below $10, a single 62256 (or 43256) seemed to be a natural fit for all but the largest of embedded controller applications. The second requirement is a means of loading this RAM with the EPROM image. The on-board UART and processing power of the 803 1 made it a logical choice because host independence was a primary goal. Next, you must have an interface to the target that resembles a real EPROM as closely as possible. Finally, isolation is needed between the host load circuitry and the target emulation RAM to prevent conflicts with the target system. This last requirement led to an interesting discovery.

A HARDWARE TRICK

At first glance, the main part of the circuit seems to be just another 803 1 with some memory and glue (see Figure 1). The first important difference is the processor. It must be CMOS, either an 80C3 1 or 80C32, because there is no way to tristate the buses on an 8031, which makes isolating it from the emulator RAM without additional components impossible. A little digging in the data book revealed the CMOS version can be effectively tristated by placing it in the software-controlled pcwer-down mode.

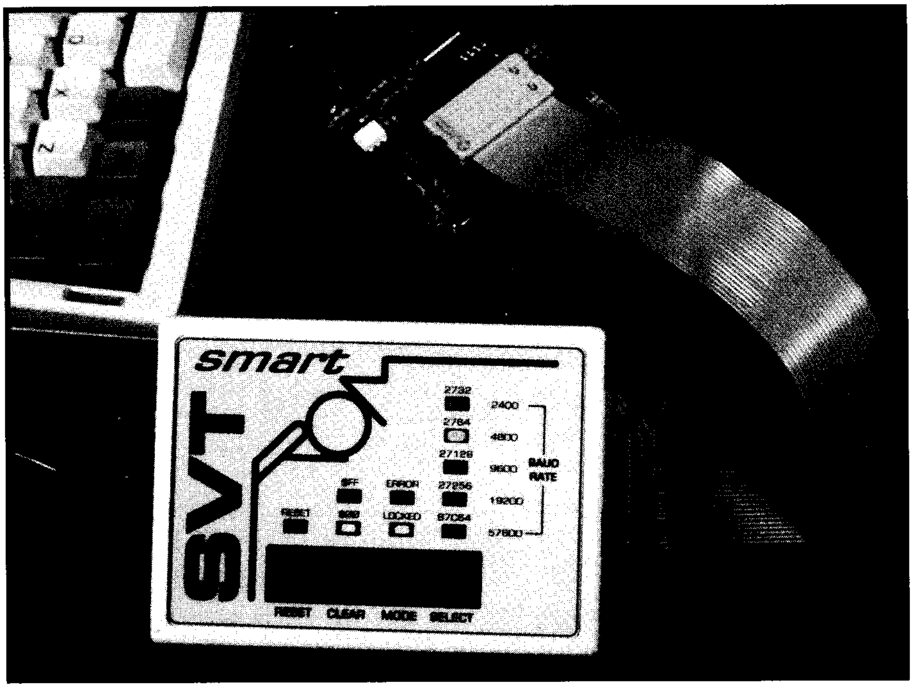
Understanding how this arrangement can be made requires a review of the normal operation of the 80C31. During program execution from an external EPROM, Port 0 is the multiplexed lower address and data bus and Port 2 is the upper address bus. Port O’s functionality is lost in this mode, so data written to the Port 0 Special Function Register (SFR) will never appear at the port pins when using external program memory. Data written to the Port 2 SFR will appear at the port only when an access to external RAM is made using one of the MOVX @Ri opcodes.

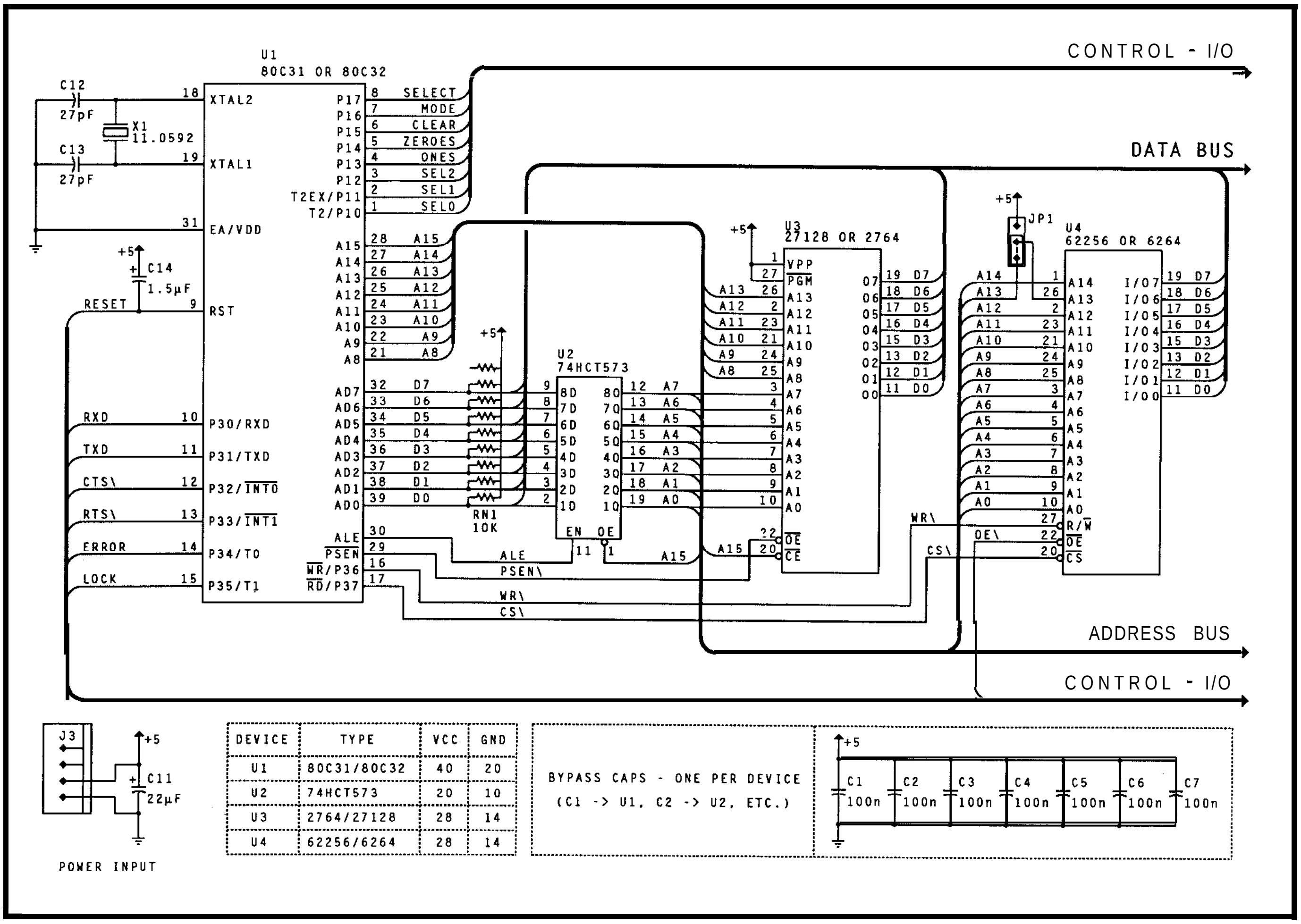
The oscillator is stopped when power down is invoked by firmware, putting the CPU and all on-chip peripherals to sleep. (The only exit from power down is a hardware reset.) Internal RAM contents are unaffected as long as VCC is maintained, but power consumption is reduced to a few microamps of leakage current.

More importantly, Port 0 now enters a high-impedance state, and the data in the Port 2 SFR appear on the upper address lines. If this value is all ones ($FF), A8–A15 are pulled weakly high, essentially the same as tristating. ALE and * PSEN are forced low, making it necessary to disable the outputs of the address latch U2 and the program EPROM U3 in order to completely relinquish the buses

Al5 is used as the control line for this purpose-in fact, it is the key to the entire design! The firmware writes $FF to Port 2 before executing the power-down command, but Al5 doesn’t actually go high until the processor has gone to sleep. Al5 is connected to the output enables of the latch and the EPROM, tristating both devices upon going high. No other port pin on the 80C31 could provide this control function because the program would hang just as soon as the port pin was written. The downside to this approach is SmartROM is limited to 32K of RAM because Al5 can never be permitted to go high while the 80C3 1 is running.

Another departure from standard 803 1 designs is the buffer RAM access method. Normally, the RAM output enable (• OE) would be connected to P3.7 (the 80C31*RD signal), with the RAM chip select (‘CS) tied to ground. Because the 80C31 must share access to the RAM with the target micro, both signals need to be enabled upon entering power down. This condition is satisfied by using P3.7 as ● CS and the complement of Al5 as *OE. The RAM outputs are enabled when Al5 goes high, assuming that ‘CS has first been written low.





DEVICE	TYPE	VCC	GND
U1	80C31/80C32	40	20
U2	74HCT573	20	10
U3	2764/27128	28	14
U4	62256/6264	28	14

Figure 1— *The design of the EPROM emulator requires a CMOS processor, the 80C31 or 80C32.* Memory size *supported for emulation is limited to 32K because address line A 15 is used in selecting host or target mode.*

Although this method works when emulating, it cannot be used for reads by the 80C31 because bringing Al5 high during normal operation invites a trip into the weeds. A way around this problem exists, but first a look at the target interface circuitry is in order.

RIGHT ON TARGET

The target interface consists of data bus buffer U5 and a pair of PALs (U6 and U7) acting as address buffers or latches (see Figure 2). These chips serve to isolate the target from the remainder of the emulator load circuitry, connecting to a target EPROM socket via a short ribbon cable through header J1.

The Port 1 control lines labeled SELO, SEL1, and SEL2 solve the problem of RAM reads by the 80C3 1. U7 decodes these lines with A15 to supply the RAM *OE signal. If all three select lines are high, or if A15 is high, *OE will be low, enabling the RAM outputs. In order to read from RAM, the 80C3 1 must first bring SELO, SEL1, and SEL2 high while ● CS is high, then pulse ● CS low by performing a normal read. When writing, it must ensure these lines are not all high before bringing *CS low to select the RAM, followed by a normal write.

The select lines also determine which type of EPROM is to be emulated. When SEL2 is low, SELO and SEL1 are decoded to select 2732, 2764, 27128, or 27256 EPROMs by forcing unused address lines low as required. In these modes, the PALs are configured as gated buffers. For example, if the target socket is meant to hold a 2764, address lines Al3 and Al4 to the emulator RAM are held low regardless of the state of target pins XA13 and XA14. With a 27256 selected, all address lines pass through unchanged.

When SEL2 is high, the PALs are both configured as transparent latches, similar to a 74LS373. Input *XCE from the target socket (which will normally be ALE from the target processor] latches the state of address lines XAO through XA12 on its falling edge. Al3 and Al4 are forced low, and the emulator will now act like an 87C64 latched address EPROM. SELO and SEL1 are low when emulating the 87C64.

A 74HCT541 was chosen as the data bus buffer because it has two output enables corresponding to an EPROM's chip select and output enable. This buffer acts very much like the output drivers of a real EPROM because the RAM outputs are held active when emulating. Both *XCE and *XOE (from the target socket) must be low to read from the emulator RAM. ● XCE is gated by Al5 to provide *GXCE, ensuring the '541 outputs float when not emulating. The resistors on the outputs provide stability when driving capacitive loads.

SmartROM is socketed for 32K or 8K of RAM, either standard or non-volatile. JP1 is used to select the correct type. The emulator cannot be configured as a 27128 or 27256 with a 6264 installed.

An external power supply capable of providing 5 volts ±5% at 500 mA (max) is required for operation. Despite the use of the power-down mode of the 80C31, the emulator actually uses more power when emulating than when the download processor is awake, due primarily to the PALs in the target interface. No power is drawn from the target system, but some leakage current may be drawn from the emulator when the target is powered off.

A piggyback board contains the user interface and RS-232 converter, a MAX232 (see Figure 3). Q1 is driven by *A15 to halt the target system while the emulator is being loaded. This output may be connected to almost any microprocessor reset input where a switch would be appropriate. If the target reset is active low, connect the emitter of Q1 to ground and the collector to the reset input. For an active-high reset, connect the collector to target VCC and the emitter to reset. Three momentary switches are used to configure SmartROM, while a fourth resets the 80C31. U10 decodes SEL0, SEL1, and SEL2, driving LEDs to show the type of EPROM being emulated. U8 drives more LEDs to indicate the status of the emulator. The switches and LEDs perform multiple functions depending on what mode the emulator is in, which brings me to..

USING SMARTROM

The basic operation of SmartROM is straightforward. The LEDs flash briefly at power up, and the emulator RAM is sized and tested. Each test illuminates one of the device LEDs. Assuming all is well, the RAM is filled with $FF, the "Ones" and "2764" LEDs are illuminated, and the emulator is ready to accept a download at 9600 bps.

If an error is detected during a RAM test, it will halt and the "Error" LED will flash. At this point the emulator is about as useful as a pet rock, although pressing the Mode key will permit reading the emulator RAM using an EPROM programmer.

The Clear key clears the buffer RAM and readies the emulator for downloads. If RAM is already "blank," pressing Clear will fill it with the complement of the current value, illuminating the "Ones" or "Zeros" LED as appropriate. (Certain Motorola MCUs with on-board EPROM are erased to all zeros. SmartROM may be used as a master PROM for correctly programming these devices.]

The Select switch is normally used to choose the device being emulated. Operation is disgustingly intuitive: push the button and the LEDs cycle through the various EPROM types. The "27128" and "27256" options will be skipped when a 6264 is installed in the RAM socket.

As long as the emulator is in the cleared state, pressing Mode will allow selection of baud rate. The "Ones," "Zeros," "Error," and "Lock" LEDs will flash in a somewhat circular sequence, and the device LEDs will then show the current speed. The Select key advances through the available baud rates.

A sixth selection (no device LEDs illuminated) provides 1200-bps operation with a standard 80C31 installed, or 115,200 bps using an optional 80C32. Timer 2, which is present only in the 80C32, has greater resolution than Timer 1, allowing a wider selection of baud rates. Both Timer 1 and Timer 2 are configured by the firmware as baud-rate clocks. If Timer 2 exists, it becomes the baud clock automatically, while Timer 1 is ignored. Otherwise Timer 1 is used, with no firmware changes necessary. I would love to take credit for this idea, but it's just the way Intel implemented the extra timer.

Pressing Clear exits the baud-setting mode, and the device LEDs revert to displaying the EPROM type. SmartROM should now be ready for the current development session. It is not necessary to set the baud rate and other parameters every time a new file is downloaded because all of the configuration settings described above are saved in internal RAM and recalled when the emulator is brought out of power down by pressing Reset.

Downloading an object file is simplicity itself. On an IBM DOS-type machine, enter the command line copy /b my_gizmo.obj coml (assuming you are using coml}. Using the /b (binary) switch, especially at higher data rates, speeds things up two to three times. The "Ones" and "Zeros" LEDs will flicker during the download, providing visual feedback.

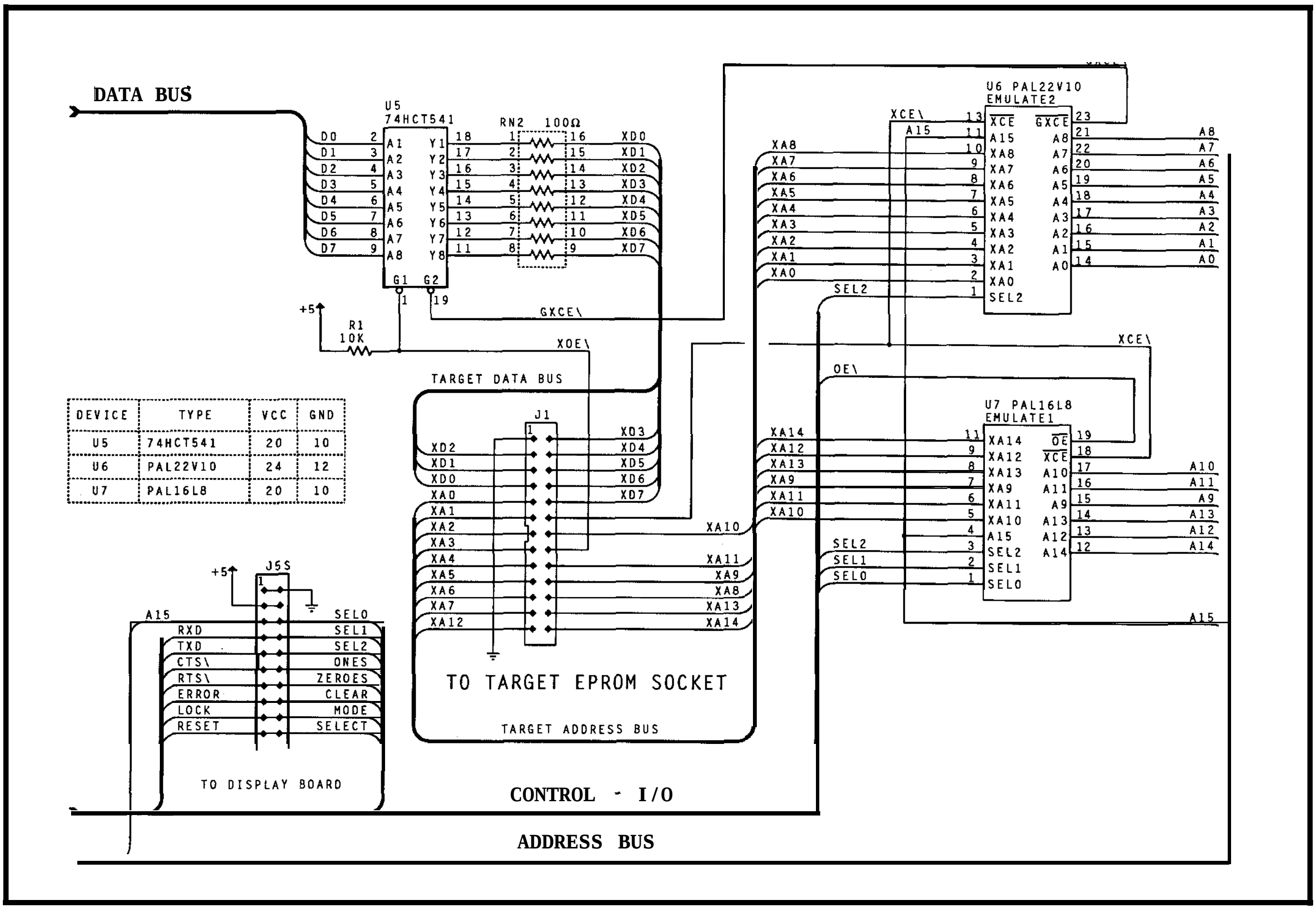
At 115,200 bps, an Intel Hex file that fills all 32K takes about 8 seconds on my 12-MHz '286 clone. Your mileage may vary. SmartROM does not require any delays or handshaking between incoming bytes, regardless of the baud rate in use. (I have included several small .COM programs for setting rates faster than the 9600 bps allowed by MS-DOS. These should work on most serial cards, although XTs typically cannot exceed 57,600 bps.) SmartROM will happily accept data in 8N1,7N2, or 7P1 formats, although it ignores any parity bit.

The "Lock" LED should be lit continuously after a successful download, indicating that SmartROM has automatically begun emulating. The target system will begin execution of the downloaded code, and you may start tearing your hair out. A touch of the Reset button will wake up SmartROM and halt the target processor. The "Lock" LED now flashes rapidly, evidence that the emulator RAM still contains the last download. Pressing Mode will restart the target, and pushing Clear will fill the RAM with the previously chosen blank value and prepare the emulator for a new download.

The "Error" LED starts flashing if a download error occurs. Rapid flashing indicates a checksum error, and medium flashing reports that the download was incomplete. Slow flashing means an invalid record type was detected. If you want to troubleshoot, the emulator can be forced into power-down mode using the Mode key. In this case, both the "Lock" and "Error" LEDs will be illuminated. Pressing Clear will enable a retry. Most errors are caused by mismatched baud rates or faulty cables.

A cold boot can be forced by holding the Clear key down while the emulator is coming out of reset. All configuration settings will be returned to their default values.

SmartROM can store EPROM images even when powered off if a nonvolatile RAM module such as the Dallas DS1235 is installed. However, RAM is normally cleared upon power up, so a means for preserving this data is needed. Holding down the Mode key during power up (before the RAM tests begin!) fakes a warm boot. The baud rate, device type, and buffer-clear value are set to their defaults, but the RAM is not cleared or tested. The "Lock" LED flashes rapidly, and operation resumes just as if the Reset button had been pressed after a successful download. The emulator also checks the battery condition, illuminating the "Error" LED when the battery in a



DEVICE	TYPE	VCC	GND
U5	74HCT541	20	10
U6	PAL22V10	24	12
U7	PAL16L8	20	10

Figure 2—*A 74HCT541 was chosen as a data bus buffer for fhe target interface because if acts very much like the output drivers of a real EPROM*

Dallas NVRAM module has failed. Normal operation is not affected by this failure.

BARE METAL

Processing a continuous data stream at 115,000 bps cannot be considered a trivial task once you realize that a new byte can arrive every 80 machine cycles. The firmware for SmartROM is highly modular, relying heavily on interrupts and status flags.

Once the power-up test subroutines have finished, the main loop simply waits for a key press or a download. *CTS to the host is asserted and serial interrupts are enabled. Pressing Clear or Mode buttons temporarily disables the host interface while the RAM is cleared or the baud rate changed. When the action initiated by the key press is finished, control returns to the loop where the host and serial interrupts are again enabled.

Each byte of a download triggers a serial port interrupt, setting a flag (RXIP) **to** alert the main loop that a download is in progress. After stripping off the high bit, the byte is examined to see if it matches the first character of any supported hex format. If it doesn't, RXIP is cleared and control returns to the main loop.

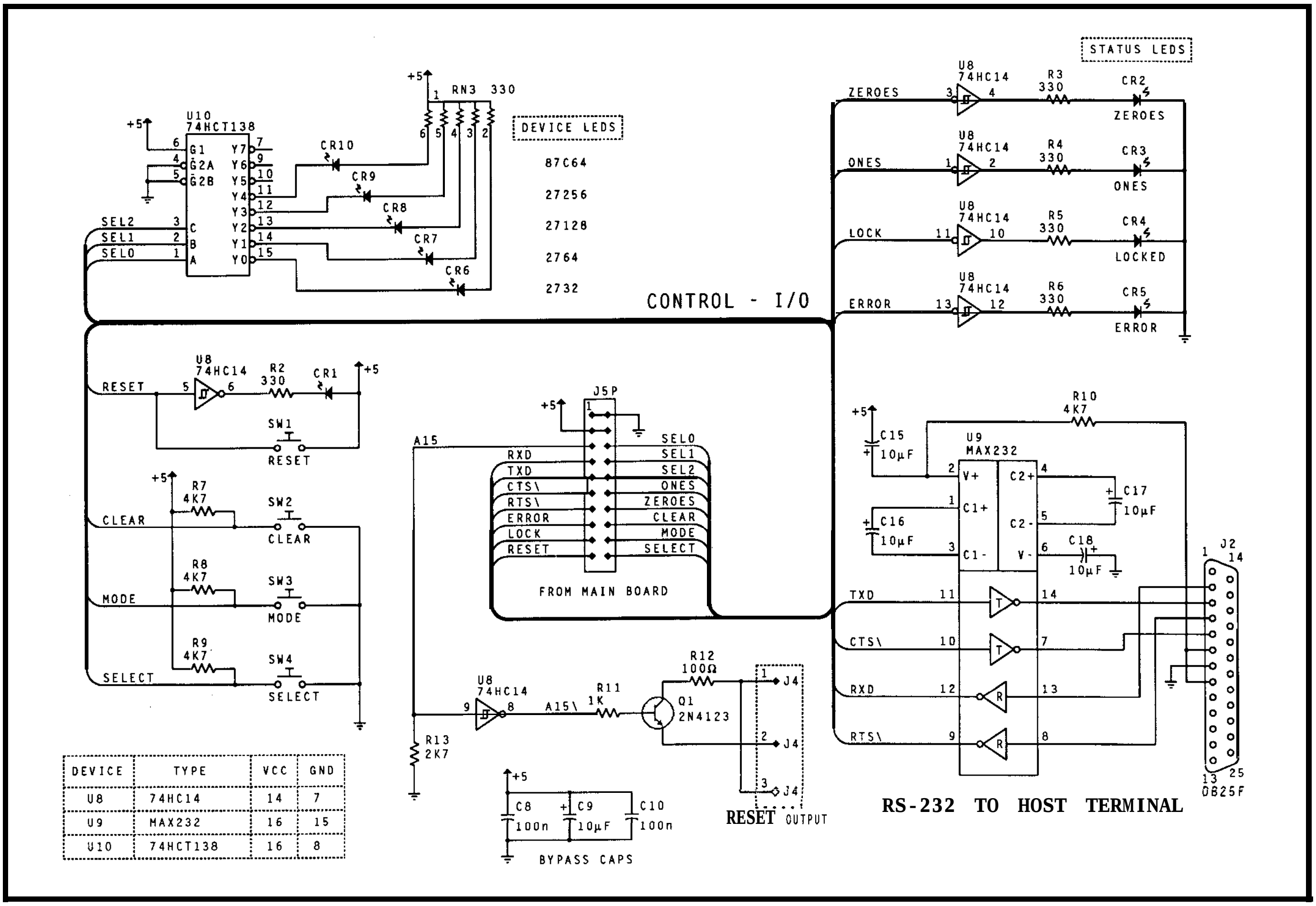
Once a match is found, RXIP remains set for the rest of the download. This flag then causes the main routine to enter a second loop, waiting for the Select key or the end of the download. The Clear and Mode switches are now ignored.

All object file formats currently supported have a similar structure, with data in ASCII form. A single line is called a record, and typically consists of a start character identifying the file type, a record-type identifier (data, null, or end-of-file), a 16-bit starting address, a data byte count, the actual data, one or two checksums, a carriage return, and an optional line feed. (For an in-depth discussion of the Intel Hex format, see "The Mystery of Intel Hex Format" by Ed Nisley, in issue #20 of the *Computer Applications Journal.*)

The receive interrupt handler uses a state machine to pass incoming characters to the appropriate conversion subroutine. A variable, STATE, is manipulated by the conversion routines to indicate the function of each byte in the record, and build an index into a jump table pointing to the requisite subroutine.

The object conversion routines actually consist of a number of independent modules, each dedicated to a particular data type. Characters needing conversion to binary are passed to ASCII-to-hex subroutines, which also perform checksum accumulation to ensure data integrity. Completed bytes are returned to the object routine for further processing. Control characters, spaces, and garbage are flagged as unusable, but generally are ignored.

The object conversion modules process each record, setting up the current EPROM address, tracking the



DEVICE	TYPE	VCC	GND
U8	74HC14	14	7
U9	MAX232	16	15
U10	74HCT138	16	8

Figure 3—*A piggyback board contains the RS-232 serial port interface for host connection and user interface for operation of the EPROM emulator via switches and LEDs.*

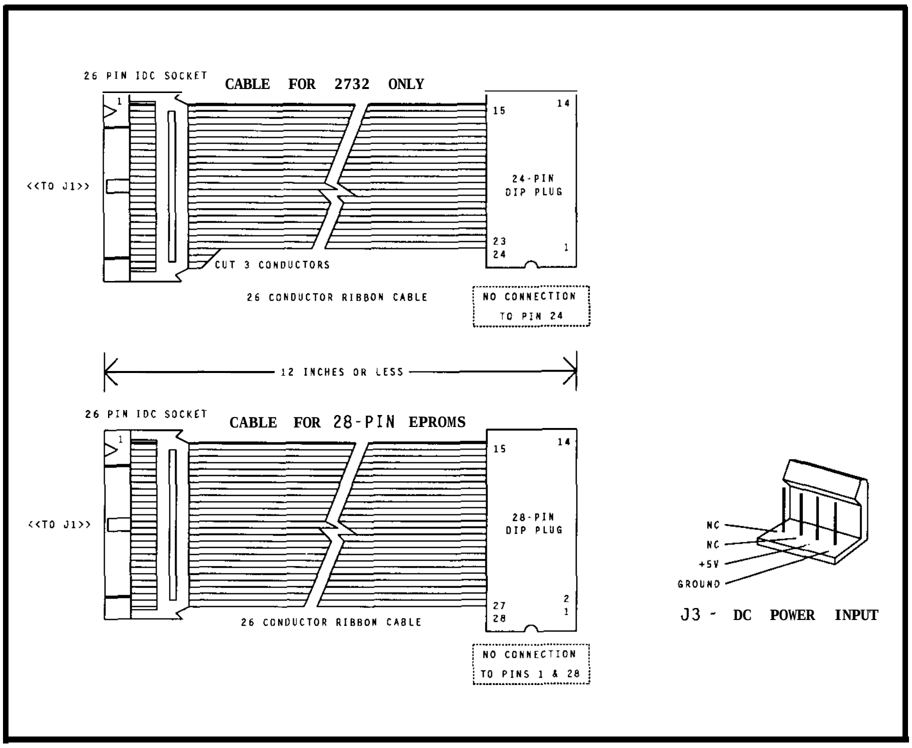


Figure 4—A *common target interface connection supports 24-pin or 28-pin EPROMs by changing interface cables.*

number of data bytes, loading the RAM, and verifying the checksum(s). If a checksum failure is detected, an error flag is set and the "Error" LED flashes rapidly.

Data destined for addresses higher than $7FFF are mapped into the available space by clearing the highest address bit because SmartROM is limited to 32K of RAM. Addresses higher than $1FFF will wrap around when only 8K of RAM is installed. No attempt is made to protect previously loaded addresses.

The STATE word is incremented each time a module completes its task, passing control to the next function upon receipt of another character. The final module resets STATE when it detects a carriage return, after which all bytes are ignored until a new start character arrives. The download is flagged DONE when the record-type field indicates an end record. The DONE flag is cleared if another start character is received, so multiple object files download sequentially.

The Timer 0 interrupt handler monitors the progress of the download, decrementing a watchdog counter. The receive handler reloads this counter each time a byte is received, preventing it from reaching zero. The watchdog counter does reach zero when 250 ms have elapsed without a character arriving at the serial port, and the Timer 0 interrupt handler flags the true end of the download. This handler also performs all housekeeping for time delays, switch debouncing, and LED flashing.

RXIP is cleared when the watchdog times out, and serial port interrupts and 'CTS are disabled, signaling the main loop that the download has ended. A time-out error is generated if the object conversion routine does not flag the end record. Unless an error was flagged, a power-down command is issued and SmartROM begins emulating. An error handler is called if any problems occur, requiring you to either clear the emulator or lock it in the emulate mode.

Given that the only way out of a power down is a reset, there must be a way of distinguishing between a power-up reset (cold boot) and a push-button reset bringing the processor out of power down. This delineation is made by dedicating four registers in the 80C31 internal RAM space as power-on status flags. The firmware knows the reset was not the result of a cold boot if these registers hold a particular set of values. The configuration settings are checked for corruption as well; a cold boot is performed if any are out of range. The LED and RAM tests are not executed during a warm boot.

The status flags are examined after a warm boot to determine whether or not the last download was successful. If it was, the "Lock" LED is flashed rapidly to show that the emulator RAM still holds valid data. The RAM buffer is cleared if any of the error flags were set during the last download.

The remainder of the firmware consists of subroutines to handle mundane activities such as checking for key presses, filling the RAM buffer, and setting the baud rate. The source file is heavily commented, so you shouldn't have too much trouble unraveling the code.

BUILDING THE PERFECT BOX

No matter how elegant a project may be, it is incomplete without suitable packaging. Squeezing all of the circuitry into a plastic enclosure measuring approximately 2.75" x 4" necessitated the use of two circuit boards. The prototype was point-to-point wired on perfboard with plated through-holes. The I/O board plugs into a female header on the main board when the box is closed, eliminating the need for any connecting cables.

Each target interface cable is terminated with a 26-pin IDC connector on one end, with a 24-pin or 28-pin DIP plug mating to the target socket (see Figure 4). The 28-pin DIP plug is probably the hardest component to locate because most manufacturers seem only to make 24 and 40-pin versions. Amp and 3M make suitable parts. I used an ordinary right-angle Molex connector for the power input.

The most challenging part of this project was modifying the enclosure. The plastic box I used is made by Unibox, and has a rather attractive textured surface. After drilling holes for the LEDs and filing out slots for the connectors, I milled a shallow indentation in the top surface to accommodate the top-panel graphics. (I discovered much too late that Serpac makes a virtually identical box with a smooth insert area already molded in!)

The basic design for the top panel overlay was roughed out using TangoPCB-Plus, then output to a PostScript file. I tweaked this file to add different fonts and center the function labels, proofing the design on a laser printer. I took the file to a desktop publishing service, where it was output to film on a Linotronic imagesetter. I handpainted the logo in red on the reverse side of the film, then spray-painted the background with white paint fading subtly into gray. The LED windows were colored with bits of translucent adhesive film. A matte transparent laminate, glued to the indentation in the box with double-sided adhesive, gave the film some protection. The result looks very similar to the polycarbonate overlays seen on many commercial products, although I have yet to find a truly rugged clear laminate for the top layer of this sandwich.

I hope you will find this device a useful addition to your collection of development tools. Having done my share of crashing and burning, I find the capabilities of this little box have spoiled me. Now if I could only find that can of Raid.. .. ▣

Sanjaya Vatuk is a field engineer for Eastman Kodak and enjoys designing MIDI and other music-related projects in his spare time.

SOFTWARE

Software and PAL source code for this article are available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

IRS

401 Very Useful
402 Moderately Useful
403 Not Useful

SOURCE

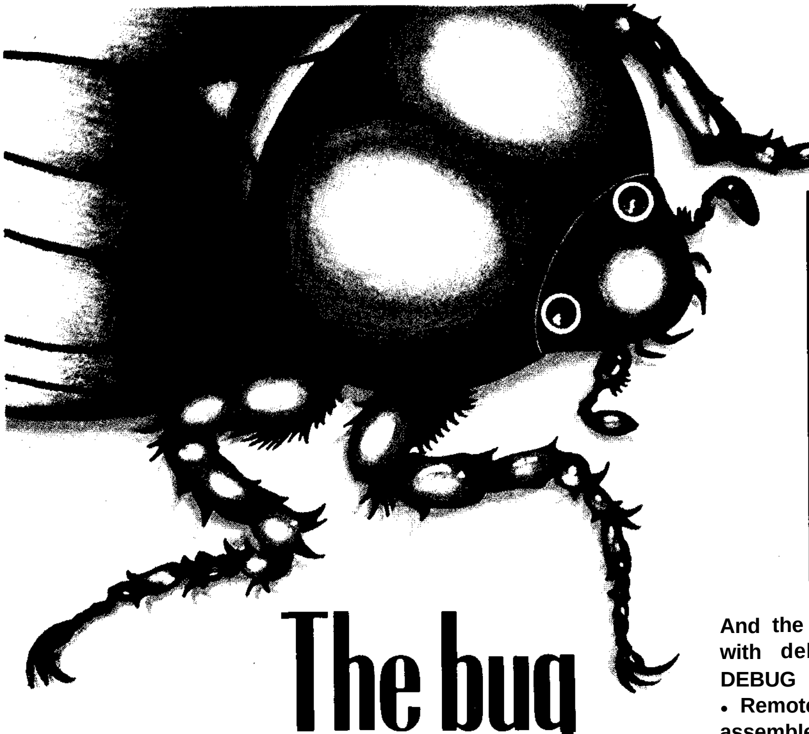
The following are available from

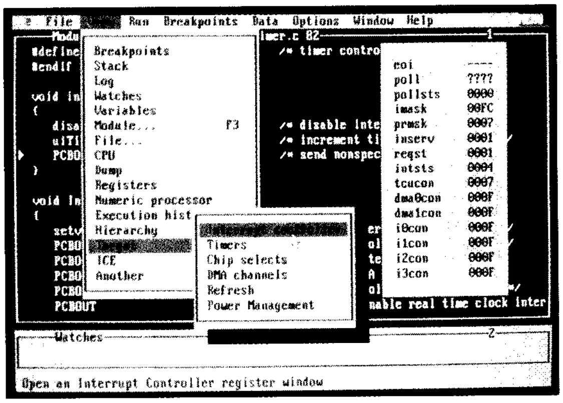
Sanjaya Vatuk
1130 S. Beloit Ave.
Forest Park, IL 60130-2306
(708) 771-1928

1. Experimenter kit consisting of two double-sided, solder-masked, and silkscreened PCBs; preprogrammed PALs and EPROM; complete user manual; source code; and communications utilities on a 5.25" PC floppy disk (3.5" available). . . . $59

2. A 26-pin IDC to 28-pin DIP plug target cable assembly 9" long. A 2" version should be specified for systems using the 87C64... **$11** (with kit), $14 (without kit)

Check or money order only. Prices include shipping.





```
≡ File       Run  Breakpoints  Data  Options  Window  Help
Modu                                   imer.c 82                        1
#define  Breakpoints                      /* timer contro
#endif   Stack                                          eoi       ----
         Log                                            poll      ????
void in  Watches                                        pollsts   0000
{        Variables                                      imask     00FC
   disa  Module...           F3           /* disable inte prmsk   0007
   uiTi  File...                          /* increment ti inserv  0001
 ▶ PCBO  CPU                              /* send nonspec reqst   0001
}        Dump                                           intsts    0004
         Registers                                      tcucon    0007
void In  Numeric processor                              dma0con   000F
{        Execution hist                                 dma1con   000F
   setv  Hierarchy                                   er i0con     000F
   PCBO                Timers                        ol i1con     000F
   PCBO  ICE           Chip selects                  te i2con     000F
   PCBO  Another       DMA channels                  A  i3con     000F
   PCBO                Refresh                       ol                  */
   PCBOUT              Power Management              nable real time clock inter
Watches                                                                 2

Open an Interrupt Controller register window
```

68705 SLUSH

FEATURE ARTICLE

Robin Brophy

Not Quite an ICE, But Just as Useful

What happens when you need a hardware-based debugging tool, but don't need the power of a full-blown In-Circuit Emulator (ICE)? You make something not quite as cool and call it a SLUSH.

As we cruised down 4th Street in his green '75 AMC Matador, which my kids dubbed the Green Slime Machine, I casually asked my dad, "Aren't you going a little fast?" Dad gave me a look reserved for backseat drivers and replied, "Maybe." When I questioned him about his actual speed he said, "How should I know, the speedometer broke last winter."

With a career in instrumentation, I can't tolerate broken or misreading gauges and meters, so I said, "Why don't you fix it?" To that he fired back, "Why don't you, smart guy!"

One thing led to another and I ended up promising to build a digital speedometer for the Slime Machine. I figured the job would amount to nothing more than a couple of magnets on the drive shaft, a Hall effect sensor, a three-digit LED display, a counter, and a timebase. I was sure others had done it a million times before and probably used only three or four chips tops. Then I realized that this project was the perfect opportunity to have a custom IC made, one of those ASICs I kept reading about. The chip would simply connect to a sensor and directly drive a multiplexed, seven-segment LED display.

I tried to implement my plan, but I hit a roadblock. When you're a one-person firm, chip dealers don't want to know you. Perhaps you know the story. You call a company about a new chip, and the first thing they ask is, "-and what company are you with?. . What is your annual projected usage!"

I started looking into microcontrollers and came upon the Motorola MC68705P3S. This chip comes in a 28-pin DIP, has an on-chip RAM, an EPROM, a timer, interrupts, and 20 lines of I/O, eight of which are buffered to drive LEDs. Bingo! This chip was the one to make a true single-chip speedometer for the Slime Machine.

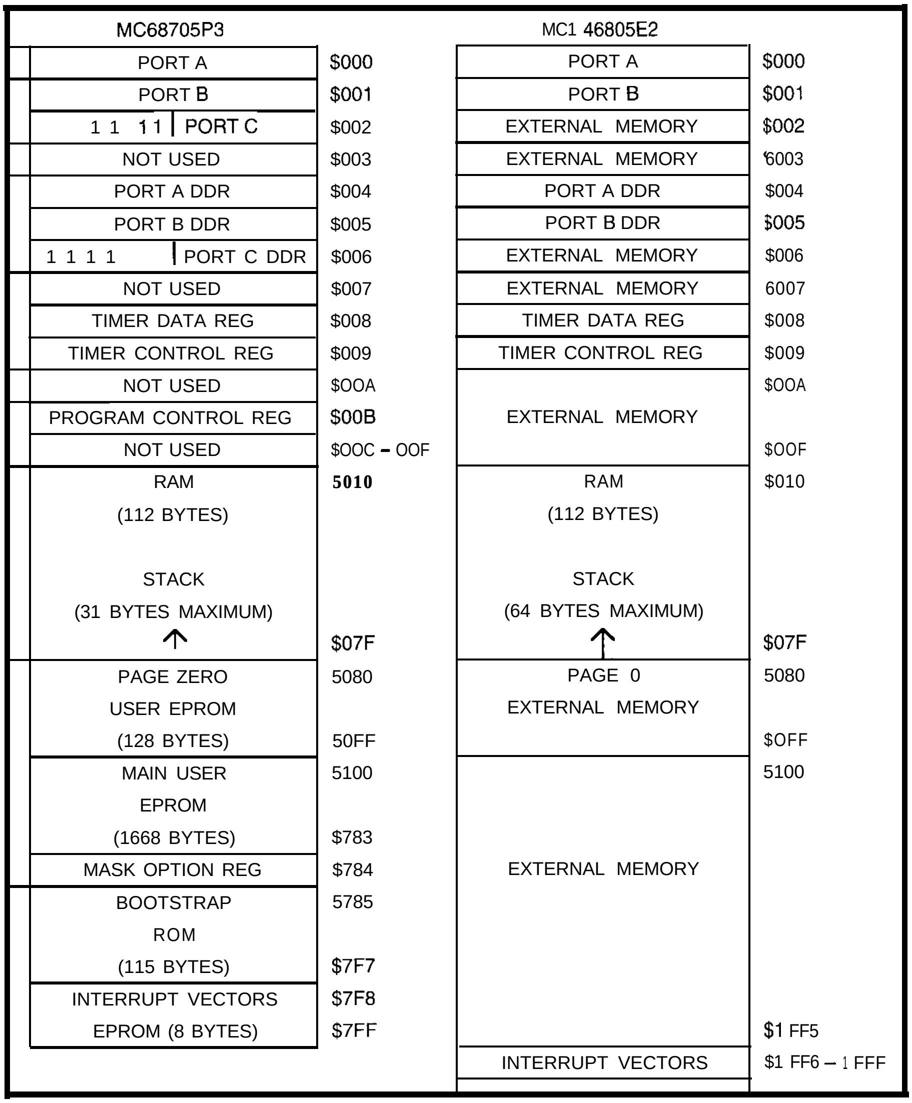
So I sent away for a couple. My motto is: "Always order two when you try something new." I added a crystal, a capacitor for reset, three transistors for digit strobes, a DIP switch for entering calibration data, a voltage regulator, a Hall effect switch, and a few resistors. I now had a single-chip speedometer.

I then discovered the program costs extra. Not a problem. I sent away for a cross-assembler and hacked out my own program in a couple of nights. However, I didn't know how to plug it into the PROM programmer. The Motorola databook says that the 68705 programs itself, but what if you don't happen to have the handy little circuit they show? Oh well, I wiped off the bifocals, cranked up the stereo, and got out the wire-wrap tool again.

With the programmer built, a programmed 27 16 (the 68705 programs itself by copying the contents of a 2716 EPROM), and a blank 68705 resting in their sockets, I fired up the self-programming circuit. The lights blinked for a while, and the *verified* light turned on eventually. I was having fun now! I plugged the 68705 into the speedometer board, hit the power, and-wait a minute, I thought LEDs were red instead of black.

I realized I probably missed a bit somewhere, so I grabbed the source code and started digging through it, promising myself that I would put comments in this time. Ah-ha! I discovered the error: port *B* should be output not input.

Round two. I reassembled the code, burned another 2716, burned a 68705 (you'll have to wait for the first one to erase if you don't have two), and tried it again. It worked, but I thought the scan rate was a tad low; there was a lot of dark time between successive digits, about 2 to 3 seconds. The solution to this problem was simple. I just shortened that delay loop by a factor of 1000.



MC68705P3		MC1 46805E2	
PORT A	$000	PORT A	$000
PORT B	$001	PORT B	$001
1 1 11 | PORT C	$002	EXTERNAL MEMORY	$002
NOT USED	$003	EXTERNAL MEMORY	'6003
PORT A DDR	$004	PORT A DDR	$004
PORT B DDR	$005	PORT B DDR	$005
1 1 1 1 | PORT C DDR	$006	EXTERNAL MEMORY	$006
NOT USED	$007	EXTERNAL MEMORY	6007
TIMER DATA REG	$008	TIMER DATA REG	$008
TIMER CONTROL REG	$009	TIMER CONTROL REG	$009
NOT USED	$OOA	EXTERNAL MEMORY	$OOA
PROGRAM CONTROL REG	$00B		
NOT USED	$OOC - OOF		$OOF
RAM (112 BYTES) STACK (31 BYTES MAXIMUM) ↑	5010 $07F	RAM (112 BYTES) STACK (64 BYTES MAXIMUM) ↑	$010 $07F
PAGE ZERO USER EPROM (128 BYTES)	5080 50FF	PAGE 0 EXTERNAL MEMORY	5080 $OFF
MAIN USER EPROM (1668 BYTES)	5100 $783	EXTERNAL MEMORY	5100
MASK OPTION REG	$784		
BOOTSTRAP ROM (115 BYTES)	5785 $7F7		
INTERRUPT VECTORS EPROM (8 BYTES)	$7F8 $7FF		$1 FF5
		INTERRUPT VECTORS	$1 FF6 – 1 FFF

Figure 1—*The* Motorola *MC68705P3* has the advantage of being *small and se/f* contained; *however, if doesn't support any external memory (above left). To ease the development cycle, the nearly* equivalent *MC146805E2 may be used for its support of external memory (above right) until the final code is complete.*

Round three. I reassembled the code, waited for the 2716 to erase (only bought two), burned the 2716, waited for the 68705 to erase (that's right, only bought two), programmed the 68705, and tried it again. All three digits on the display lit up showing zeros. Perfect. I stroked a magnet past the Hall effect sensor a few times and-viola! It told the speed just as the old speedometer did: 000 MPH. I figured the problem must be the interrupt. Maybe the timer.

Working in this way for several more rounds told me one thing: there has to be a better way. While my designs rarely work the first time and I expect a fair amount of problems, I was growing tired of the assemble, burn 2716, burn 68705, test, and scratch the head cycle (i.e., ABBTSTH cycle). I have read about, but never used, an in-circuit emulator (ICE), and I figured it was what I was missing. I placed a call to Radio Shack and asked them if they stocked an ICE for a MC68705P3. I was told the bottle shop in the other mall carried it, as did the convenience store on the corner.

If I could just get the 68705 to read the 2716 directly, I could eliminate one burn step in the ABBTSTH cycle. However, with such a low pin count, the 68705's inability to read external memory was no surprise. Nice try.

Then I came across another Motorola chip, the MC146805. It has almost the same memory map and registers as the 68705, plus it can read external memory. I guess someone at Motorola had the same problem before me; imagine that. Figure 1 shows a comparison of the memory maps for the MC68705P3 and the MC146805E2. A plan started coming together: 2K EPROM for a monitor program, 2K RAM for the application program, a serial port, some glue, and I/O lines brought out on a ribbon cable to a DIP plug. Instant ICE. Instead of using ABBTSTH, my development cycle would now be: plug the umbilical DIP cable into the hardware under test, assemble the program, download to application RAM, and execute the program. I eliminated all burn and erase stages. There you have it: the 68705 SLUSH (almost ICE).

With the basic ideas for the SLUSH in mind, I compiled a list of features I wanted:

1. Executes assembled programs from RAM
2. Views or modifies memory
3. Executes a program a single step at a time
4. Uploads assembled programs from a PC
5. Starts program execution from any address
6. Uses an AC/DC supply (I like wall-mounted transformers)

HARDWARE

Figure 2 shows a schematic of the system I arrived at after a few different trials. The circuit has nothing fancy like PALs; it's quite simple and straightforward. As I mentioned previously, the system consists of a power supply, a serial port, the CPU, and some memory.

U4 decodes the 8K address space of U5 into four 2K chunks. The first 2K block ($0000–07FF) is RAM and is used for the application program. The highest 2K ($1800–1FFF) is EPROM and contains the monitor program along with reset and interrupt vectors. The address range $1000–17FF is mapped to RAM and is used for system variables. A serial port is located at $0800–0FFF. The serial port occupies a full 2K although it only requires four locations. No other devices required memory address space, so any further decoding for the serial port was unnecessary.

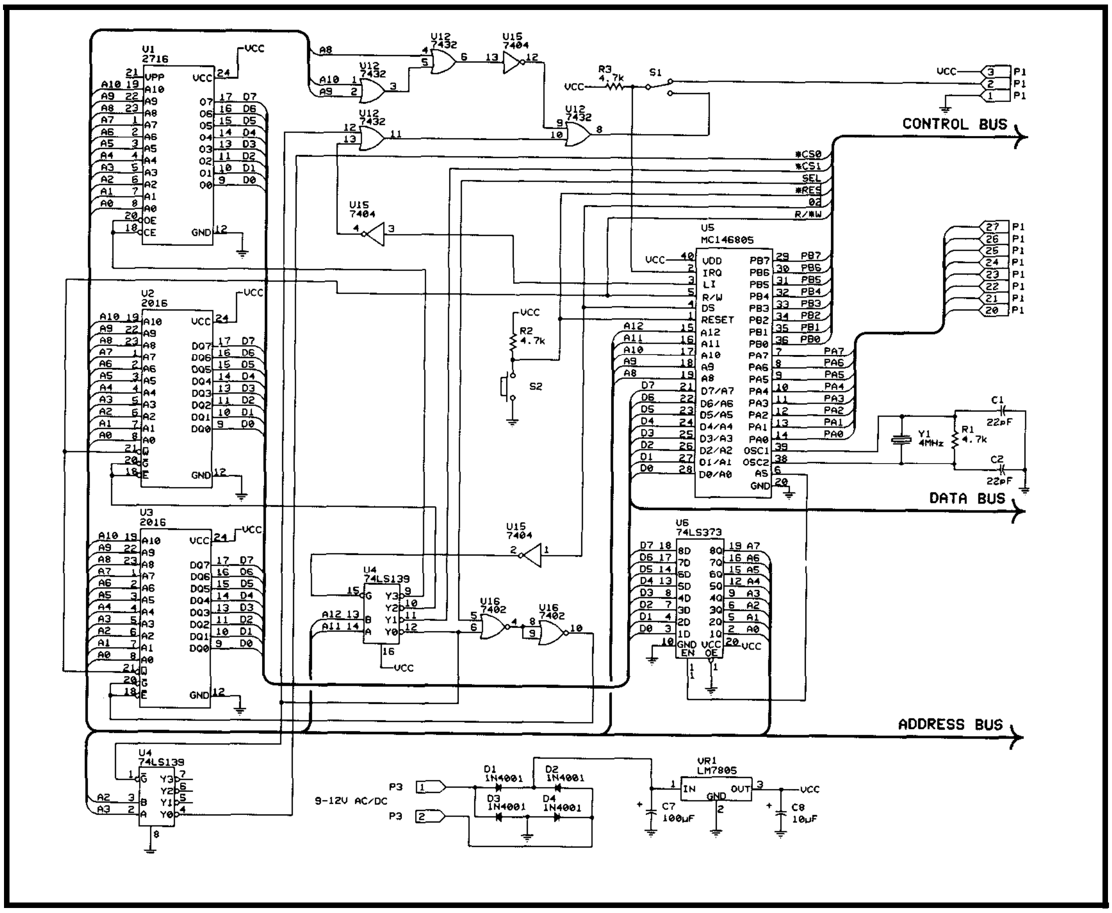


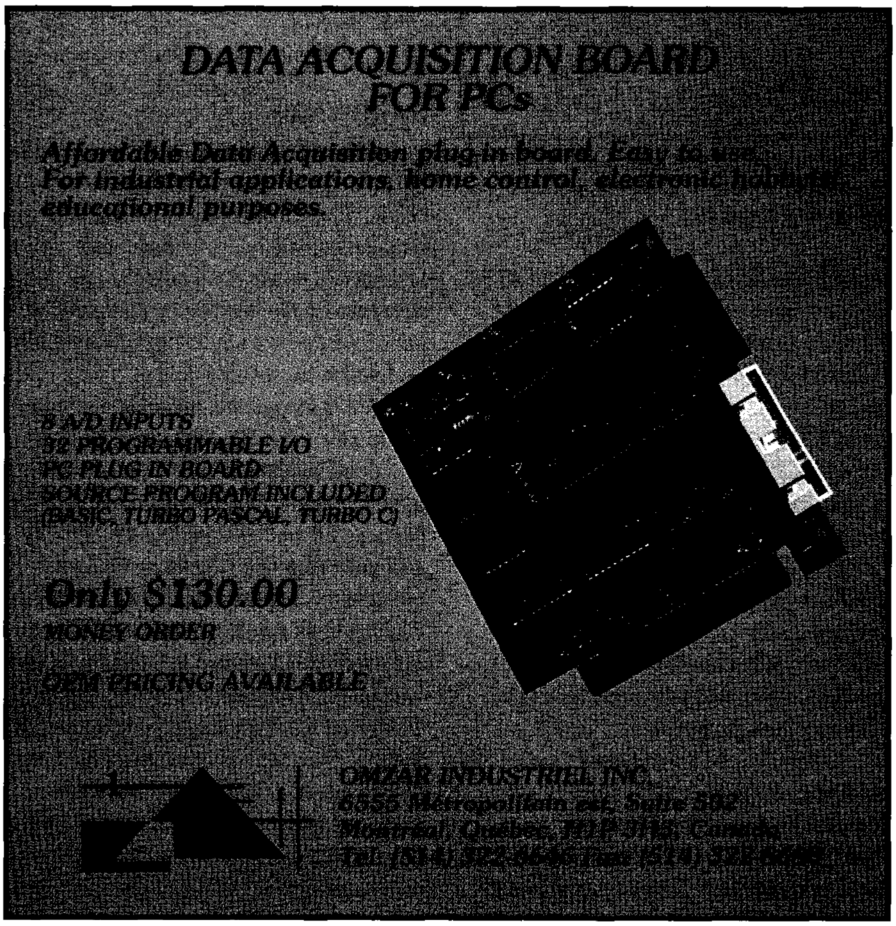
Figure **2a**—*The* 68705 *SLUSH is* based on the Motorola *MC146805E2. The* processor's *8K* external memory space is *broken up into 2K chunks. One chunk is used for a monitor program in EPROM, two chunks are used for RAM, and the last chunk is used for a serial port.*

A slight difference in the memory maps between the 68705 and the 146805 meant some additional hardware was required. The 68705 has a 4-bit I/O port [port C) addressed at location $0002. The 146805 has nothing at this address. U8 and U9 make an 8-bit I/O port. Making an 8-bit port was just as easy as making a 4-bit port, plus I now had an extra four I/O bits to play with.

U13 and U14 decode the address space down to a single location ($0002) for this port. When a read at location $0002 is done, the values present at U9 are placed onto the data bus. When a write occurs at location $0002, the values on the data bus are loaded into the latches in U8. In the 68705, a data direction register is also associated with port C. The SLUSH doesn't have this register, so the direction of the port C lines are realized with jumpers. Each bit must be jumpered to be either an input or an output.

U10 and U11 make up the serial port. I chose a 6551 to perform the serial interface duties simply because it was what I had laying around. U11, a MAX232, is a 5-volt-only RS-232 driver-receiver. This chip sure has done a lot to simplify serial port design, eliminating the need for any extra power supplies or DC-DC converters.

I added some buffering capability to port *B* in order to drive a display



because my intention was to use the SLUSH in the development of the LED-based speedometer. U7 provides for driving loads of up to 20 mA, which is plenty for LEDs, when port *B* is used for output. Port *B* uses jumpers to select between buffered output or regular input/output for each bit.

To create the feature for single stepping through an application program, I decided to generate an interrupt whenever memory in the application RAM was accessed. An interrupt handler would then display register contents and return control to the application program. Tying the chip select of U3 to the IRQ line on U5 would do just that. Then I realized that multiple interrupts would occur for all multiple byte instructions. This aspect was not what I was after.

After a few failed ideas, I noticed the LI pin on the 146805. The databook says this pin indicates that the next opcode is being fetched, but that it is used only for certain debugging and test systems. The pin turned out to be just what I needed to generate the signal required. When single stepping is enabled, an interrupt occurs for every instruction executing from the application RAM. U12 generates this interrupt.

Sl is a switch that chooses between external interrupts or the single-stepping interrupt. If you are debugging interrupts on the circuit under development, Sl routes that interrupt signal to U.5. In this case, single stepping through the application program is not possible.

The power supply uses a standard 5-volt regulator with a bridge rectifier on the input. This device allows the use of a wide range of input values, either AC or DC.

SOFTWARE

A 27 16 contains the software, which is very straightforward. Routines for serial I/O (the serial port is not yet interrupt driven), memory viewing or modifying, uploading memory from the PC, and program executing are available.

I had to overcome one problem in the software. The 146805 has no form of indirect addressing. This omission made performing the memory examining or modifying routines a little difficult, as well as the "start execution at" routine. To get around this problem, I put a small stub of code in system RAM that would perform LDA, STA, or JMP instructions. I stuffed the memory locations that followed the actual instruction with the desired addresses and then executed the stub routine. Perhaps you could consider it *indirect-indirect addressing.* (Some people would call this technique self-modifying code, but sometimes you do what you have to do.)

The 146805 has interrupt vectors in EPROM at $1FF6–1FFF while the application program has its vectors in RAM at $07F8–07FF. To allow for generic interrupt handling, I set up the vectors at $1FF6–1FFF to jump to locations $07F6–07FF. In this manner, the vectors at $07F6–07FF can be easily changed under program control. When a reset occurs, the vectors at $07F6–07FF are loaded with default values.

When single stepping through a program, the contents of the program counter, accumulator, X register, and the condition codes are displayed on the screen. This feature involves a little detective work. When a single-step interrupt occurs, the current conditions are pushed onto the stack and control transfers to the interrupt handler. These values could be displayed from the top of the stack, but there is no way to know where the top of the stack is exactly. To solve this problem, the very first thing that executes in the interrupt handler is a JSR. Doing a JSR pushes the current address onto the top of the stack, so control will return after an RTS. This address will be the address of the second instruction in the interrupt handler routine.

The address of the interrupt handler is known at assembly time, so it is assembled into the code. To find the top of the stack, you have only to look through the stack area until the known address is found. Then, the following locations will contain the registers in question. Stack instructions like PUSH and PULL would make this process much easier.

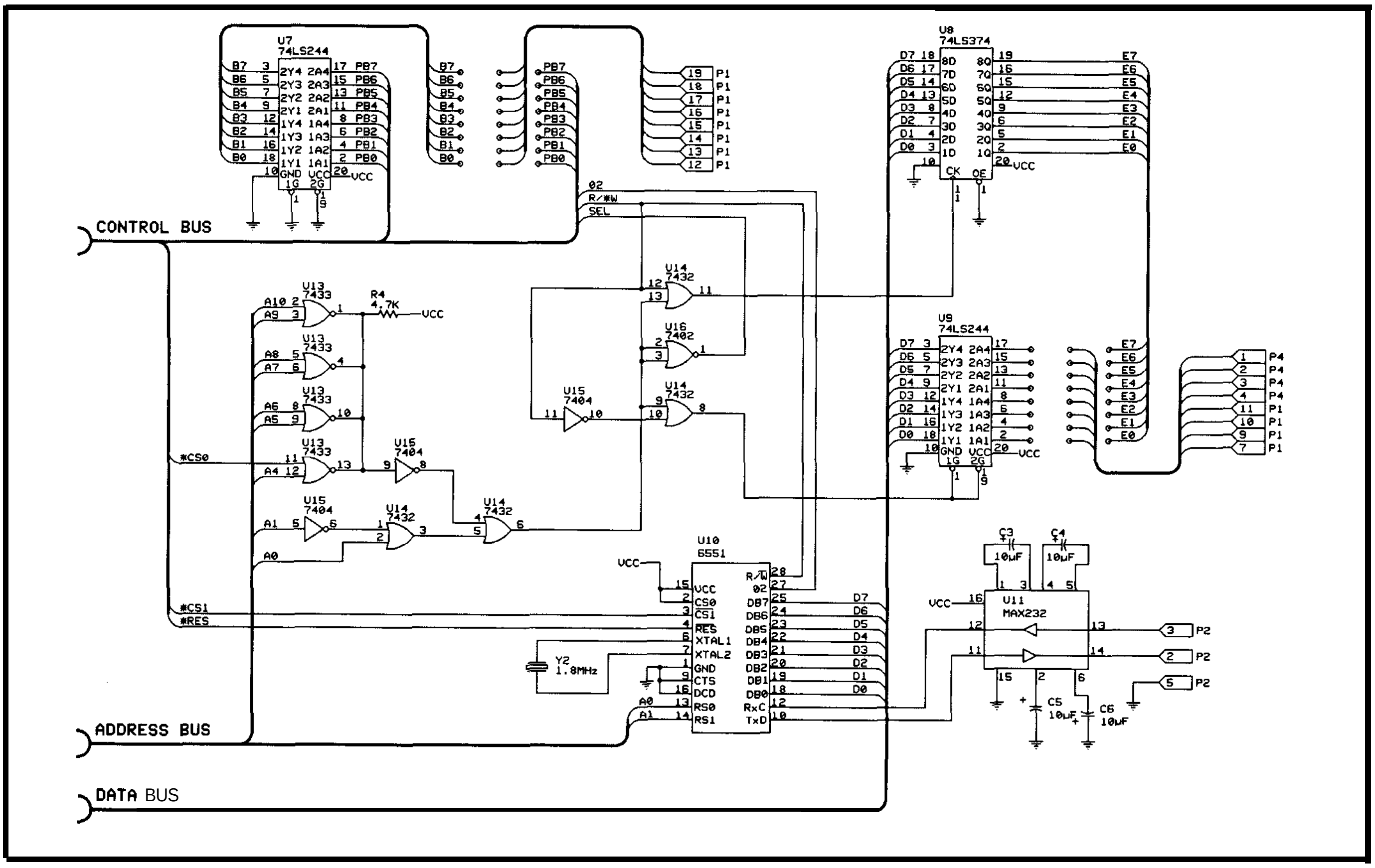
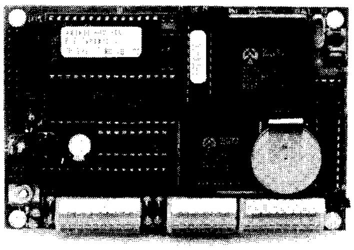


Figure 2b—*A MAX232 makes interfacing the 6551 serial port to* standard RS-232 *easy. The SLUSH also has drivers on board specific to the LED-based* speedometer first developed with *the* board.



COMMANDS

SLUSH has seven basic commands. They are listed below along with a description of each.

M-examine or modify memory. An *M* followed by a four-digit address will display the contents of that location and allow you to enter a new two-digit value to place into that location, if desired. Pressing space will advance to the next location and open it for changes. Pressing Enter quits.

D-memory dump. A *D* followed by an address will dump 128 bytes of memory to the screen starting at the input address. The program then waits for a key to be pushed. Pressing *D* again will cause the next 128 bytes to be dumped. Pressing Enter will terminate the routine and return control to the monitor.

G-start execution at a given address. A G followed by a four-digit address will cause the program to go to that location and start running.

L-upload memory block. *L* causes the SLUSH to wait for an *S* format record from the serial port. *S* format records are generated by most Motorola-compatible cross-assemblers. This command is how programs are uploaded for testing.

T-trace a program in single-step mode. A *T* followed by an address causes execution to start at the desired address in single-step mode. After each instruction, the contents of the program counter and registers are shown on the screen. Pressing any key executes the next instruction.

E-enable interrupts. *E* performs a CLI, which clears the interrupt mask flag. This step must be performed before a program can be traced in single-step mode.

O-turn interrupts off. 0 disables further interrupts by setting the interrupt mask flag.

WRAP UP

The SLUSH I developed has saved me a lot of time during software development and testing. I plan to incorporate a programmer onto the SLUSH board in the near future. I will also add a routine to the monitor to program the 68705 directly from the on-board RAM.

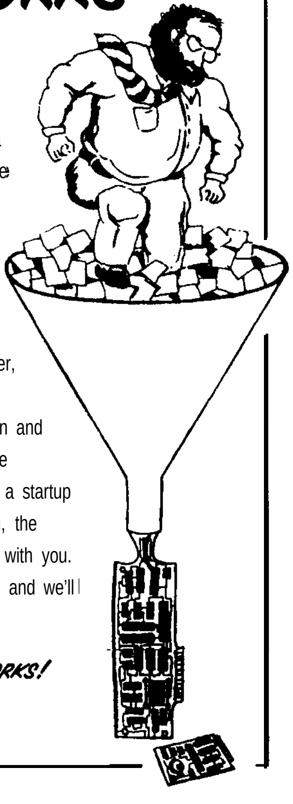
Now when I'm out cruising with my Dad in the Green Slime Machine and I ask, "Hey Dad, how fast are you going?" he smiles and says, "Oh, I would say exactly 55,000,000 micromiles per hour." ❑

Robin Brophy holds a BSEE and an MSEE from North Dakota State University. He currently works for a major sugar company doing instrumentation and process control.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

404 Very Useful
405 Moderately Useful
406 Not Useful



Program 8748149s with the DAR-49

The 8748/9 has been around forever, but people still have trouble when it comes time to program its internal EPROM. With the right combination of dumb hardware and smart software, the DAR-49 is one inexpensive solution.

FEATURE ARTICLE

Paul Hitchcock

With new and improved microcontrollers popping out of the ether every other week, confusing "old" with "obsolete" is easy to do. Case in point: the Intel 8748/49H programmable microcontrollers. Though ancient by high-tech standards (they're well into their second decade of production] and seldom chosen for new designs, these chips still pack enough power to be useful in an extraordinary number of embedded applications.

Of course, if you're designing a system to control a nuclear power plant or an interplanetary space probe, going with something a bit more capable might be a good idea. However, if you only want to monitor a few sensors, turn on a motor or two, and handle the occasional interrupt, the 8748/49H (priced under $10) is the simplest and most cost-effective solution. After all, the hammer has also been around for a long time, but it's still pretty handy to have near your workbench.

The 8748/8749H is a 40-pin, single-chip microcontroller having 64 or 128 bytes of on-chip RAM, 1K or 2K EPROM, 27 general-purpose I/O pins, a built-in timer, and a single interrupt input. Although I/O and RAM are expandable, perhaps the ruling virtue of these controllers is simplicity-you can construct a working system using only four components: the controller itself, a crystal, and two capacitors. On the software side of the street, the 8748/49H has an instruction set composed of 56 instructions, most of which execute in a single machine cycle, cramming a lot of program into a small amount of memory.

Although the 8748/49H is fairly inexpensive, the cost of the requisite programmer made me want to run into the street screaming. To minimize the damage to my wallet and avoid confirming the neighbors' opinions of me, I decided instead to design the DAR-49 8748/49H programmer shown in Photo 1. The DAR-49 connects to a standard IBM-compatible printer port, uses only readily accessible ICs, and programs an entire 8749H in just under three minutes. However, such simplicity has its price. The programmer is a little short in the gray-matter department (OK, it's stupid) and must be controlled by software running on the host PC. The acronym "DAR"? Well, it stands for "Dumb As a Rock."

PROGRAMMING THE **8748/49H**

To program an 8748/49H, a set of appropriate and properly sequenced signals must be sent to as many as 16 different controller pins. Rather than go into a long-winded discussion of the programming algorithm, I have summarized it in Table 1. If you require further details (such as signal timing information), please refer to the 1989 edition of Intel's *8-Bit Embedded Controller Handbook,* pages *4-29* through 4-32.

THE PC PRINTER PORT

Before explaining how the DAR-49 works, examining the relevant characteristics of the standard IBM PC printer port is necessary. The printer port interface uses three consecutive 80x86 I/O port addresses referred to as *BASE, BASE*+1, and *BASE*+2. Depending on the jumper or DIP-switch settings on a given printer card, *BASE* may have the value $278, $378, or $3BC, with $378 being the standard PC/AT assignment for LPT1.

Port *BASE* shadows the 8-bit output-only port normally used to send character data to an attached printer. When a byte is written to port *BASE,* each bit in the byte is mapped into a TTL-level signal that correspondingly appears on one of pins 2 through 9 (least- to most-significant bit) of the printer port's DB-25 output connector. These bits provide eight of the eleven control signals needed by the DAR-49 programmer.

The remaining three control bits are obtained using port *BASE*+2. Bits 1, 2, and 3 of this port control the signal levels on pins 14, 16, and 17 of the printer output connector, respectively. These pins are configured as open-collector I/O devices, but the DAR-49 uses only the output function. If resistors are connected between each of these pins and a 5-volt supply, and a byte having bits 1 and 3 set is sent to *BASE*+2, then a TTL " 1" will appear at pins 14 and 17. However, the function of bit 2 is inverted, so a "0" sent to bit 2 of port *BASE+2* will result in a TTL "1" at pin 16 and vice versa.

Lastly, the programmer has to send information back to the host computer. Fortunately, pin 13 on the printer output connector is a single-bit input port that may be read by checking bit 4 of port *BASE*+1; if bit 4 is set, then a TTL " 1" is present at pin 13.

Table 2 describes the relationship between the printer port pins and the programmer, and Table 3 gives a bit-wise description of the programming algorithm adapted to the DAR-49 hardware.

HARDWARE DESCRIPTION

Referring to Figure 1 and Table 2, notice that two serial-to-parallel converters (IC2 and IC3) and one parallel-to-serial converter (IC5) handle all communication between the host computer and the programmer in a serial fashion. Bit 2 of port *BASE* transfers data-address information to the programmer a bit at a time, with bit 0 providing a pulse to clock in the data and lower eight bits of the address to IC3; similarly, bit 1 of port *BASE* clocks in the upper three address bits to IC2. Data coming from the programmer to the host PC is transferred over bit 4 of *BASE*+1 via IC5, with bit 5 of port *BASE* providing the clock pulses for the transfer.

Of the remaining four bits available from port *BASE,* bits 6 and 7 control the RESET and TEST0 inputs on the 8748/49H, respectively. Bit 4 controls the output-enable pin of IC4, an LS373 octal buffer, making the isolation of the 8748/49H data-address bus from the output of IC2 possible. Finally, bit 3 of *BASE* controls the parallel-load pin (. PL) on IC5.

The reason why IC5 is a CMOS component while all the other ICs are LS TTL is because the 8748/49H bus must at times "float" (i.e., the bus must be disconnected from the circuit) during the programming cycle (see Table 1). When the output of IC4 is disabled (* OE high), the high-impedance inputs of IC5 ensure the bus will indeed achieve a floating state. Do not substitute an LS device for IC5, or the programmer will not work!

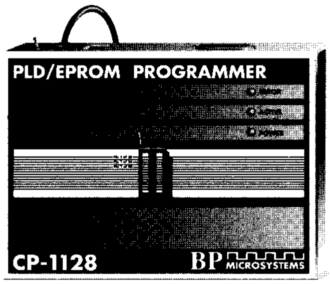
On-board regulators REG1 and REG2 supply the required 18- and 21-volt programming voltages (after you connect a suitable external supply, of

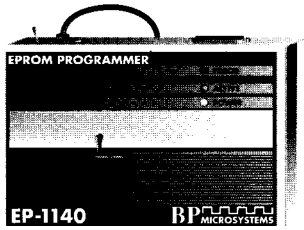
		Vdd	RESET	TEST0	EA	BUS	PROG	**P20–P22**
Step	Description	(28)	(4)	(1)	(7)	(12-9)	(25)	(21-23)
1	Initialization	5V	0V	5V	5V	float	float	don't care
2	Insert 8748-49							
3	Select program mode			0V				
4	Activate program mode				18V			
5	Apply address					addr. low		addr. high
6	Latch address		5V					
7	Apply data to BUS					data in		
8	Apply program power	21V						
9	Apply program pulse						18V (50ms)	
10	Remove program power	5V					float	
11	Enter verify mode			5V				
12	Read and verify data					data out		
13	Exit verify mode			0V				
14	Remove address latch		0V					
15	Continue or Exit	See notes c and d						

Notes:

a. Column header numbers in parentheses are device pin numbers
b. A blank entry denotes no change from last nonblank entry in the column
c. To program next address, repeat from step 5.
d. When programming is completed, perform step 1 (Initialize), pause to remove 8748/49

Table I--The *8748/9 programming* algorithm is readily available, but not an easy one to implement. The *old* microprocessor requires fifteen separate steps (eleven *after* the first address) and three different voltages to accomplish the task.





Printer Port Pin #	Type	Address	Bit No.	DAR-49 Function
2	output	BASE	0	BUS SP clack (IC3)
3	output	BASE	1	HIGH Address SP clock (IC2)
4	output	BASE	2	SP data input (IC2 & IC3)
5	output	BASE	3	PS parallel load function (IC5)
6	output	BASE	4	● OE control, 74LS373 (IC4)
7	output	BASE	5	PS clocdut (IC5)
8	output	BASE	6	8748/49 TESTO, pin 1 (ZIF)
9	output	BASE	7	8748/49 *RESET, pin 4 (ZIF)
13	input	BASE+1	4	PS data output (IC5)
14	O.C. I/O	BASE+2	1	Vdd control (Q5)
16	O.C. I/O	BASE+2	2	EA control (Q1)
17	O.C. I/O	BASE+2	3	PROG control (Q3)
18-25	ground	——	——	ground

Notes:

O.C. I/O – "open collector, input or output"
BASE – lowest address of PC I/O port assignment for printer
PS – Parallel to Serial
SP – Serial to Parallel

Table *2-Each bit on the PC's parallel printer port is assigned a specific function on the DAR-49. Communications are handled serially reducing the number of bits required.*

Step	I/O Port BASE		I/O Pot-I BASE +2		Comment
	Binary	Hex	Binary	Hex	
1	10010000	90	00000100	04	Initialize the DAR-49
2					Chip insertion pause
3	00010000	10	00000100	04	Select program mode
4			00000000	00	Activate program mode
5	oooooxxx	ox			Load address bus
6	01000000	40			Latch address
7	01 oooxxo	4x			Load data bus
8	01000000	40	00000010	02	Apply program power
9			0000x010	ox	Send program pulse
10			00000000	00	Remove program power
11a	01010000	50			Disable LS373 (float bus)
11b	11010000	DO			Activate verify mode
11c	11011000	D8			"Freeze" LSI 65 data
12	11x11000	X8			Read 8748/49H data bus
13	01010000	50			Deactivate verify mode
14a	00010000	10			"Unlatch" address
14b	00000000	00			Enable LS373
15					Repeat from 5 until done
16	10010000	90	00000100	04	Restore initial state
17					Chip removal pause

Notes:
a. A blank entry denotes no change from last nonblank entry
b. "x" and "X" denote quantities that change during the step (see Listing 1)
c. Data in step 12 appears in bit 4 of I/O port BASE+1

Table 3—*The driver program on the PC follows a strict recipe while burning a program info an 8748/9. Each step is referenced within the program comments.*

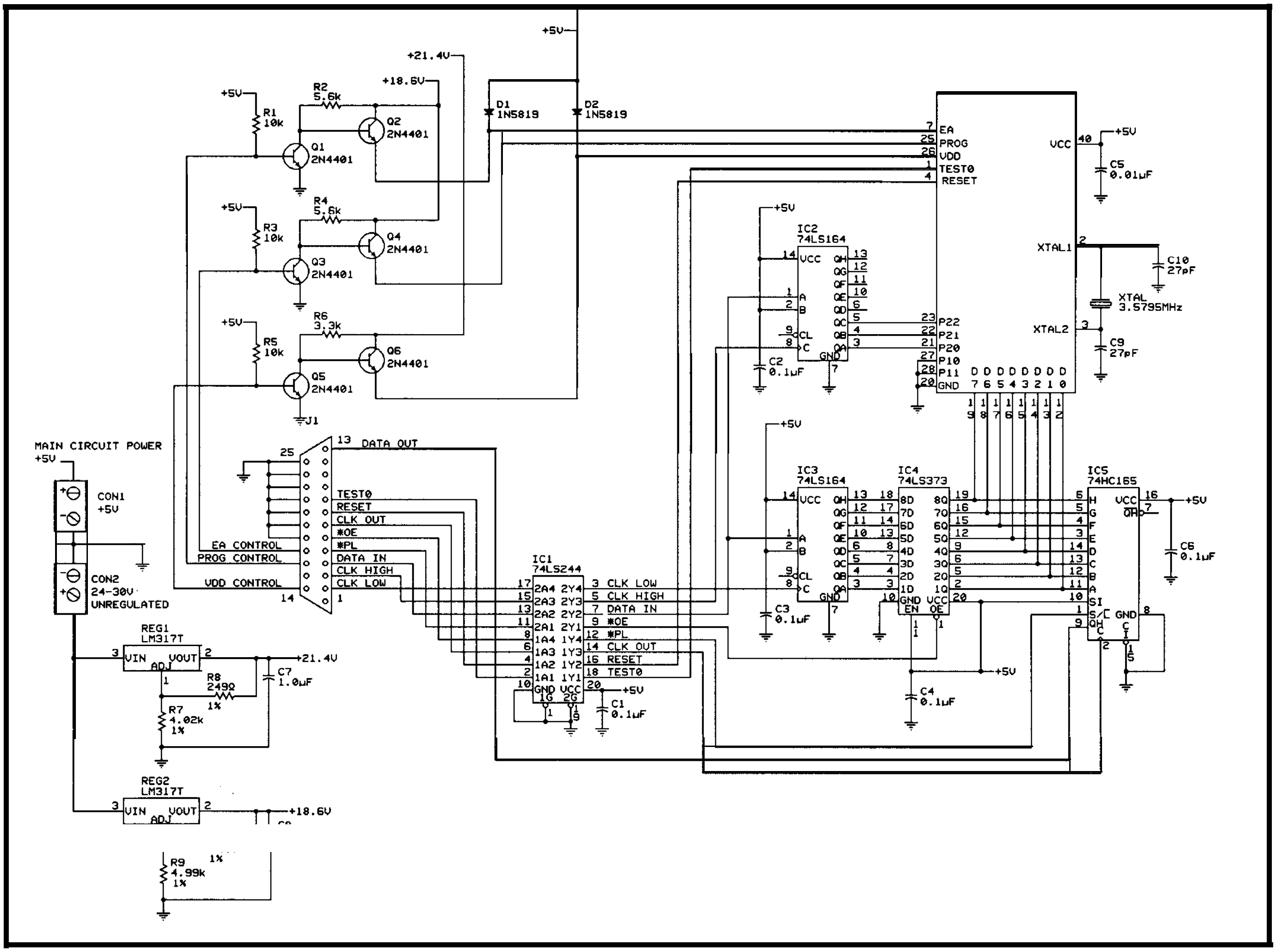


Figure 1—*All necessary programming voltages are generated on the DAR-49 and may be switched on and off by the PC. Because of limitations in the PC's parallel port, most communications are handled serially with a pair of serial-to-parallel converters and a single parallel-to-serial converter. Note that IC5 must be a CMOS part for the programmer to work properly.*

course), which are controlled by writing to bits *1* through 3 of port *BASE+2* in the manner I previously described. Note also that the corresponding printer port pins are connected directly to the bases of transistors Q1, Q3, and Q5. You may ground these pins when testing the circuit after construction, but do not connect any of them to the TTL supply without using a current-limiting resistor! Emitter-base junctions make excellent fuses.

CONSTRUCTION

You can use nearly any construction technique you like with the DAR-49 because it is not a high-speed device. For example, I built the prototype using a combination of wire-wrap and point-to-point wiring with a cheap Radio Shack grid board to mount it. Adhere to the usual restrictions, including keeping wiring runs short and using a printer port cable shorter than 3 feet long, unless you want a really slow programmer, an unlicensed radio station, or both.

The programmer requires two external power supplies: a standard TTL-type supply to power the logic and a 25- to 30-volt source for the programming voltages. The latter doesn't have to source much current (about 30 mA), so you can probably get away with using a series combination of Y-volt alkaline batteries if you don't have a suitable bench supply lying around. Here, I must confess to a certain degree of pragmatism (pronounced "laziness") in the design—why complicate the circuit with an expensive on-board supply when I will use the programmer only infrequently?

Finally, a word about the *1%* resistors used in the regulator portion of the circuit. If you can't obtain the values shown in the parts list, substitute 240-ohm resistors for R8 and R10, and 10k trimpots for R7 and R9. Adjust the trimpots to obtain the voltages shown in the schematic and you'll be ready to go. (Incidentally, I built the prototype using trimpots and revised it only after finding a dusty box of 1% resistors in the bottom of a drawer marked "capacitors." You could call this serendipitous approach *top-down engineering,* meaning start at the top of a pile of stuff and look through it until you find a part that works or can be made to work.)

USING THE DAR-49 PROGRAMMER

Without software, the DAR-49 is just a rather flimsy doorstop, so I've included the simple driver program given in Listing 1. Although written in

Listing 1—*All the smarts for the admittedly* dumb DAR-49programmercome from *the* PC to which if's connected.

```
PROGRAM PROG8748;
USES Dos,Crt;

CONST
  HexStr  : string[16] = '0123456789ABCDEF';
  BASE    = $378:         { Base address of LPT1 printer port }
  BASE1   = BASE+1;
  BASE'2  = BASE+2;

  LOOPCNT = 100: { Used to provide a delay between port accesses }
                 { on a 16-MHz 386SX. This number must be        }
                 { proportionally increased for systems with     }
                 { faster clock speeds.                          }

TYPE
  action  = (PGM,RD);

VAR
  Intel_buf:  RECORD
                bytes:array[0..255] of byte; { decoded hex line }
                HADDR:word;                  { starting addr of line }
              END:

  Hexfile    : text;                     { Intel hex file           }
  HexLine,                               { Hex line read from file }
  FHexName   : string:                   { Intel hex file name      }
  i          : word:

  ReturnVal:  byte;
  ch         : char;


 PROCEDURE wait(x:LongInt);
 VAR
   count : LongInt;
 BEGIN
   FOR count := 1 TO x DO: { simple delay loop }
 END;


 FUNCTION DoByteOp( DATUM    : byte:        { byte to be programmed }
                    ADDR     : word;        {  address of interest }
                    REQUEST : action        { ProGraM or ReaD       }
                    ): byte:
 {
   This function accepts an address and a byte as input and,
   depending on the setting of REQUEST (PGM=program, RD=read),
   either programs a byte in an 8748/49H microcontroller, or
   retrieves the current byte at address ADDR. If program is
   selected, the value returned is the programmed value which
   may not equal DATUM in case of an error. }

 VAR
   InByte: byte:   { holds byte read from 8748/49H}
   COUNT : word;   { gen. purpose counter         }

 BEGIN  { DoByteOp}

   FOR COUNT := 1 TO 3 DO BEGIN          { 5a. Load upper addr bits }
     IF (ADDR and $400)<>0 THEN BEGIN { bit 1 of BASE is clock    }
       Port[BASE] := 4; wait(LOOPCNT);{ Clock out a "1"}
       Port[BASE] := 6; wait(LOOPCNT);
       END
     ELSE BEGIN
       Port[BASE] := 0; wait(LOOPCNT);   { Clock out a "0"}
       Port[BASE] := 2: wait(LOOPCNT);
```

(continued)

Listing 1-continued

```
    END: {IF}
    ADDR := ADDR shl 1;
  END:

  FOR COUNT := 1 TO 8 DO BEGIN          { 5b. Load lower addr bits }
    IF (ADDR and $400)<>0 THEN BEGIN { bit 0 of BASE is clock    }
      Port[BASE] := 4; wait(LOOPCNT);{ Clock out a "1"}
      Port[BASE] := 5; wait(LOOPCNT);
      END
    ELSE BEGIN              { Clock out a "0"}
      Port[BASE] := 0; wait(LOOPCNT);
      Port[BASE] := 1; wait(LOOPCNT);
    END: {IF}
    ADDR := ADDR shl 1;
  END;

  Port[BASE] := $40;wait(LOOPCNT);    { 6. Latch address }
  {
      Steps 7 through 10 are skipped if we're only going to
      read the byte at ADDR.
  }
  IF REQUEST = PGM THEN BEGIN
    FOR COUNT := 1 TO 8 DO BEGIN          { 7. Load data bus }
      IF (DATUM and $80)<>0 THEN BEGIN { bit 0 of BASE is clock }
        Port[BASE] := $44; wait(LOOPCNT);{ Clock out a "1" }
        Port[BASE] := $45; wait(LOOPCNT);
        END
      ELSE BEGIN
        Port[BASE] := $40; wait(LOOPCNT);{ Clock out a "0"}
        Port[BASE] := $41; wait(LOOPCNT);
      END: {IF}
      DATUM := DATUM shl 1;
    END:
    Port[BASE]  := $40;wait(LOOPCNT);

    Port[BASE2]:= $02;                 { 8.  Apply program power }
    Port[BASE2]:= $0A;                 { 9.  Apply program pulse-}
    Delay(55);                         {     width approx. 55 ms }
    Port[BASE2]:= $02;                 {     end of pulse        }
    Port[BASE2] := 0;                  { 10. Remove program power }

  END: { IF REQUEST = PGM }

  Port[BASE] := $50;wait(LOOPCNT);{ lla. "float" BUS              }
  Port[BASE] := $D0;wait(LOOPCNT);{ 11b. Set verify mode          }
  Port[BASE] := $D8;wait(LOOPCNT);{11c. "freeze" data in HC165}

  InByte := 0;                       { 12. Read BUS               }
  FOR COUNT := 1 TO 8 DO BEGIN
    InByte := InByte shl 1;          { Adjust "bit accumulator" }
    IF (Port[BASE1] and 16)=16 THEN { Incoming bit is "1"       }
      InByte:=InByte+1;
    Port[BASE]:=$F8; wait(LOOPCNT);{ Clock in next bit }
    Port[BASE]:=$D8; wait(LOOPCNT);{ bit 5 of BASE is the clock }
  END;

  Port[BASE] := $50: wait(LOOPCNT);{ 13. Exit verify mode         }
  Port[BASE] := $10; wait(LOOPCNT);{ 14a. "unlatch" address      }
  Port[BASE] := 0:   wait(LOOPCNT);{ 14b. Enable LS373            }
  DoByteOp  := InByte;             { return with byte read        }
END; { DoByteOp }


PROCEDURE DecodeLine(VAR Hstr : string);
VAR
  index,
  chptr : word;      { character pointer }
```

(continued)

Listing 1-continued

```
  FUNCTION HexToDec : byte;
  BEGIN
    HexToDec := 16 * (Pos(Hstr[chptr], HexStr)-1)
                 + Pos(Hstr[chptr+1], HexStr)  1;
    chptr  := chptr + 2;
  END:

BEGIN
  chptr := 2;
  WITH Intel_buf DO BEGIN
    bytes[0]:= HexToDec;
    IF bytes[0] = 0 THEN exit;
    HADDR := HexToDec:
    HADDR := 256*Intel_buf.HADDR+HexToDec;
    chptr := 10;
    FOR index := 1 TO bytes[0] DO bytes[index]:= HexToDec;
  END; { WITH }
END;  { DecodeLine}


BEGIN
  ClrScr;
  WRITE('Hex file name? '); READLN(FHexName);
  ASSIGN(HexFile,FHexName);
  {$I-}
  RESET(HexFile);
  {$I+}
  IF (IOResult <> 0) OR (FHexName='') THEN BEGIN
    WRITELN('File not found:  program stopped.');
    HALT;
  END:

          { *** INITIALIZE 8748/49H *** }

  Port[BASE]  := $90 ; wait(LOOPCNT);
  Port[BASE2] := $04 ; wait(LOOPCNT);

          { *** PAUSE TO INSERT CHIP *** }

  WHILE KeyPressed DO ch := ReadKey;         { clear kbd buffer }
  WRITELN;
  HighVideo;
  WRITE(chr(7),'Insert 8748/49H, then press a key');
  LowVideo;
  WRITELN;
  ch:= ReadKey;

  Port[BASE]   := $10: wait(LOOPCNT);{ 3. select program mode   }
  Port[BASE2]  :=   0: wait(LOOPCNT);{ 4. activate program mode }

  WHILE not eof(HexFile) DO BEGIN
    READLN(HexFile,HexLine);
    WRITELN(HexLine);
    DecodeLine(HexLine);

            {*** PROGRAM THE 8748/49H ***}

   { Intel_buf.HADDR      <-- the starting address of the line
     Intel_buf.bytes[0]   <-- the number of bytes to be programmed
     Intel_buf.bytes[1..]<-- the bytes to be programmed }

    WITH Intel_buf DO FOR i:= 1 TO bytes[0] DO BEGIN
      ReturnVal := DoByteOp(bytes[i],Haddr+i-1,PGM);
      IF ReturnVal <> bytes[i] THEN
        WRITELN(chr(7),'Bad value at ',Haddr+i-1,'= ',ReturnVal,
              '  [',bytes[i],'expected]');
    END: {WITH..FOR}
```

(continued)

Listing 1-continued

```
  END: {WHILE}
  CLOSE(HexFile);

        {*** REINITIALIZE 8748/49h ***}

  Port[BASE1 := $90   ; wait(LOOPCNT);
  Port[BASE2] := $04   ; wait(LOOPCNT);


        {*** PAUSE TO REMOVE CHIP ***}

  WHILE KeyPressed DO ch := ReadKey;           { clear kbd buffer }
  WRITELN:
  HighVideo;
  WRITE(chr(7),'Remove the 8748/49H, then press a key');
  LowVideo;
  WRITELN;
  ch:= ReadKey;
END.
```

Borland's Turbo Pascal 6.0, the program should be easy to expand or translate into BASIC, C, or whatever.

To use the DAR-49 programmer, connect it to the PC printer port, turn on the host computer, then apply power to the programmer. Next, run the driver program in Listing 1.

During the program's run, it prompts you for the name of an Intel hex file containing the data to be programmed. Assuming a valid file name is typed in, you are prompted to insert the 8748/49H into the ZIF socket, then to press any key to begin programming. The DAR-49 driver program reads in the file, programs the controller, and when complete, asks you to remove the controller from the socket. Do not insert or remove the 8748/49H unless prompted to do so or you will damage the chip.

Note that the driver has minimal error checking. If a byte is incorrectly programmed, a message is displayed giving the address of the byte in question, followed by both the incorrect value and the value of the byte as read in from the file.

The only other point worth mentioning about the driver program is the method used to generate the programming pulse width. The program uses the standard Turbo Pascal delay procedure with a 55-ms argument-right in the middle of Intel's 50- to 60-ms programming specification. I measured the pulse width using a 20-MHz oscilloscope and found it to be accurate to within 1 ms. However, if you are uncomfortable using this admittedly approximate (according to Borland's documentation) timing procedure, you might consider changing this part of the program.

The DAR-49 programmer gives you a quick and inexpensive entry into the wonders of 8748/49H programming-now go forth and control your world! ❑

Paul Hitchcock holds a bachelor's degree in physics from the University of California at Berkeley and has published many computer-related articles over the past 10 years. He is currently developing controller applications to aid in teaching high school and undergraduate physics.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

IRS

407 Very Useful
408 Moderately Useful
409 Not Useful

LCD Lineup

Getting Graphic with the LM213B

FEATURE ARTICLE

Tom Cantrell

Picture this: you have an application that needs more than just plain text display capability. Here's a graphic example of an LCD module that can handle bit-mapped drawing.

If you're like me, you've got a *Things To Do* list. For quite some time, one item on my list that I've been gathering the bits and pieces for is my own personal SDI (Suburban Defense Initiative) system: X-10 gadgets, sirens, beepers, bells, and even an electric (pooper-scooperless) dog. Of course, my security system needs a display panel to warn me of impending invasions, so gathering dust with all the other junk is an LM213B LCD display.

As other popular LCD modules discussed previously in these pages (see "An HCS II LCD Terminal" and "The True Secrets of Working with LCDs" by Ed Nisley, in issues #27 and #8, respectively, of the Computer *Applications Journal),* the LM213B has a built-in character generator and easily handles alphanumeric displays. However, a unique feature of the LM213B is a bit-map graphics mode.

Most people would be happy with a security system that provides a textual description of the nature of the threat, but I prefer a more graphic approach. I envision the display showing the layout of my house and yard with some kind of "bad guy" icon tracking the progress of the culprit while I haul out the heavy artillery.

Therefore, having made the big [and exceedingly unusual) move to shorten my *Things To Do* list, my first step was figuring out how to make the LM213B graphics mode work.

MAKING A FEW CONNECTIONS

The LM213B is an approximately 2" x 6" display with built-in smarts in the form of an HD61830 controller IC and a bunch of HD44100 row and column drivers. Together, the ICs insulate you from the gory details of driving the "glass" itself and present an easy-to-use 6800-family interface (see Figure 1).

The LCD receives and sends commands and data over the 8-bit bus *DBO–DB* 7. Unlike smaller alphanumeric-only modules (based on the HD44780 controller), the HD6 1830/ LM213B has no provision for a 4-bit data bus mode.

The other key signals to complete a transaction on the data bus are *RS, R/W*, CS*,* and *E. RS* stands for *Register Select* and serves to differentiate accesses between the HD61830 instruction register *(RS* = 1) and data register *(RS = 0). R/W* (Read/Write)* similarly serves to select between read *(R/W** = 1) and write *(R/W* = 0)* accesses. *CS** is the typical *Chip Select* function.

RS, *R/W** and *CS** are status-type signals which should be asserted at the beginning of an access cycle and kept stable throughout the access. *E,* which stands for *Enable,* is the strobe signal asserted to clock the data to and from the LCD. Figure 2 shows the typical timing of a data transfer.

Of course, you have to assert *RES** (Reset] before doing anything. This aspect is another difference from the smaller HD44780-based modules, which feature an internal power-on-reset circuit. An inital reset to the LM213B will save a lot of needless head scratching.

Pin #	Symbol	Pin #	Symbol
1	V_{SS}(GND)	11	DB4
2	V_{DD}	12	DB5
3	V_0	13	DB6
4	RS	14	DB7
5	R/$\overline{W}$	15	$\overline{CS}$
6	E	16	RES
7	DBO	17	V_{EE}
8	DB1	18	N.C
9	DB2	19	N.C
10	DB3	20	N.C

Figure 1— The *LM213B* LCD module *presents* an easy-to-use *6800-family interface.*

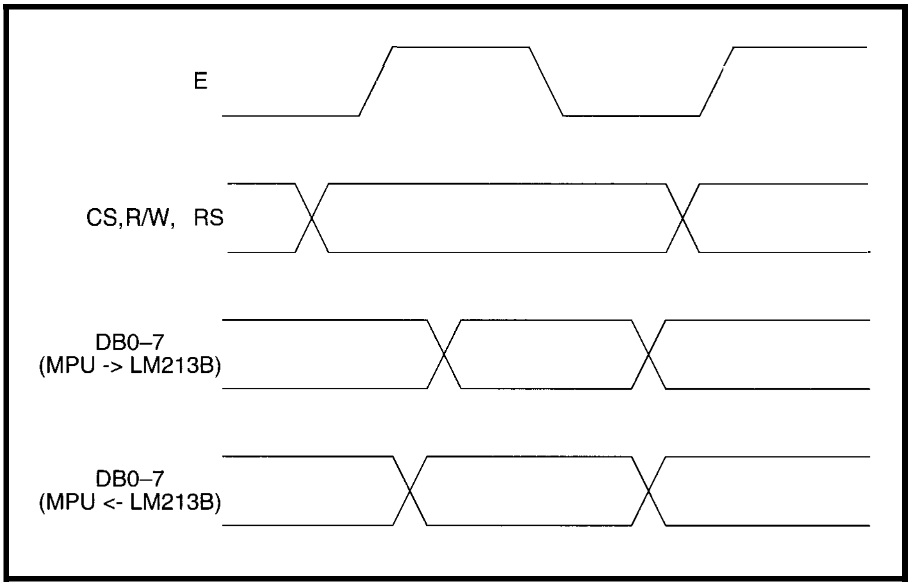


Figure 2— *During a typical data transfer cycle, E (Enable) is asserfed to clock data to and from the LM213B.*

The rest of the signals connect the various power supplies, *Vss* (ground) and *Vdd* (+5 volts) for the ICs, and *Vee* (-9 volts) and *Vo* (≈10.2 to 13.2 volts) for the LCD drive. Figure 3 shows how they should be connected. In particular, note the trimpot between *Vdd* and *Vee* generating the adjustable *Vo*. The trimpot works like a contrast dial, even though the viewing angle is what is actually being changed. At extreme contrast adjustment, the LCD display panel will appear either completely dark or completely light (from a given viewing angle), which certainly complicates software development and debugging. Verify early that the contrast setting will let you see what is going on to avoid the "calling for TV repair service when the set isn't plugged in" syndrome.

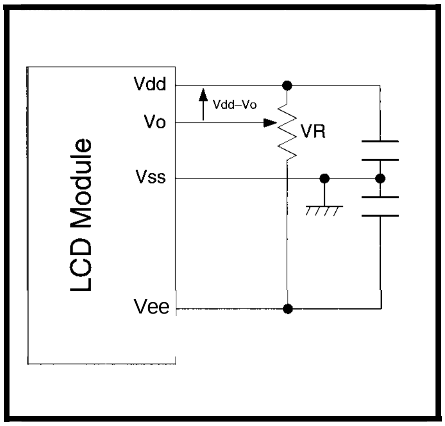


Figure 3— Power *connection includes a trimpot between Vdd and Vee that works like a contrast dial.*

Now that I've explained the official specification, I can give you a couple of I/O line-saving tricks. First, running the LM213B in write-only mode is quite possible, meaning you can simply ground the *R/W** line. The main reason for reading the LCD (although there are others, which I will discuss later) is to check a "BUSY" bit from the LCD before issuing a new command. However, the LM213B is fast enough that a typical system may never encounter BUSY. Even if your system is fast enough to run into BUSY, you can prevent it from doing so simply by introducing a delay between commands in software.

Similarly, you can ground *CS** as well to free up another bit. Because *E* is what actually causes data transfer to occur, the LCD won't do anything unexpected when permanently selected as long as *E* remains low.

PPI FOR THE MASSES

The controller I'm driving the LCD with is a Micromint RTC180, which includes an HD64180 CPU, up to 96K of memory, an eight-channel ADC, a two-channel serial I/O (RS-232 and RS-485), and an 8255 PPI with 24 TTL I/O lines.

The board includes a stacking expansion bus with all the signals needed to connect to the LCD directly (indeed, an add-on board, the RTC-LCD, is available for just that purpose). However, because my LCD is mounted in a small case with the RTC180, I didn't have room to use a stack-on board. I also preferred to adopt a less hardware-specific interface so I could use the LCD with different systems in the future. Finally, I know from experience that although the LCD is reasonably fast in human terms, it is slow enough that byte-level access speed is not normally a bottleneck.

Therefore, I opted to use the 8255 PPI for the interface instead of direct microprocessor bus connection. A control system that doesn't offer an 8255 or similar TTL I/O chip is rare, so porting my driver software to a different computer or controller is relatively easy. On the other hand, switching an interface from one microprocessor-specific bus to another would undoubtedly call for dragging out the wire-wrap gear.

The 8255 PPI features a number of operating modes that vary the 24 I/O lines' functions. I set up the connection between the 8255 and the LCD using one 8-bit port for the data bus (*DB0–DB7*) and 4 bits of a second port to drive *E, RS, R/W**, and *RES* (see* Figure 4). Using 12 I/O lines is preferable to 13 because the 8255 handles its ports in 4-bit chunks, so I took advantage of the previously mentioned trick and simply grounded CS *.

BEYOND THE BASICS

The RTC 180 includes a nice ROM-based BASIC. Unlike most other BASICs that are interpreters, BASIC-180 is actually a compiler and makes fairly speedy programs.

While I'm a pretty big fan of BASIC (I realize that isn't considered

8255—LM213B

PAO-DO	PB4—RS
PA1—D1	PB5—R/W
PA2—D2	PB6—E
PA3—D3	PB7—RES*
PA4—D4	
PA5—D5	
PA6—D6	
PA7—D7	

note: LM213B CS* input is grounded.

Figure 4— *The 8255-LCD connection uses Port A for the d-bit data bus and Port B for the control signals.*

"technically correct" today, but I'd rather be known as an independent thinker), I must admit the typical language constraints of short variable names, mandatory line numbers, weak looping constructs, and so forth are wearisome.

Therefore, I've been working on a program over the past couple of years called the BASIC Developer's Tool, or BDT. It combines editor, preprocessor, debugger, and terminal emulator in one easy-to-use tool for BASIC-based SBC development.

I don't want to go into a full discussion of BDT here [but I will add doing one to my *Things To Do* list). For now, I'll just explain why the code listings I give here look as if they are written at a somewhat higher level than the typical BASIC program.

Back to the LM213B. The first step is to reset the LCD and initialize the 8255 to the proper mode. Next, you need an lcdout routine that sends a byte to the LCD (so far, I was still using write-only software, even though the *R/W** line was connected). Listing 1 shows the code to accomplish these tasks.

At this point, the LCD is on the air and you can talk to it, so I guess now is the time to describe just what to tell it to do.

YOUR WISH IS MY COMMAND

As I mentioned earlier, the LCD has two registers: instruction *(RS* = 1), and data *(RS = 0).* In almost all cases, writing to the instruction register (to specify the operation) then to the data register (to specify the operands) performs operations. Figure 5 summarizes the operations and operands supported by the LCD.

Now take a look at Listing 2, which is a listing of the code necessary to set up the LCD in graphics mode. This loop reads each byte in the **DATA** statement, toggles *RS,* and sends the byte to the LCD. Thus, the **DATA** list is interpreted as a sequence of eight opcode-operand pairs (e.g., the first pair is opcode = O/operand = 32H, the next is opcode = I/operand = 7, and so forth). By referring to Figure 5 and Listing 2, you can understand the following description:

- opcode = O/operand = 32H

Turns the display on, specifies master mode (which a single LCD setup always uses), and selects graphic (vs. character) mode.

- opcode = I/operand = 7

Specifies the vertical and horizontal character pitch. In graphics mode, the vertical pitch [most-significant nybble) is meaningless, so it is set to 0, while the horizontal pitch specifies the number of bits to be displayed for each byte of display memory. Naturally the answer is 8, but note that the LCD wants to see "pitch minus 1" (i.e., 7) in the register.

- opcode = 2/operand = 31

Specifies the number of horizontal bytes to be displayed. Because the LCD has a 256 horizontal x 64 vertical bitmap (2K bytes) and each byte displays 8 bits, the number of horizontal bytes is 32 (32 x 8 = 256). Once again, the LCD expects "bytes minus 1," so 31 is written.

- opcode = 3/operand = 63

Sets the display duty cycle, which varies for different displays. The LM213B calls for 1/64 duty cycle, so "duty cycle minus 1" (63) is written.

- opcode = S/operand = 0, opcode = 9/operand = 0

Opcodes 5 through 7 aren't defined, so opcodes 8 and 9 are next in the sequence. These specify the least- and most-significant portions, respectively, of the display start address, which is the one corresponding to the top left corner of the LCD. Note that in graphics mode the address is interpreted as a bit address (allowing smooth scrolling) while in character mode it is a byte address.

- opcode = OAH/operand = 0, opcode = OBH/operand = 0

Similarly, these specify the cursor address. Even though a cursor isn't displayed in graphics mode, this address needs to be set or manipulated because the cursor address is also the address to and from which screen data is written and read. In either graphics or character mode, the address is interpreted as a byte address.

DOTS NOT ALL FOLKS

If you're lucky, at this point the LCD screen looks like garbage. Don't worry. What you're seeing on power up is just the random data in the 2K frame buffer.

Therefore, before you go further, you should make a *clear screen* command (see Listing 3). First, this routine uses the previously mentioned set cursor address commands (OAH, OBH) to return the cursor to home (top left). Next, the **FOR/NEXT** loop uses

Listing 1— The 8255 *setup and LCD reset code should be run before using the subroutine to send data.*

```
   OUT pmode,$80 'set 8255 ports to output'
   OUT pb,0 'set LCD RES* low-reset the LCD'
   OUT pb,$80 'set LCD RES* high, RS,R/W,E low'

 'subroutine to output a byte to the LCD'
PROC lcdout  'output lcdbyte to the LCD port determined by RS
               (1=cmd,0=data)'
INTEGER pbtmp 'work var for 8255 portb manipulation'
BEGIN 'lcdout'

 '8255 porta and high 4-bits(d7-d4) of portb connect to LM213 LCD
  porta = 8-bit data to DO-D7
  portb  =  d7=RES*, d6=E, d5=R/W*, d4=RS'

   OUT pa,lcdbyte '8255 porta=output byte'
   pbtmp=(rs*16)+128 'RES*=high,E=low,R/W=low,RS=rs'
   OUT pb,pbtmp
   pbtmp=pbtmp+64 'RES*=high,E=high,R/W=low,RS=rs'
   OUT pb,pbtmp
   pbtmp=pbtmp-64 'RES*=high,E=low,R/W=low,RS=rs'
   OUT pb,pbtmp

END  'lcdout'
```

Opcode O-Mode Control
Establishes basics such as cursor on/off, cursor/character blink, and, most importantly, character (on-board character generator) or graphics (bit mapped) mode.

Opcode l-Set Character Pitch
Specifies the number of vertical and horizontal dots displayed per character. In graphics mode only the horizontal pitch-specifying the number of dots displayed for each byte of frame buffer (typically 8)-is meaningful.

Opcode P-Set Number Of Characters
Specifies the number of characters (character mode) or bytes (graphic mode) displayed in the horizontal direction. A likely choice for the former is 42 and the latter 32.

Opcode 3-Set Time Division
Sets the LCD display multiplexing duty cycle which typically ranges from 1/32 to 11128 depending on the LCD specification.

Opcode 4-Set Cursor Position
This doesn't move the cursor (see opcodes 0Ah and OBh)-rather, it specifies the vertical position within a character cell that the one-line (horizontal character pitch wide) cursor occupies.

Opcode 8-Set Display Start Low Address
Opcode 9—Set Display Start High Address
These specify the address in the frame buffer that corresponds to the top left corner of the display. In character mode, it is a "character" address while in graphic mode, it is a "dot" address.

Opcode OAh-Set Cursor Low Address
Opcode OBh-Set Cursor High Address
These move the cursor to the specified "character" address (character mode) or "byte" address (graphic mode). This is the address that will be accessed by subsequent read (opcode 0Dh) and write (opcode 0Ch) commands.

Opcode OCh-Write Data
The 8-bit operand is written to the frame buffer at the cursor address and the cursor address is automatically incremented.

Opcode ODh-Read Data
So far, this is the first opcode to require "reading." It returns the 8 bits of data at the cursor address and automatically increments the cursor address. Watch out-the first data read immediately after the cursor address has been set (opcodes OAh,OBh) doesn't return good data-a second read is required.

Opcode OEh-Clear Bit
Opcode OFh-Set Bit
In graphic mode, the cursor address points to a "byte" of frame buffer. These commands specify which bit in the byte is modified.

Busy Flag Read
A read of the instruction register (i.e., R/W*=1,RS=1) returns the "busy" flag status in the most-significant bit (1 =busy, O=not busy). In principle, the status should be checked to make sure the HD61830 is "not busy" before an instruction is issued. Alternatively, software delays can ensure the HD61830 will never appear "busy" if "write-only" operation is desired.

Figure 5— *Most* LCD command sequences use a single-byte opcode *followed by a single-byte operand; note that opcodes 5 through 7 are undefined.*

the *write data* (0CH) command to clear [fill with 0s] the 2K frame buffer. The set cursor address command isn't needed for each byte write; it is a property of the write data command to increment the cursor address automatically each time. After the last write into the frame buffer, the cursor position automatically wraps around to the home position (i.e., . ..2046. 2047, 0).

This step should finally give you a blank screen. Hopefully, this moment isn't when your spouse asks to see a demo of what you've been working on for "so long." Better get something on the screen right away, so check out Listing 4, which is a dot routine.

On entry, the **dot** routine expects a dot address in variables **x** and y. This address is a "bit" address (i.e., x = 0 to 255 and y = 0 to **31).** In addition, a variable onoff specifies whether the dot should be turned on **(onoff = 1) or** off(onoff = **0).**

The first four lines take the x,y bit address and convert it to a byte address and bit offset. Next, the *set cursor*

Listing **2—** *A sequence of eight opcode-operand pairs initializes the LCD to graphics mode.*

```
'LCD init data'
DATA 0,$32,1,$77,2,$1f,3,$3f,8,0,9,0,$a,0,$b,0

FOR i=1 TO 16 'send the LCD initialization string'
  READ lcdbyte: rs=BXOR(rs,1): GOSUB lcdout
NEXT i
```

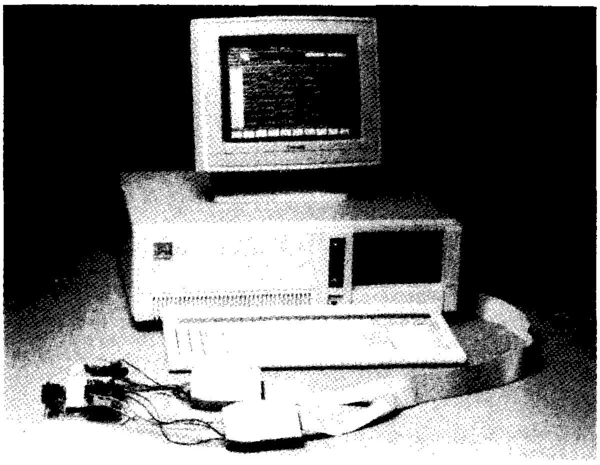
Listing **3—** The *screen display is c/eared by filling the 2K frame buffer with 0s.*

```
PROC lcdclr 'clear the LCD screen'
INTEGER i  'for/next counter'

BEGIN 'lcdclr'
  lcdbyte=$0a: rs=1: GOSUB lcdout 'move to home (0,0) position'
  lcdbyte=0: rs=0: GOSUB lcdout 'addr low = 0'
  lcdbyte=$0b: rs=1: GOSUB  lcdout
  lcdbyte=0:  rs=0: GOSUB lcdout 'addr high = 0'

  FOR i=0 TO 2047 'screen is 2K bytes (256*64 bits)'
    lcdbyte=$0c:rs=1:GOSUB lcdout 'write data command'
    lcdbyte=0:rs=0:GOSUB lcdout 'write 0s'
  NEXT i

END 'lcdclr'
```



address commands (opcodes OAH and OBH) are issued to specify the low and high bytes of the computed byte address. The cursor is now pointing to the byte that contains the bit (dot) of interest. To write it, the last two lines of the routine issue either a *clear bit* (OEH) or a *set bit* (OFH) command (determined by onoff=0 or 1, respectively) whose operand is the previously computed bit offset within the byte address. For example,

```
onoff=1: x=255: y=31
GOSUB dot
```

will turn on one lonely dot at the bottom right corner of the LCD. This feature isn't very impressive, so maybe you'd better hold off on the demo for a little while longer.

TOEINGTHE LINE

At this point, you can do almost anything you want if you're willing to type in a zillion x,y dot pairs. However, there is a better way.

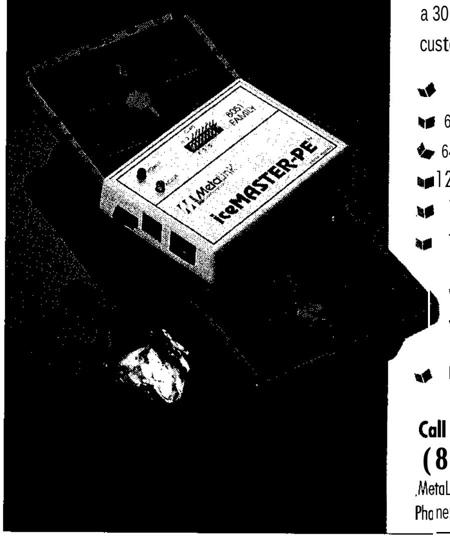
The next step up the functionality ladder is to implement a line routine that simply takes a start (x,y) and end (x1,y1) address and fills everything in between automatically. Time to dust off the old "computer graphics" textbooks.

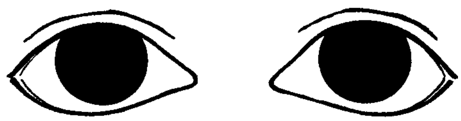
Sure enough, a basic class of algorithms for drawing lines (and other figures such as circles) does exist and is called *digital differential analyzers* (DDA). As the name implies, the DDA algorithms work by generating the points to be drawn from the differential equation of the figure. This feature works particularly well for lines whose equation is simply

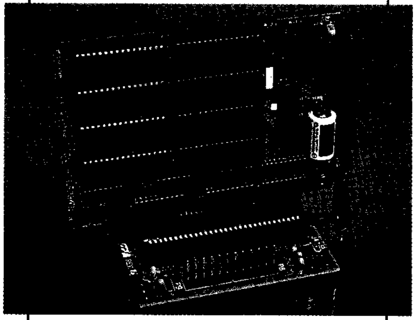
```
dy/dx = y/ x
```

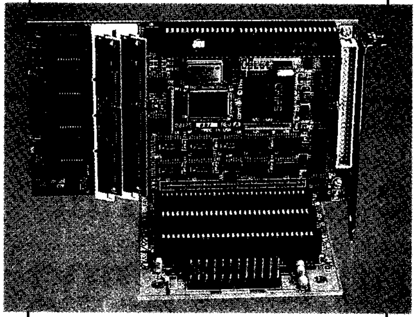
Using my variable names, x = ABS(xx1) and y = ABS(y-y1), which is the absolute value of the difference between the start and end points in each axis. The DDA works by repeatedly incrementing x and y by a factor based on x and y, respectively:

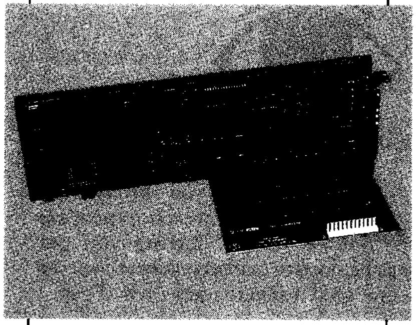
```
Current x = (factor * x)
          + Previous x
```











Listing 4—By specifying x and y bit addresses, any dot on the LCD display can be turned on or off.

```
PROC dot 'turn the dot at x,y on (onoff=1) or off (onoff=0)'
INTEGER  byte, 'byte address of dot'
         bit,  'bit offset of dot within byte'
         tmpx 'holder for x'
BEGIN 'dot'
'an x(0-255),y(0-63) coordinate is converted into
 a byte address and bit offset
 the cursor is moved to the byte address and
 the dot at bit offset is turned on'

  tmpx=x
  bit=x-((x/8)*8) 'det. remainder (bit offset) from byte x addr'
  tmpx=tmpx/8 'determine byte x address'
  byte=(y*32)+tmpx 'each horizontal line has 32 bytes (32*8=256)'
  lcdbyte=$0a:rs=1:GOSUB lcdout 'set addr low byte command'
  lcdbyte=byte 'dot byte addr low'
  rs=O: GOSUB lcdout
  lcdbyte=$0b:rs=1:GOSUB  lcdout  'set addr high byte command'
  lcdbyte=byte/256 'dot byte addr high'
  rs=O: GOSUB lcdout
  lcdbyte=$0e+onoff:rs=1:GOSUB lcdout 'clear or set bit command'
  lcdbyte=bit:rs=0:GOSUB lcdout 'turn the dot on'
END 'dot'
```

Listing 5—The Bresenham line-drawing algorithm can be implemented with on/y basic math operations.

```
'Bresenham line algorithm'
PROC line  'draw a line from x,y to x1,y1'
INTEGER e, 'error accumulator for Bresenham algorithm'
        dx,dy, 'distance in x&y direction'
        absdx,absdy,  'absolute values of dx,dy'
        dirx,diry,  'left/right (dirx), up/down (diry) dir. flag'
        dotmaj,dotmin,  'major/minor axis dot address holder'
        dmaj,  'distance in major axis direction'
        absdmaj,absdmin,'absolute value of dmaj,dmin'
        dirmaj,dirmin,  'major/minor access direction flags'
        savex,savey, 'x,y (and x1,y1) unchanged by this routine'
        i  'misc. for/next counter'
BEGIN 'line'
  savex=x:savey=y  'save x&y'
  dx=x1-x:dy=y1-y  'compute distance in x&y directions'

'First, determine the direction and absolute value
 of the distance in x axis...'

  IF dx<O THEN BEGIN
    absdx=-dx:dirx=-1 'left'
  END
  ELSE BEGIN
    absdx=dx:dirx=1  'right'
  END

'...and the y axis'

  IF dy<0 THEN BEGIN
    absdy=-dy:diry=-1 'up'
  END
  ELSE BEGIN
    absdy=dy:diry=1  'down'
  END

'Next determine the lines major and minor axis
 and assign the appropriate values ex:dmaj=dx/dmaj=dy'
```

(continued)

Listing 5—*continued*

```
  IF absdx>=absdy THEN BEGIN

'x is major axis'

    dmaj=dx 'major axis distance'
    dotmaj=x:dotmin=y 'major/minor axis dot address'
    absdmaj=absdx:absdmin=absdy  'abs values of major/minor axis
distance'
    dirma¸i=dirx:dirmin=diry 'major/minor axis direction flags'
  END
  ELSE BEGIN

'y is major axis'

    dmaj=dy
    dotmaj=y:dotm  n=x
    absdmaj=absdy  absdmin=absdx
    dirmaj=diry:d  rmin=dirx
  END

'Finally, the Bresenham algorithm draws the line'

  e=(2*absdmin)-absdmaj 'initial ire error accumulator'
  FOR i=O TO dmaj STEP dirmaj 'step in the major axis'
    IF absdx>=absdy THEN BEGIN  if x is major axis'
      x=dotmaj:y=dotmin  'assign dot addresses'
    END
    ELSE BEGIN 'y is major axis
      x=dotmin:y=dotmaj  'assign dot addresses'
    END
    GOSUB dot 'draw each dot'
    IF e>O THEN BEGIN 'if error passes threshold'
      dotmin=dotmin+dirmin  'make a move in minor axis'
      e=e+((2*absdmin)-(2*absdmaj)) 'and reset the error
accumlator'
    END
    ELSE e=e+(2*absdmin) 'otherwise, keep accumlating error'
    dotmaj=dotmaj+dirmaj  'and make a move in major axis'
  NEXT i

  x=savex:y=savey 'restore x&y'

END 'line'
```

Current y = (factor * y)
+ Previous y

But what is the right value for the factor? A few extremes will illustrate how the algorithm works. If the factor is **1,** the algorithm will simply generate two points, only x,y and x1,y1—the specified start and end points.

On the other hand, if the factor is infinitely small, the algorithm will generate an infinite number of points between x,y and x1,y1. This makes for a line drawn very accurately, but only if the display has infinite resolution!

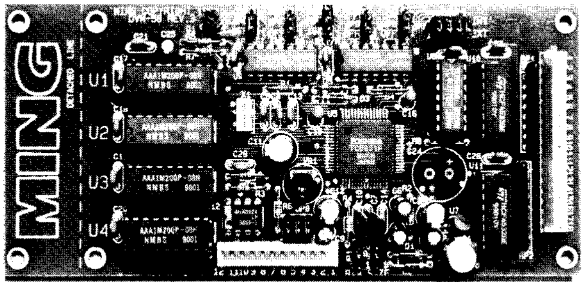
The right choice for the factor is one that generates neither too many points (i.e., unneeded computations) nor too few (i.e., gaps in the line] because the display resolution is finite. These conditions imply that the current and previous point coordinates should always differ by 1 for either the x or the y axis, but which one? A little thought reveals that the answer is the "major" axis-the one in which most movement occurs. Therefore, choosing a factor equal to the reciprocal of " major" axis is convenient. Then, each step through the point plotting loop will increment the major axis address by 1, which implies the minor axis address will be incremented by less than **1.** However, assuming real numbers are used for *x,y* addresses, the fraction accumulating in the minor

ELECTRONICS 123
A DIVISION OF MING Engineering & Products, Inc.

ORDER TOLL FREE
I-800-669-4406
FAX ORDERS : (818) 912-9598

DVM-58C w/ 1 Meg DRAM $89.95

The DVM-58C with its variable length message capabilities can store up lo 16 individual messages for immediate playback to enhance most any electronic application or project. Use it to add real voice prompting or instructions at an affordable price.



DIGITAL VOICE PRODUCTS

DVR-120M The DVR-120M is a complete digital record/playback unit that incorporates the DVM-56C within an ABS enclosure, including all the necessary components required for immediate use. $149.95

DVR-240 The DVR-240 is an entirely digital, single channel, looping player, which will continuously cycle the entire message without suffering from the effects of tape wear and audio degradation. $249.95

DVM-5805 The DVM-5805 digital voice module is capable of record/playback of up to 5 messages from its 32 second DRAM memory. $49.95

DVM-2804 A compact digital voice module designed for playback of up to 32 seconds of prerecorded messages from a single EPROM IC, which must be programmed with our DVMP Programming Tool. $49.95

DVMP The DVMP programming tool is a non-PC based system designed to be used with your existing EPROM programmer for the purpose of transferring voice data onto EPROM IC's that are used with the DVM-2804. $495.95

FLUKE DMM Series 10

FLUKE 10

FLUKE 11

FLUKE 12

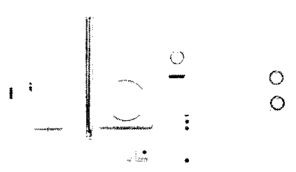
- AC & DC voltage measurements to 600V.
- Ohms to 40 Meg & diode test.
- 4000 count digital display.
- Auto & manual ranging.
- Fast continuity beeper.
- Reliable by design.

FLUKE 10	$69.95
FLUKE 11	$79.95
FLUKE 12	$89.95

We stock a Full Line of X-I 0 Accessories

- Lamp Modules
- Appliance Modules
- Floodlight Control
- Receptacle Module
- Remote Sirens
- Mini Timer
- Wireless Motion Detectors
- Wall Switch Modules
- Telephone Responder
- Wall Switch Module
- Burglar Alarm Interface

X-10 POWERHOUSE (SS5400)

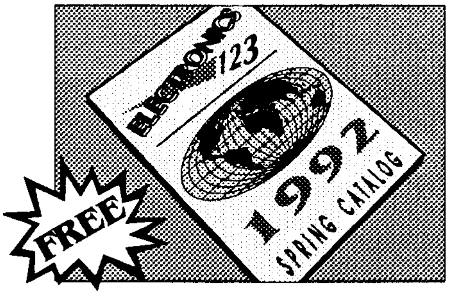


A wireless 4 piece home security system that installs in minutes without tools or wiring. The professional features and reliability of the SS5400 can protect up to 16 doors I windows and control up to 8 remotely operated lights simply by adding additional X-I 0 PowerHouse modules to the system.

SYNTAX PROTOTYPING PCS
FOR ALL PC-BUSES: AT, MULTI, PS, S-100, VME, & XT.
FOR ALL PC-BOARDS: IC, TRANS, & UNI.

RADIO COMMUNICATIONS
MOTOROLA SPIRIT I, II, & III, & all the accessories.

BATTERIES
Panasonic Alkaline & Ni-Cd, Power sonic scaled lead-acid.



SECURITY SYSTEM
Vehicle alarm, PIR, sirens, & home security.

TESTING EQUIPMENT
IC Tester, Fluke Meters.

DIGITAL VOICE PRODUCTS
DVM-2804, DVM-58C, DVM-5805, & more.

RF TRANSMITTER & RECEIVER
TX-88 RF Transmitter
RE-99 RF Receiver

F1208D1

MAIL ORDERS:

ELECTRONICS 123
17921 Rowland Street
City of Industry, CA 91748

TECHNICAL SUPPORT
(818) 912-9864

ALL PRICES SUBJECT TO CHANGE WITHOUT NOTICE

VISA. WC 6 DISCOVER CREDIT CARDS ACCEPTED

FREIGHT CHARGES ADD:	
UPS - Ground	$ 5.00
Second day (blue)	$ 8.00
Next Day (red)	$15..00
C.O.D. ADD:	$ 4.00
CALIF. RESIDENTS ADD:	8.25% Sales *Tax*

CANADIAN ORDERS:
All orders must be PREPAID by check or credit card (VISA, MC).
Add a flat rate of US$15.00 for shipping and handling by U.S. POST OFFICE (air).

#140

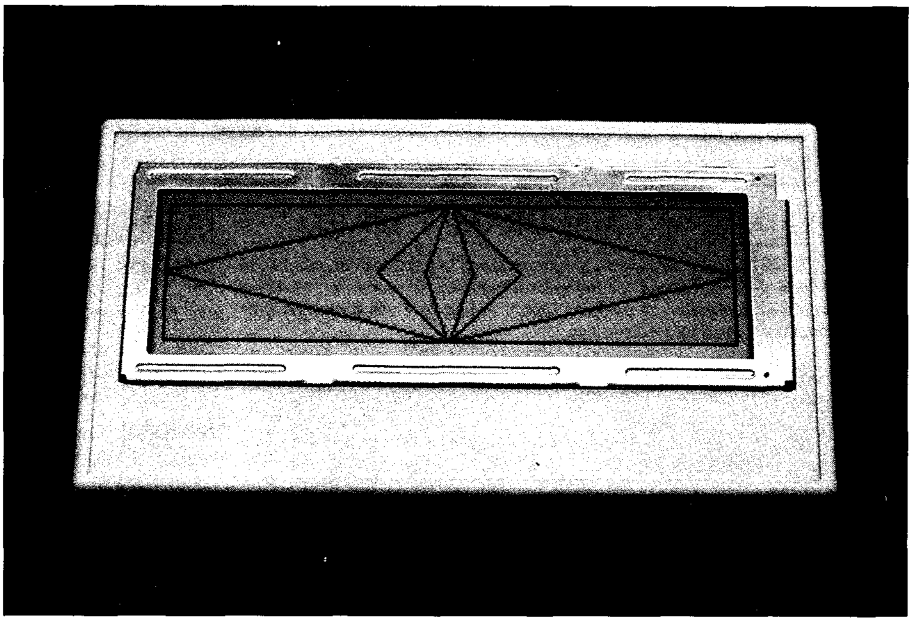


Photo 1— The test pattern covers boundary conditions by specifying start and end points for lines that write to the corners and edges of the LCD.

axis address will eventually cause the integer portion to increment. For example, a sequence of iterations through the loop might look like

```
;Assume X is major axis
Iteration CalcX CalcY  PlotX,Y
    1       0    0         0,0
    2       1    0.25      1,0
    3       2    0.50      2,0
    4       3    0.75      3,0
    5       4    1         4,1
    6       5    1.25      5,1
```

and so forth. The main problem with this so-called simple DDA is the need to perform real math (divisions no less) at each point on the line.

BRESENHAMTOTHERESCUE

Fortunately, J. E. Bresenham of IBM wrote a paper in 1965 describing a simple line-drawing algorithm requiring no real math or divide operations, only integer addition, subtraction, and multiplication. Indeed, even the multiplication isn't needed because the algorithm only multiplies by 2, which is a simple left shift.

The algorithm is similar to the previously described DDA in which it steps in the direction of the major axis while incrementing the minor axis less frequently. The difference is that instead of carrying or computing a fractional move in the minor axis, it keeps track of an error term that represents the difference between the integer minor axis address that is plotted and the real (noninteger) minor axis address that would be computed by a DDA. By algebraic gyrations (multiplying everything by a constant-namely, 2 times the major axis], the error term need not involve fractional computation. Instead, only the sign (positive or negative) matters. If the sign of the error is positive, the minor axis address is incremented, otherwise it isn't.

Without further ado, Listing 5 shows my implementation of the Bresenham algorithm. First, I determined the absolute values of x and y and a corresponding direction flag. Next, I figured out which axis was major and then assigned the various parameters appropriately (because the main FOR/NEXT loop of the algorithm must step in the major axis). Finally, the algorithm went to work, computing and plotting each point along the line from x,y to x1,y1.

As an example of applying the algorithm, Photo 1 shows the results of a program that creates a test pattern of various lines including horizontal, vertical, 45-degree, x-major, and y-major. Besides testing each type of line, the program clears each line by going in the opposite direction. Drawing a given line in both directions (i.e., x,y to x1,y1 and x1,y1 to x,y) and confirming that they are exactly the same is important because subtle bugs or typos could cause the routine to generate a slightly different line in each direction (a real head scratcher).

NEED FOR SPEED

Well, I finally got it working, but I must say the performance was a little pokey-about 600 dots per second. This rate may sound fairly speedy, but considering that the whole screen contains 16K dots, you would need almost 30 seconds to fill it with lines.

Time to optimize!

First, I rewrote the 1 cdout, 1 cdc 1 r, and dot routines in assembly language, taking advantage of BASIC-180's coROM that make calling and passing variables back and forth easy. Using these features wasn't very hard; I only needed about 100 lines of code.

Next, I optimized the line-drawing algorithm itself. While the Bresenham algorithm certainly works for horizontal, vertical, and 45-degree lines, using it for such cases is overkill because these lines are easily implemented as "step x, hold y constant," "step y, hold x constant," and "step x and y" loops, respectively.

Together, the optimizations speed things up by about a factor of 10 and the lines fly onto the screen quickly. You'll find my final code on the Circuit Cellar BBS. With this foundation, I hope to add more features like circle or arc commands, bitblt (bit block transfer), bitmapped fonts, and so forth.

I now know why my *Things To Do* list never gets shorter. Every time I take something off it, I add two or three others! Oh well, if the rule is "Whoever dies with the biggest *To Do* list wins!" I'm destined to be awarded the prize. ❑

Tom Cantrell holds a B.S. and M.B.A. from UCLA and has been in Silicon Valley for more than ten years working on chip, board, and systems design and marketing. He can be reached at (510) 657-0264 or by fax at (510) 657-5441.

REFERENCES

Principles Of Computer Graphics, Newman and Sproul, McGraw-Hill

"Algorithm For Computer Control Of A Digital Plotter," J. E. Bresenham, *IBM System Journal, 1965*

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

I R S

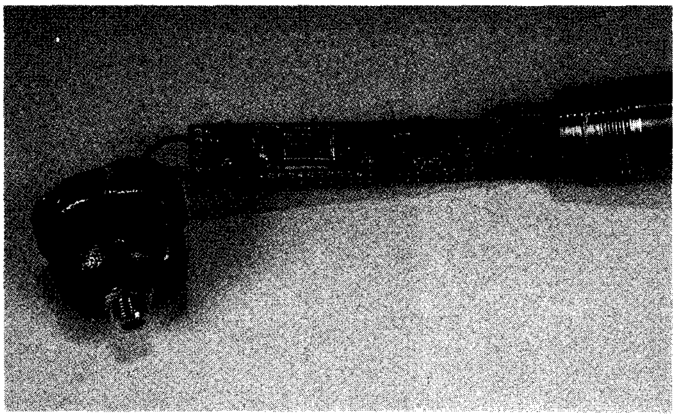
410 Very Useful
411 Moderately Useful
412 Not Useful

DESIGN CONTEST WINNER

project descriptions
by Lisa Nadile

All too often, the creative aspect of our work as computer specialists goes unrecognized. The ability to imagine a project and follow through its development to its fruition is a talent not universally possessed by everyone in our field. During the 500th anniversary of the discovery of America, we salute those who strive to breaknewground. Each project you see pictured on these pages is the product of one manifestation of an explorer: the inventor. Below, we present the winners of the Fourth Annual Circuit Cellar Design Contest, whoareperfectexamplesofwhymanyconsidertheingenuity and creativity of the scientist the modern day pioneering spirit.

As in past years, we plan to ask each winner to write up their project as a full-length article to appear in future issues of the Computer Applications Journal. *For example, the SmartROM EPROM emulator project featured in this issue was a second place winner in last year's contest, while the robot sensor featured in the last issue was a first place winner last year. The photos and brief descriptions we present here are sure to pique the interest of more than a few of you. However, we can't give out addresses or phone numbers of the winners to allow you to contact them for more information about theirproject. If you just can't wait for a feature article, then you may write to any of the winners in care of us and we will be sure to forward your letter. Send inquiries to Design Contest Winner, The Computer Applications Journal, 4 Park St., Vernon, CT 06066.*



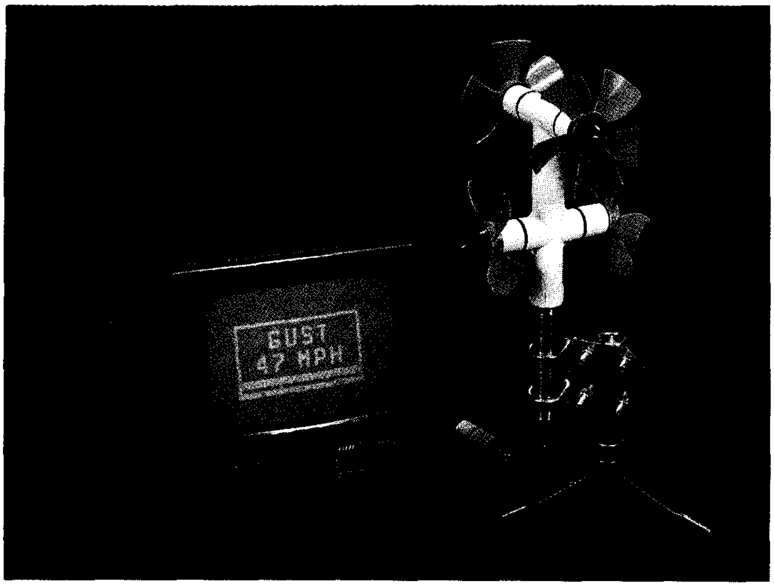
FIRST PLACE: $500

A Single-DIP Video Wind Gauge
by Philip C. Pilgrim

Our winner clearly stood out for its efficiency of design, low-cost components, and imaginative application. Philip's approach to building a wind gauge is the product of his wish to display the results of its measurements in a decidedly unique way: using an inexpensive NTSC video signal and a single microcontroller with no other ICs. Could it be done? Yes! He built the gauge using an 18-pin Microchip Technology PIC16C56, which is capable of 5 million instructions per second from a 1024-word by 12-bit PROM. It has 32 8-bit registers (25 of which are general purpose), 12 bits of CMOS-level I/O, a real-time clock/counter, and a watchdog timer.

The wind gauge has two units. The first is for the roof and has four propellers and one PIC chip to measure the wind and generate the video signal. It uses a single RG-59 coax cable to carry power to the roof unit and the resulting video signal back to the monitor. The second unit is for the indoors and serves as a power supply and DC-restorer for the video signal.

The PIC chip is the only IC in both units, with the exception of the two voltage regulators, and is solely responsible for monitoring the propellers, converting the results into wind speed and direction, and generating the video signal that may be displayed on any composite video monitor.



FOURTH ANNUAL DESIGN CONTEST RESULTS

SECOND PLACE: $200

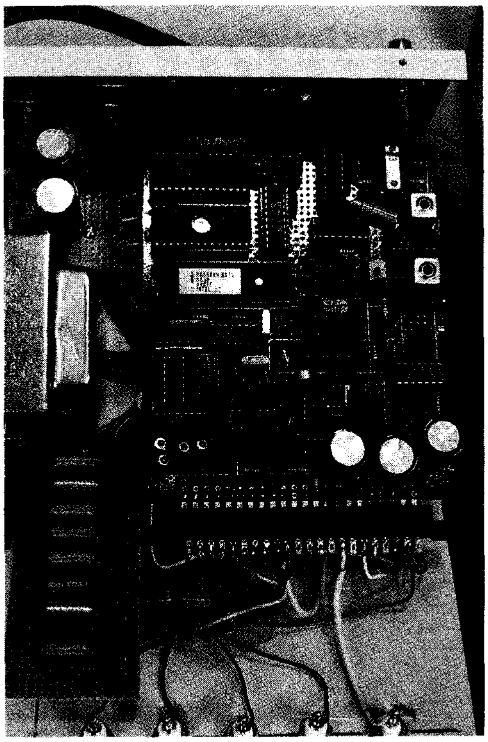
The Vertical Blanking Interval Explorer

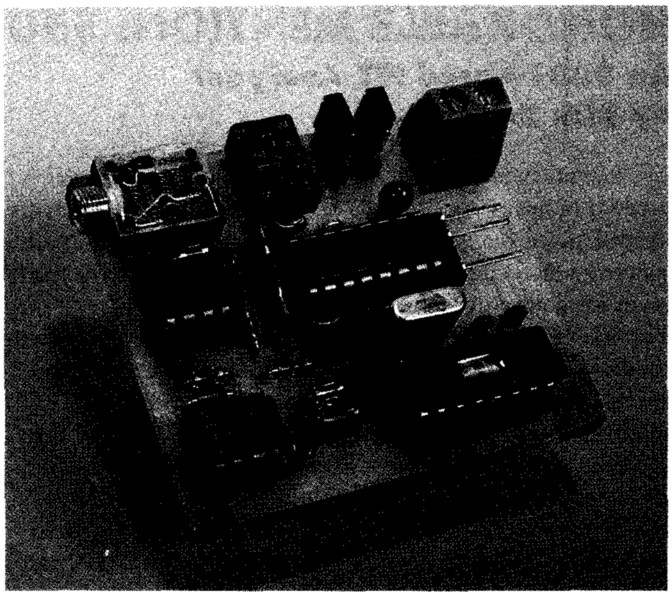
by Mike Barnes

One goal we at the *Computer Applications Journal* always strive for is to bring you the newest tools of our trade that are practical, economical, and have a high degree of functionality. Mike meets our goal by taking a new look at a device not often thought of as a computer specialist's tool: your television.

The Vertical Blanking Interval (VBI) Explorer is an 803 I-based project that allows you to analyze the first *21* lines of each video frame (the VBI) sent to your television. This interval is the portion of the picture you don't normally see unless you adjust the vertical hold so the picture rolls. Located in this space is a noninter-active teletext; the most common form in the U.S. is known as closed caption-ing. In addition to this aid for hearing impaired television viewers, you can find other digitally encoded signals in the VBI. For example, the three major networks carry a time-stamp containing the time and date. Mike found this signal, which is accurate to a second or two, a useful time reference for noncritical activities.

The VBI Explorer includes a serial port through which you interact with the unit as you explore the video signal. Connectors on the front allow easy access to the processed signal to make further exploration with an oscillo-scope easy.





THIRD PLACE: $100

Fluke 87 Data Acquisition

by Derek Matsunaga

In keeping with our theme of discovery, what is better than learning about a little-known capability on a familiar device? Unsup-ported and unpromoted functions are often present in the technology we use. Perhaps certain information remains undiscussed because it does not coincide with the marketing focus the company has given the product or because it is reserved for the new and improved promotions planned for the future.

The Fluke 80 series multimeter's capability to broadcast the information contained in its displays is one such hidden feature. Fluke uses this ultrasonic output for final testing at the factory, so the first goal Derek had was to figure out the code. This multimeter transmits display informa-tion through a small speaker using On-Off keying modulated at about 16 kHz. After figuring out Fluke's data format, Derek modified the multimeter with a small MC68HC05-based circuit to convert the data to ASCII and transmit it at 9600 bps in RS-232 format.

Derek has many applications for this multimeter's little-known feature, including using it to make true RMS measure-ments and as an automated data logger for environmen-tal measurements, such as temperature (with a thermal probe) or humidity, motor RPM modulation, and long-term power line monitoring. You can also use this multimeter as an automated lab notebook where you can record anything the meter measures.

HONORABLE MENTION: $50

An 8031-based FFT Analyzer
by Rick Smith

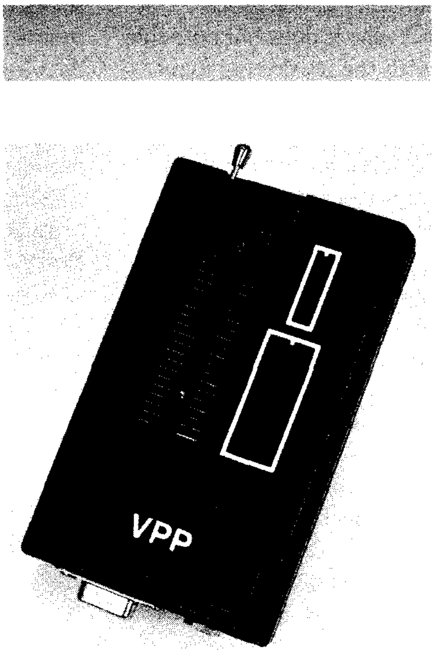
Ingenuity is often the product of necessity. Rick needed an inexpensive spectrum analyzer attachment for an oscilloscope. The 803 1 FFT Analyzer uses the 805 1 microcontroller for control and for performing math-intensive calculations of FFTs. The Analyzer uses an 8-bit Analog Devices AD7569 for an ADC, DAC, and sample-and-hold circuitry. Rick's project covers the entire audio range, providing selectable bandwidths of 5 kHz, 10 kHz, and 20 kHz. Three types of averaging modes and overrange indication are given, and all functions are accessible through the front panel.

The 803 1 FFT Analyzer is similar to a project we present in the August/ September '92 issue of the *Computer Applications Journal.* However, that project used a DSP chip to do the processing, and, in all fairness, Rick's project showed up at the same time that issue hit the stands, so neither of us knew what the other was planning.

HONORABLE MENTION: $50

The PIC Microcontroller Development System
by Les Hildenbrandt

Occasionally, we find ourselves in the middle of a project and the one part necessary for its completion is either hard to find or expensive. Such is the case with PIC development systems, so Les created his own. The system consists of a programmer based on an Intel 8748 and MS-DOS software, including an assembler and control software for the programmer. The system communicates with the PC through a 9600-bps RS-232 interface and will program the 18-pin PIC 16C54/56 and the 28-pin PIC 16C55/57.



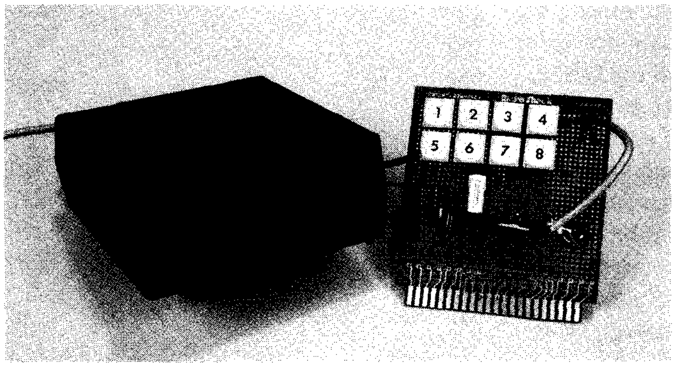
HONORABLE MENTION: $50

The Heathkit Digital Wind Speed and Direction Indicator
by David Penrose

We all have certain pieces of equipment we've relied on over the years and count among our group of favorite devices. Working with a favorite part feels as if you've just put on your old pair of slippers after a long workday-comfortable and familiar. When you find a device that you like, preserving its functionality over time is more than just maintaining parts in working order. There comes a time when updating your equipment is the only option.

David's project updates the Heathkit Digital ID-1590 Wind Speed and Direction Indicator, which he describes as "a strange mixture of the analog and the digital." Using reed switches and diodes, this remote sensing device encodes the wind speed and direction. It produces the wind speed on a digital readout and shows the wind direction a compass rose indicator.

The update meant bringing the Indicator into the computerized world with an RS-232 output that could be interfaced to any computer's serial port. David uses an inexpensive 8748 microprocessor and six alphanumeric LED displays.



SPECIAL SECTION

Russ Lindgren

Microvolt Measurements

Use a 20-bit A/D Converter in Your Next Design

Memory sizes continue to grow, data buses get wider, and effective resolutions on A/D converters keep increasing. Find out some tricks to getting 20 bits of precision from this ADC.

Making precise analog measurements requires skill and patience because drift and noise make attaining repeatability quite difficult. Add to this difficulty a slowly time-varying input signal, such as those found in process control, and measuring a rate accurately becomes almost impossible. The standard RS-232- or IEEE-488-based multimeters have difficulty handling this task because they require many front panel setups and special cabling. Furthermore, if postprocessing of the measurements is required, then how do you obtain readings accurate in both value and timing?

The design I present here offers real-life accuracy better than 18 bits and has the ideal data buffering to support digital filtering routines. With this design, all that's required to generate exceptionally precise measurements is good analog practice and some time spent coding a program to match the specific measurement and filtering needs.

My focus will be describing the analog circuits and digital logic of the design with an aim towards supporting software development. Overall, what is important is understanding the timing involved and coding real-time software to deal simultaneously with display, averaging, and running the hardware.

THE BEST ANALOG IS REALLY DIGITAL

The heart of the design is the AD7703, a Sigma-Delta A/D converter from Analog Devices. It offers 20-bit resolution and guarantees 16-bit linearity. Internally, it includes calibration routines and on-chip low-pass filtering, drastically reducing the

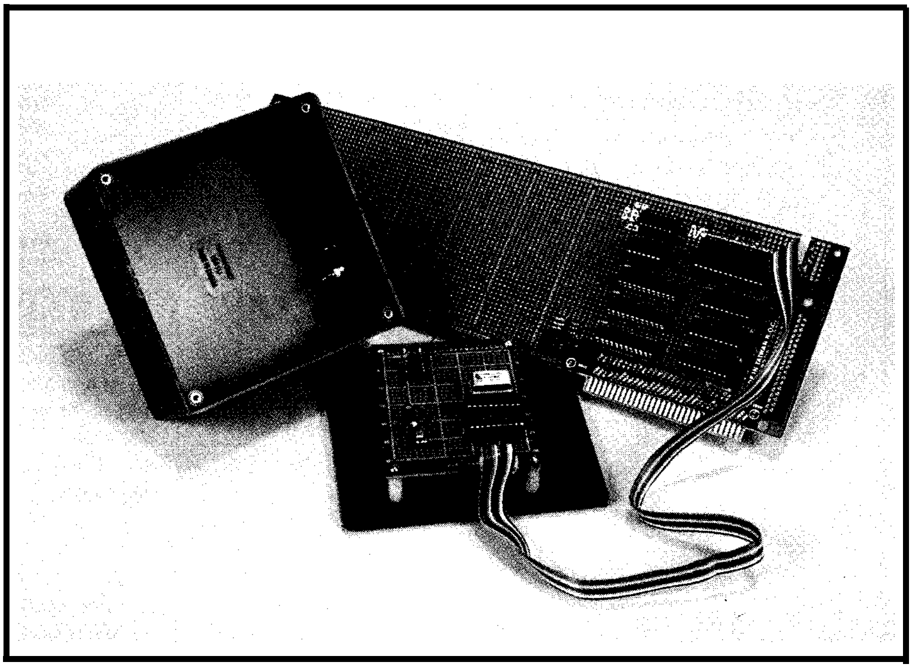


Photo 1—The main processor interface is on a plug-in board, but all the noise-critical analog circuitry and A/D converter are on a remote board connected to the main computer using a ribbon cable.

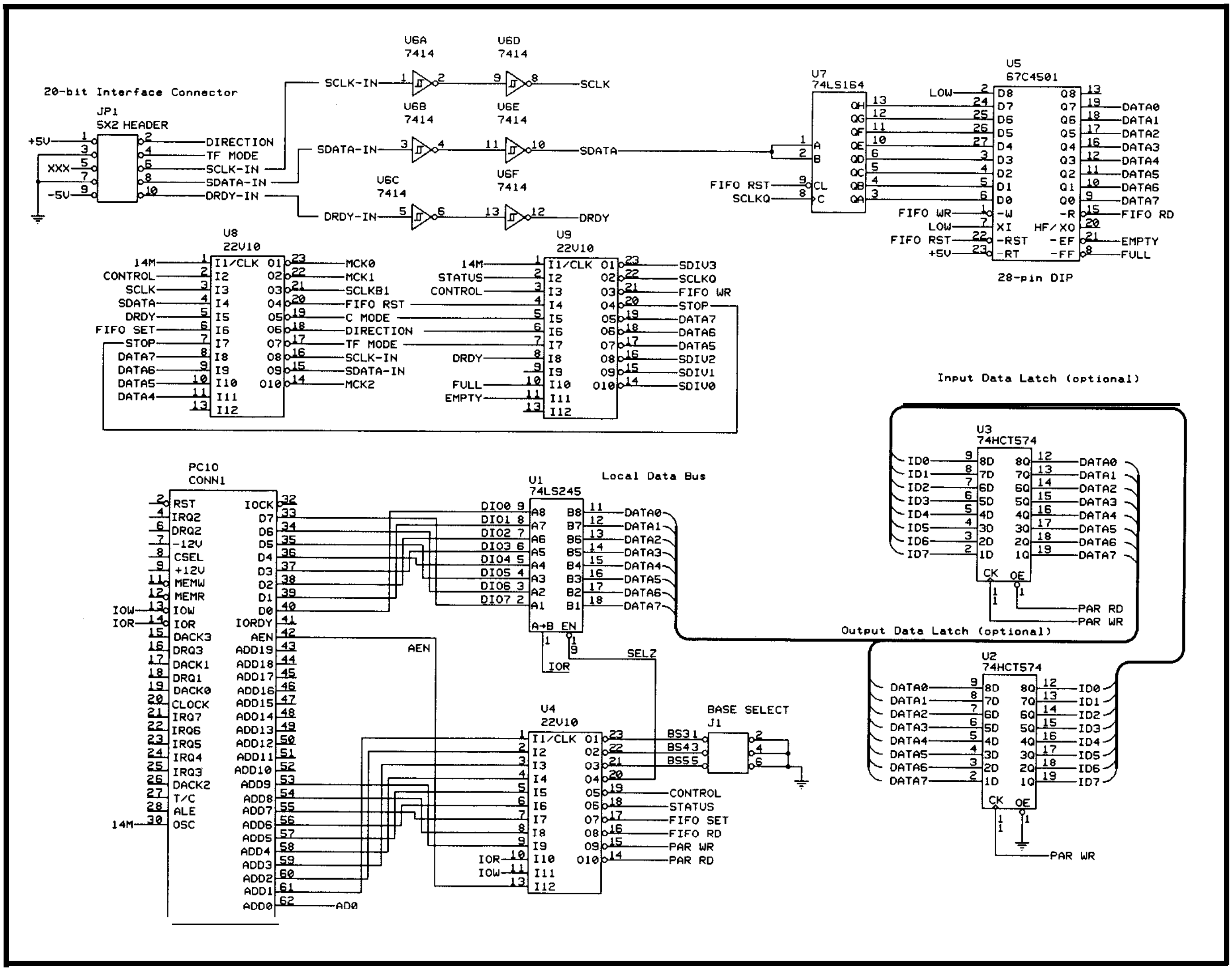


Figure I--Several *custom GALs* handle *all the tricky* timing involved *in* dealing with *the remote A/D converter. The PC bus interface consists of a buffer, some latches, and* another *GAL that* does decoding.

number of additional parts needed for a functioning system to two: a crystal or clock oscillator and a voltage reference. The capability to output a complete 20-bit measurement at a 4-kHz rate makes this part a phenomenal one. Precisely located in time and spaced 1024 clock cycles apart, each 20-bit measurement is ideal for digital signal processing.

The Sigma-Delta conversion technique makes this ADC an almost all-digital part. Sigma-Delta reduces the A/D conversion circuitry to its basic essentials-a single-bit converter. The ADC is continuously tracking the signal and updating its value. With this single-bit data stream continuously signaling either greater or lower, the IC adds or subtracts a fixed amount of "charge" from the buffered input signal, operating without the resistor ladder common in DACs. This single-bit output is simultaneously processed by the internal, clock-controlled, six-pole, Gaussian filter that not only reduces the noise in the input signal, but also removes the noise inherent in the single-bit converter.

This filter conditions the input signal by providing a steep roll-off of any frequency components (i.e., noise) above the cutoff frequency of 10 Hz. With a 4-MHz clock, it attenuates 60-Hz line noise to 55 dB. This line noise attenuation increases to 90 dB when a 2-MHz clock is used because the cutoff frequency is halved to 5 Hz. In my experience with the AD7703, the only signal conditioning required is to limit the input frequencies to less than the sampling rate of the filter, which is 1/1024th of the clock frequency.

The only drawback of such extensive filtering is the long settling time of the input filter. If the input exhibits a full-scale step, it will take the converter 125 ms to output a value correct to within 0.0007%. In this instance, the AD7703 functions quite like a tracking ADC, which must ramp up incrementally one bit at a time when tracking the input step. However, even here the AD7703 is better because it oversamples the input at 800 times its clock-controlled conversion rate.

The AD7703 also offers three types of calibration: internal autocalibration, system gain and offset calibration, and update of system offset. With system calibration, the AD7703 can calibrate additional off-chip circuitry, which is alternately switched from minimum input to full-scale input. This design only uses the internal autocalibration routines because of the difficulty in managing the system calibration modes. Software manages system offset and gain.

One last feature of the AD7703 that I have not mentioned is the price-under $30 for single unit quantities. Only a digital IC could offer this price and performance.

DATA THAT'S GOOD AND PLENTY

Before detailing the circuitry and its operation, let me review the design goals for the hardware system. Because the design is destined to work with some pretty extensive signal processing, the best platform to choose should offer a large assortment of good, inexpensive programming tools: PC/AT compatibles. Once the basic routines are up and running, transferring them to another platform is much easier. Also, in the interest of this portability, the C language tops the bill. One good option is the popular DOS-based integrated compiler-debugger from Borland (for me especially, because it's already installed on my software development system).

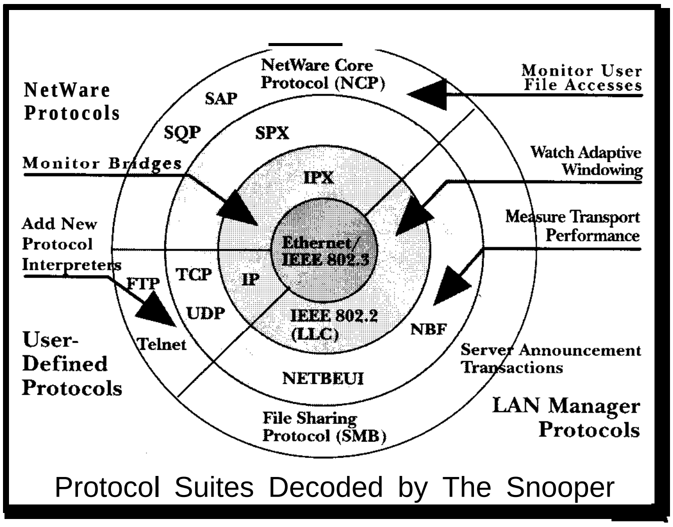
The main hardware design goal is simple. Get the most accurate performance at the least cost from the simplest circuit. This time, the goal is attainable because of the high level of integration of the ADC. Most of the other components match this level of integration. For the intermediate buffer memory sitting between the ADC and the PC bus, a single-chip static FIFO fits the bill. For the PC I/O and the timing chain, a trio of 22V10 electrically erasable GALs work well. The choice of software for developing the GAL PLD logic is also an issue and good packages are available free. The PLDSHELL+ program from Intel is the best I've seen so far.

The remaining design goal is to simplify the software support for the time-dependent operation of the hardware. For this requirement, a register-based I/O design makes the status and data more easily accessible. To speed the principal bottleneck-the data readout-the A0 address line was not decoded. Then the software can use "word" reads from the I/O port. The routines can use the AD7703 for synchronizing the data because it continuously outputs its DRDY signal. Maintaining the real-time connection on a 12-MHz PC/AT is possible because the depth of the FIFO will hold over 170 20-bit readings.

PACKAGING: LOCAL AND REMOTE BOARDS

The analog components are located on a remote circuit board connected to the digital board in the PC/AT by a 10-pin ribbon cable. The TTL-level signals are buffered to provide error-free operation at distances of 25 feet. Using the remote circuit board also increases accuracy because the path for the noise-sensitive analog signals can be really short. Power is supplied through the 10-pin cable as well, so there's no need for remote power supplies.

Another advantage of a remote analog pod is its custom configuration support. Each application, whether for voltage, current, temperature, light intensity or such, can use a custom circuit best for that measurement. For



most custom measurements, the simpler the analog circuitry, the greater the accuracy. When you need a different measurement, you can connect another pod without having to cycle the power on the PC/AT. The circuit board layouts available offer generous prototyping space.

Much of the electronic design has gone into capturing the serial data from the AD7703, and buffering the data so an AT-class machine can perform real-time signal processing. A pair of I/O data ports is also an option and is used for debugging and as I/O to other circuitry. A pair of 22V10GALs does the timing for the digital capture of the serial data. The logic capabilities of the 22V10 can get quite complex, and the development of the hardware logic equations is the key to a successful design. Being electrically erasable, the 22V10 is ideal for new development. Programmability of the 22V10s also ensures that future updates won't require a new board layout, especially because of the close similarity of the 22V10 to the EP610, a UV erasable device with twice the registers of the 22V10. I developed this logic using the Intel PLDSHELL+ logic compiler and simulator.

A 67C4501512-byte FIFO memory forms the elastic memory interface between two asynchronous cycles: the AD7703 serial data cycle and the PC/AT software processing (see Figure 1). The 5 12-byte depth of the FIFO matches the processing and cycle time of my 12-MHz AT by giving about 43 ms of time to read out 170 20-bit words. Although that's not enough time to process all the data numerically, maintaining a queue of eight active readings is possible. For every 20 readings, the first reading is used for averaging, giving a fixed data rate of 200 readings per second. A quick change of the clock to the AD7703 is an easy way to lengthen this time period; it can accept clocks from 200 kHz to 4 MHz.

A 74LS245 bidirectional bus IC and a third 22V10 make up the interface circuitry to the PC bus (see Figure 2). The GAL performs all the timing and address decoding needed by the board. I chose an 8-bit interface for simplicity and to allow a wider use in all DOS ISA systems. With I/O decoding, the ease of use outweighs the speed penalty for an 8-bit bus.

One other design option that deserves mentioning is using an 805 1 instead of the two GALs and a FIFO. The serial port of the 8051 would directly connect to the clock and data lines of the AD7703, with the 8051 controlling the data output rate (a different operating mode of the AD7703 accepts an external data clock]. The 8051 would then need to provide the timebase synchronization for the data. It could also perform some low-level averaging and data processing, which may be a good addition to the HCS II system if it was added to a COMM-Link module.

I adapted this design from some prior development I did for process monitoring and control. The original system needed a 20-bit ADC to measure light intensity extremely accurately using either a silicon

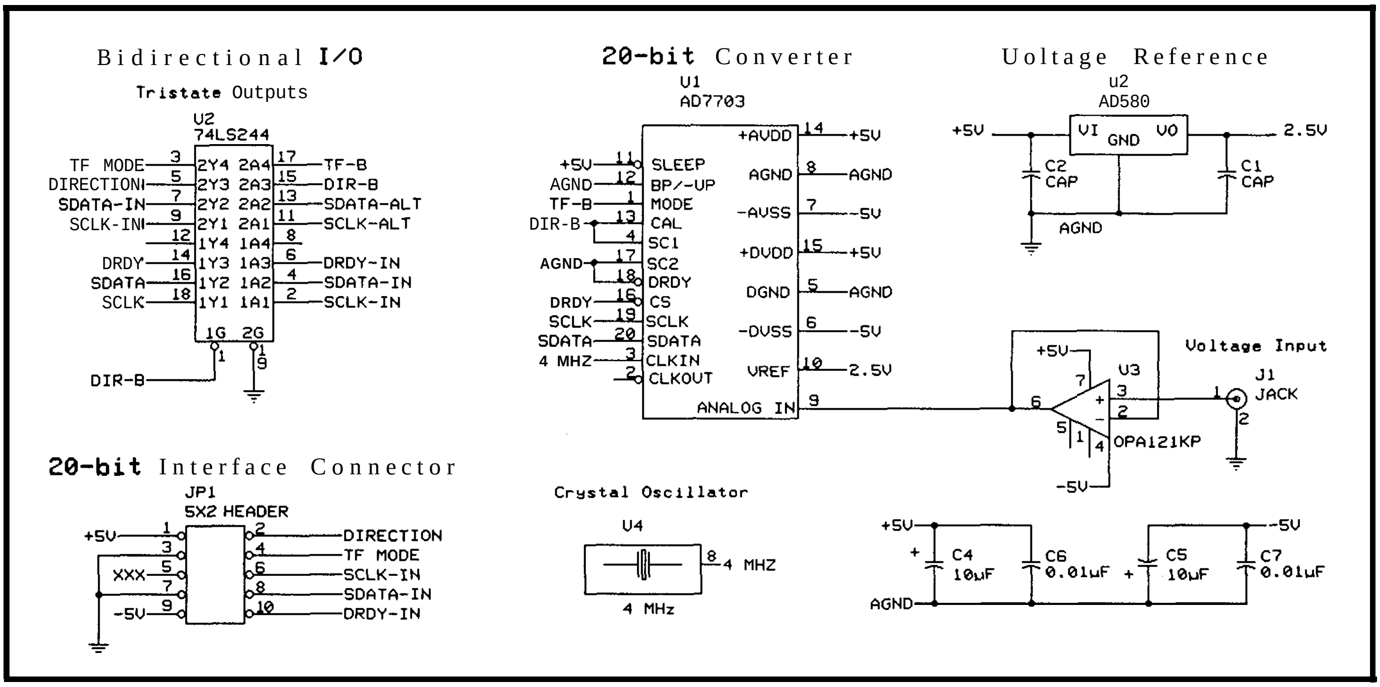


Figure 2—The remote board consists of a buffer for the input signal, the A/D converter, a digital buffer, and a handful of support components

photodiode or a photomultiplier tube. When certain process criteria were met, the software would then issue system commands via a custom interface bus to alter the light signal's rate of change. For this kind of automation to work, the signal had to be essentially noise free down to 18 bits.

TIMING IS EVERYTHING

The pair of 22V10s ties together a number of synchronous events. The first 22V10, U8, is the timing generator while the second, U9, is the write generator. The AD7703 signals the system it is ready to transfer data by pulling the DRDY line low. After 72 clock cycles, the AD7703 outputs data at one-quarter the clock frequency, typically at 1 MHz. The write generator counts each bit and issues a write after each set of 8 bits. A 74LS164 performs the conversion of serial data to parallel data prior to the FIFO. However, the DRDY line goes high after the transfer of 20 bits, signaling the end of data. The timing generator then fills in the last four bits, to round out to three bytes. At the end of the third byte, the write generator pulls the STOP line high, completing the transfer, and holding the clocks until the next DRDY cycle.

The current state of the data transfer is available through the status register. The status of DRDY and the FIFO flags EMPTY and FULL are used to maintain the FIFO buffer. If the FULL flag goes low, then the FIFO has missed some data. You can maintain a count of the reading input by monitoring EMPTY. For the best synchronization, the software should wait for the DRDY low-to-high transition before it accesses the data. Also, it should continue reading bytes until the EMPTY line goes low, which indicates that it has read all the data in the FIFO.

Now that the timing signals are known, only compiling the logic equations and programming the 22V10s remains undone. However, there's a catch. Because the operation of the two GALs is closely linked, is simulating their operation together possible? The Intel PLDSHELL+ does support this multichip option through its Intel_Arch virtual logic device (see the file 20ISA_T.PDS).

The use of GALs simplifies things even more. You don't have to wait for erasure because it's done electrically. Between the speed of logic compilation and the quick reprogramming, logic development moves quickly.

The TEST20.C program shows how to detect the 201SA hardware, maintain the FIFO buffer, and display the output of the board. It also includes support for an averaging queue or variable length.

ANALOG TECHNIQUES FOR 20-BIT RESOLUTION

I hope the internal operations of the hardware are clear, because it's time to get down to the basics: the techniques required to make those extremely accurate analog measurements of which I've been speaking. To start, forget everything you know about working with digital, where signals change on the order of nanoseconds. This board deals with measurements below 10 Hz, a rate human beings can see and feel.

Although most articles I read mention the benefits of using instrumentation amplifiers, the only types that work reliably at this resolution are the monolithic ones, such as the Burr Brown INA105. Discrete instrumentation amps will be sure to drift and will show you the difference in temperature between the two gain resistors on either side of the op-amps. Long-term drift is the real enemy, so let the price of the INA105 be the guide and use them only where really needed.

Because I've started making rules, I might as well begin with the big one. Use one single point where all ground connections come together. Sometimes your circuits will look strange with a big solder plane to which almost everything connects, but use it because it will save you hours of debugging.

I think the second rule for accuracy is to make the best connections possible. Solder or wire wrap everything. Crimp-type solderless connections are thermocouples just waiting to inform your data of their differences in temperature.

Third rule, use the lowest possible resistor values. Thermal noise increases with the square root of the resistance: a 100k resistor has ten times the thermal noise as a 1k. Large resistors also exhibit more long-term temperature drift. Depending on the output impedance and current rating of an op-amp, a low-value resistor works just fine, especially for the noninverting configuration. If you do need to use a large resistor in the feedback path of an op-amp, put a monolithic ceramic capacitor in parallel with it. This addition will serve to shunt the resistor noise and to reduce the bandwidth of the amplifier; a good idea for low-frequency measurements. However, don't forget to calculate the Gain-Bandwidth product if the amplifier is set for high gain.

Finally, the last rule is to keep the signal levels small. Surprised? If you think a small signal will be swamped in noise, then you're working in the wrong time domain. Down in the low-frequency world, noise like that exists only on your oscilloscope and is a remnant of the fast digital world. Small signals offer much better linearity. A precision op-amp, such as the Burr Brown OPA 12 1, will offer the best performance because of its high internal open-loop gain and low offset voltage.

A VIEW OF THE WORLD AT MICROVOLT LEVELS

Even though electronics is firmly entrenched in the world of science, custom measurement electronics often exhibit some baffling behavior. With the circuit I've presented, measuring curious phenomena such as the unexpected optical sensitivity of injection molded op-amps, the internal spring constant of resistor substrates, and the varying static charge on your hand is possible. Twenty-bit resolution is similar to a microscope, where the nuances of the ordinary are expanded to the point where more subtle responses become clearly visible. Second- and third-order effects begin to take precedence. Trying this circuit out will clearly show how electronics still depends on physics. ❑

Russ Lindgren received a B.S. in Engineering Science from Tufts University with a major in muth und physics. He has been designing, building, and mostly debugging measurement electronics for the past twelve years.

SOFTWARE

Software and GAL equations for this article are available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

SOURCES

A parts kit of this design is available from:

Program Systems Inc.
P.O. Box #99
Colrain, MA 01340
CompuServe 7155 1,2062

Cost is $125 for parts only, or $150 for parts and circuit board

Intel's PLDSHELL+ is available by calling (800)323-EPLD. A bulletin board is also available at (916) 9852308.

The specifications for the AD7703 are available from Analog Devices by calling (617) 329-4700.

I R S

413 Very Useful
414 Moderately Useful
415 Not Useful

Build a Computer-Controlled Multiswitch System

The most complicated part of some control systems is the interface to the outside world. A chip from Power Integrations Inc. makes building that interface just a little bit easier.

FEATURE ARTICLE

Michael Swartzendruber

Using computers or microcontrollers to control power to multiple devices or appliances certainly isn't a new idea. Nonetheless, that doesn't stop chip companies from dreaming up new ways to do so.

The PowerPort described in this article is one of the newest ways to skin the cat. It offers some very interesting features compared with other systems. For one, the output switches are MOSFET devices capable of switching high-power loads up to 400 volts at 250mA. What's more, eight MOSFETs are packed into a single DIP, making a 16-switch port a practical, low-cost reality by implementing two DIPs along with some support circuitry. If higher currents are needed, then the MOSFETs can be easily paralleled to provide current load sharing.

Another very nice feature of the MOSFET array is it sports four current feedback pins. In the chip, these pins are connected to a resistor that senses load current. These pins can be "read" by a control system to provide information about the current levels through the monitored MOSFETs.

The PowerPort project makes use of the current feedback outputs of the array by providing a digitized value of the current to the control system. This aspect makes the PowerPort a very likely candidate for a full-featured feedback-loop control system.

CIRCUIT DESCRIPTION

The PowerPort system is composed of three major subsystems: the output section, the current monitoring feedback, and the power supply.

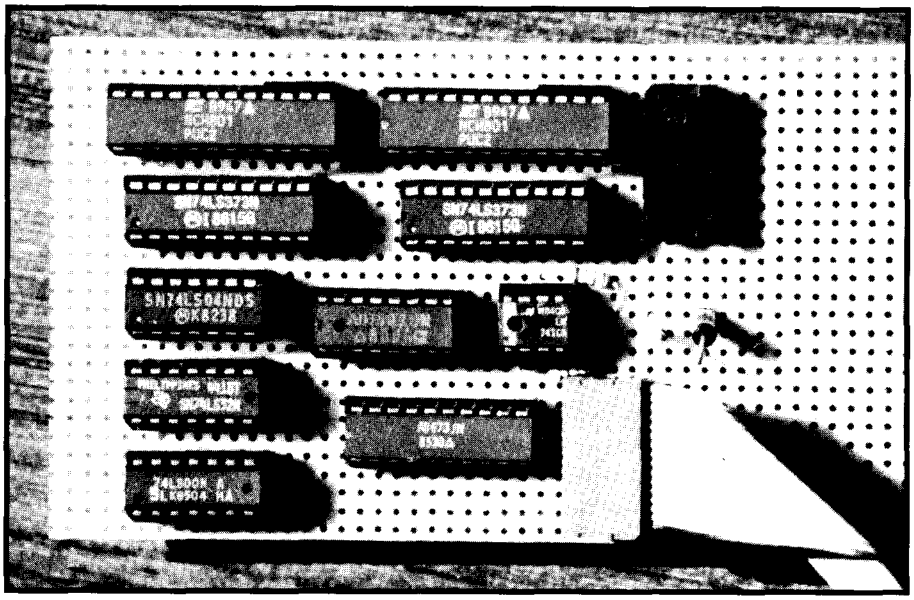
The circuit requires the following signals from the control system: ● RD and 'WR, address lines AO-A4, data lines DBO-DB7 (see Figure 1).

The PowerPort provides a *WAIT signal that can be used to hold the control system read cycle during the conversion time of the port's ADC.

The PowerPort's optional power supply is designed with features that can reduce the system's power usage.

OUTPUT SECTION OVERVIEW

The output section of the PowerPort is straightforward. The MOSFET arrays are driven by TTL-level signals. These control signals are presented to the gate of the MOSFET, which



Channel	Port (hex)	Port (decimal)
1	00000	0
2	00100	4
3	01000	8
4	01100	12
5	10000	16
6	10100	20
7	11000	24
8	11100	28

Table 1—*The upper three bits of the port* address determine which *channel is being accessed. The lower two* bits select *either the ADC or one (or* both) *of the 8-*channel interface *chiDs.*

switches on the drain-to-source junction of the MOSFET when it is high. A low signal turns the device off.

The PowerPort output section uses 74LS373 latches that serve two purposes. First is to latch the data from the control system. The second purpose is to allow the two sets of eight output switches to be addressed independently because of the way they are connected to the PowerPort's address decoder. This feature allows a 16-line output system to be controlled by an S-bit data bus system by providing the equivalent of two 8-bit ports.

The address decoder for the output port selects one of the two latches. Then, the selected latch passes and holds the data from the data bus to the appropriate octal MOSFET array. The address decoder for the output section uses AO, Al, and *WR signals. The output ports are decoded as ports land 2. Data is transferred to the MOSFET array output ports by writing to ports 1 or 2.

By writing to port 3, both ports can be written to at the same time. This feature effectively operates the two ports "in parallel," meaning they both will be in the same state, and can be useful if the MOSFET arrays in the two separate DIP packages are in parallel and are sharing the load.

THE CURRENT-LEVEL FEEDBACK SYSTEM OVERVIEW

The current-level feedback system is divided into three functional groups: an op-amp-based current-to-voltage amplifier, an analog multiplexer, and an A/D converter. This circuit is also controlled by the address decoder circuit. The analog circuitry and the multiplexing section requires ± 15 VDC power supply voltages.

Current-monitoring feedback signals originate in the MOSFET array device. The array has current output pins that mirror the current level carried in a particular MOSFET device.

All eight current signals (four from each array) from the MOSFET arrays are presented to the inputs of an analog multiplexer. This device is used as an "address decoder" for the analog channels and makes it possible to specify which one of the eight MOSFET channels to monitor. Using this chip in the circuit saves you the cost of having to provide eight separate current amplifiers and ADC circuits with their associated support circuitry.

The selected analog channel is passed through the mux and amplified by an operational amplifier. The op-amp circuit is designed to give 5-volt output for 250-mA input. The analog output of the op-amp is presented to an ADC compatible with an 8-bit microprocessor, which presents the

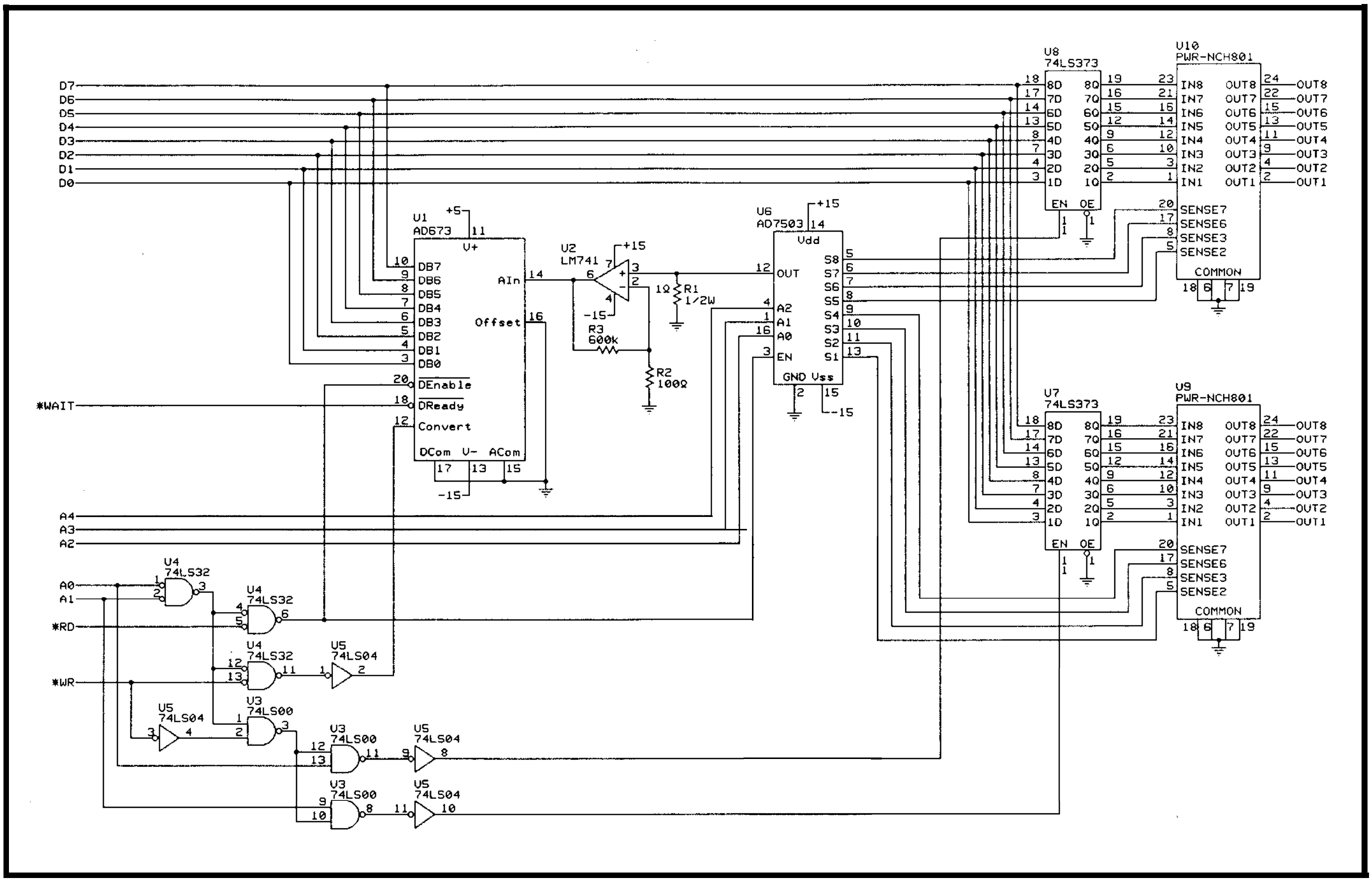


Figure 1—*The P WR-NCH801* provides *sense lines for ha/f of its drivers that* allow *the computer to defermine the load being p/aced on those drivers. An analog* multiplexer *selects one of the sense lines and an ADC converts it to something the* computer *can use.*

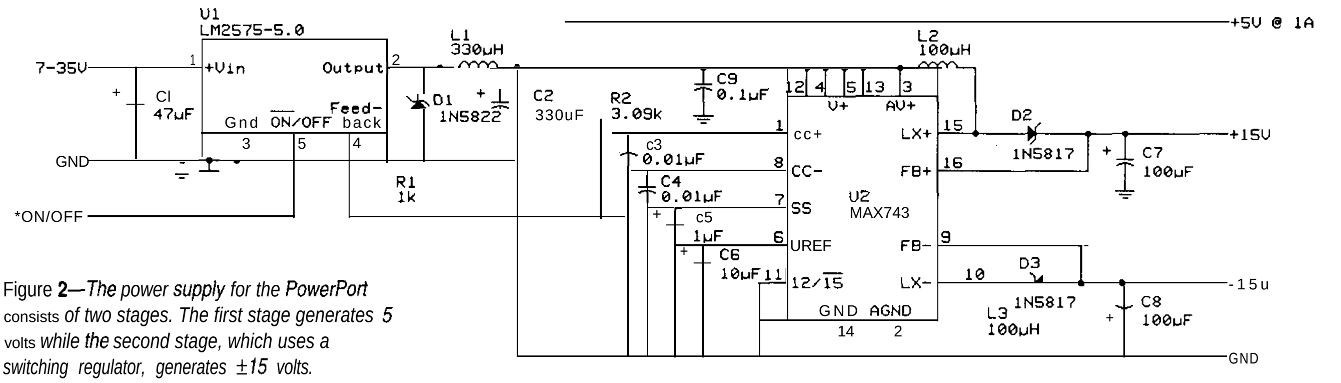


Figure 2—The power supply for the PowerPort consists of two stages. The first stage generates 5 volts while the second stage, which uses a switching regulator, generates ±15 volts.

data to the controlling system as a byte-wide digital word.

This analog feedback system provides an 8-bit parallel data interface. The data is read from the feedback system as follows:

1. Write to port 00, which resets the ADC process. This reset process can take up to 30µs. If the control system is very fast, it may have to have a delay loop before it can read data from the converter.

2. Read an analog channel (see Table 1).

Notice each of these addresses has the value xxx00. The zeros in the low two bits plus *RD are used to decode the data enable pins on both the mux and the ADC. Address lines A2–A4 determine which analog channel on the multiplexer will be selected.

THE SYSTEM POWER SUPPLY

The analog circuitry in the feedback section requires supply voltages in addition to the standard +5 VDC required by most digital systems; thus, I include the optional switching power supply subsection (see Figure 2). This optional switching power supply is a double DC-to-DC converter system. The supply is used to provide the +5 VDC for the PowerPort system logic and the ±15 VDC to the analog mux and A/D conversion sections of the PowerPort system.

The power supply also offers a power-saving, logic-driven, power-down function that can be used to cut power to the entire PowerPort if desired. This feature can be used as a "board select" to apply power to the entire circuit only as needed or as desired.

The first section of the switching supply is used to provide a regulated 5-VDC supply for the system logic and also supplies power to the next stage. The 5-VDC supply has a TTL-compatible control line that can be driven high to power down the whole port.

The second stage of the power supply is an additional switching supply that generates ±15 VDC from the 5-VDC supply. These voltages are used to power the analog sections of the circuit.

While this supply is ideal for this project because of the capability to use the logic signal to power down; any other supply that provides +5 VDC and ±15 VDC will do.

Another nice feature of this supply is it uses two supply modules by the makers of the chips used in each of the switching power supplies. The second stage of the supply is the Maxim experimenter's evaluation kit for their MAX743. The first stage is the evaluation kit from National Semiconductor for their LM2575 step-down switching voltage regulator.

CONCLUSION

The PowerPort is another way to have a computer switch power to the world of objects with which computers coexist. The unique feature of this system is its simple and inexpensive feedback circuit, which allows it to monitor the events it is controlling. This aspect closes the loop, giving the PowerPort a leg up on other, simpler, power-switching devices.

Applications for the current monitoring loop include sensing current through a variety of switched sensor arrays, switching and monitoring power to a variety of other devices, and matrix switching arrangements with current feedback as with polled alarm loops. PowerPort's uses are limited only by your imagination. ▣

Michael Swartzendruber is currently employed at System Integrators Inc., where he works with LAN design and Macintosh programming.

SOURCES

Power Integrations, Inc.
411 Clyde Ave.
Mountain View, CA 94043
(415) 960-3572

Analog Devices
1 Technology Way
P.O. Box 9106
Norwood, MA 02062-9106
(617) 329-4700

Maxim Integrated Products, Inc.
120 San Gabriel Dr.
Sunnyvale, CA 94086
(408) 737-7600

National Semiconductor Corp.
2900 Semiconductor Dr.
P.O. Box 58090
Santa Clara, CA 95052-8090
(408) 721-5000

IRS

416 Very Useful
417 Moderately Useful
418 Not Useful

DEPARTMENTS

Physical Constants & A Mini Interpreter

Open up the junk box, piece together what would normally be an ordinary Boy Scout demonstration, add Ed's magical touch, and you end up with a contraption that teaches more than one concept to more than one group.

FIRMWARE FURNACE

Ed Nisley

Most of the projects you've seen here have been, well, grimly utilitarian. Apart from a glowing LED or a few LCD speckles, they look much the same with the power on or off. Enough is enough!

This issue's project started off as a display for the Electricity and Electronics merit badge booth at the regional Boy Scout Summer Camporee. A local Scout leader, Chris White, asked me to come up with something interesting for the lads.

I decided that the traditional topics weren't nearly interesting enough. After all, bells, buzzers, bulbs, and batteries are pretty much passe today; things have changed a lot since the days of Edison. I'm not sure I have any cotton-wrapped bell wire in my heap any more! What I needed was something attractive, educational, and easy to pull off.

Preferably, it should look a lot different with the power on.. .

Photo 1 shows what I came up with. The idea goes back to Galileo, who was the first to realize that gravity was "diluted" on an inclined plane. He rolled balls down a variety of slopes, timed them using his pulse or a pendulum, and arrived at conclusions that shook the universe. I adapted his inclined plane to the material in my shop, added a microcontroller to handle timing and arithmetic, put the results on a fluorescent display, sprinkled some twinkling LEDs along the rail, and had a good time doing it.

What GALILEO does is measure the amount of time a ball takes to roll a known distance, then it displays

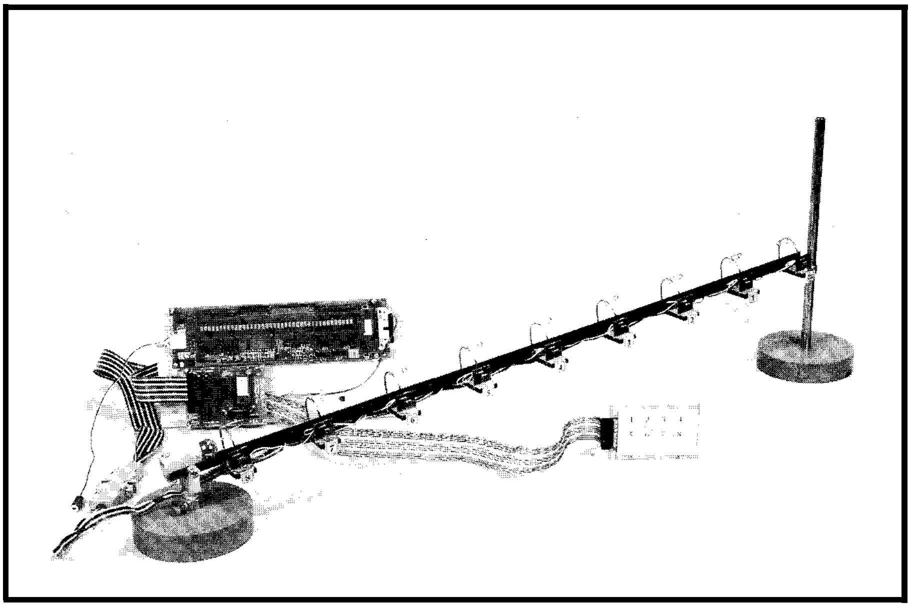


Photo I--The *GALILEO teaches the basics of the acceleration of gravity while at the same time if* illustrates *the* basics *of a data collection system.*

either the time or the average speed. You can study how the ball speeds up as it accelerates down the slope, investigate the effect of slope on the speed, plot acceleration from the speed changes, and so forth. All in all, it beats a stopwatch hands down.

As you might expect, GALILEO was well over the heads of most viewers, few of whom were *Circuit Cellar INK* readers. I'd like to think it inspired one or two Scouts to learn a bit more about physics, mathematics, and computer innards, but that was hard to tell. Everybody thought it was a neat gadget, though, and you might want to put one together for the therapeutic value of building something that is more a work of art (well, a little] than a project.

In any event, this project serves as an example of the mini interpreter technique I mentioned a few columns ago. You'll also see how to time-share I/O functions and reduce the number of I/O pins required, which is a topic near and dear to many designers with a limited hardware budget.

MECHANICAL DETAILS

As you can infer from Photo 1, GALILEO presumes you release a ball at the top of the ramp. The ball rolls downhill (honest!) and passes through nine electrical contact sensors. The microcontroller records the time of each sensor's activation and, when the ball passes the final sensor, the program computes and displays either time or velocity. A keypad selects either overall results or the values at a particular sensor.

GALILEO has all the elements of a "real" data collection system: sensor inputs, precise timing requirements, a display, and a keypad. Admittedly, it does look a little strange, but I tried for attention-getting rather than subtle effects; hence, all the contrasting colors and bright metal.

The rail is a 3-foot steel U-shaped bookshelf rod painted red. The bottom support is a grossly overengineered chunk of aluminum and steel that I turned into a milling machine project all by itself. You can substitute something much simpler with no loss of function. The top support is a steel pipe with a springlike length of stainless steel wire wound around it. A pair of 1.25" x 5" steel disks ballast the whole affair; that their half-inch axial holes matched up with an available bolt and pipe had a lot to do with their selection.

The bottom bumper is a 2" length of cardboard core from a roll of fax paper, cut with a slight downhill slant on the upper end. The ball diameter is 0.75″, which must match the bumper to keep the balls from jumping off the rail. They hit with an enthusiastic thump, so you must provide a resilient bumper, but be forewarned that a piece of rubber won't suffice.

Photo 2 is a close-up view of the sensor at the top end of the rail. The baseplate, a Lexan strip screwed to the rail, insulates and supports an arch made of 12-gauge copper wire. The contact is a length of stainless steel electric fence wire soldered to the arch, although I suspect a big safety pin would have worked just as well. The ball shorts the spring contact to the rail, completing a circuit that tells the firmware when the ball passes through the sensor.

The top sensor's spring points down the rail to allow enough room

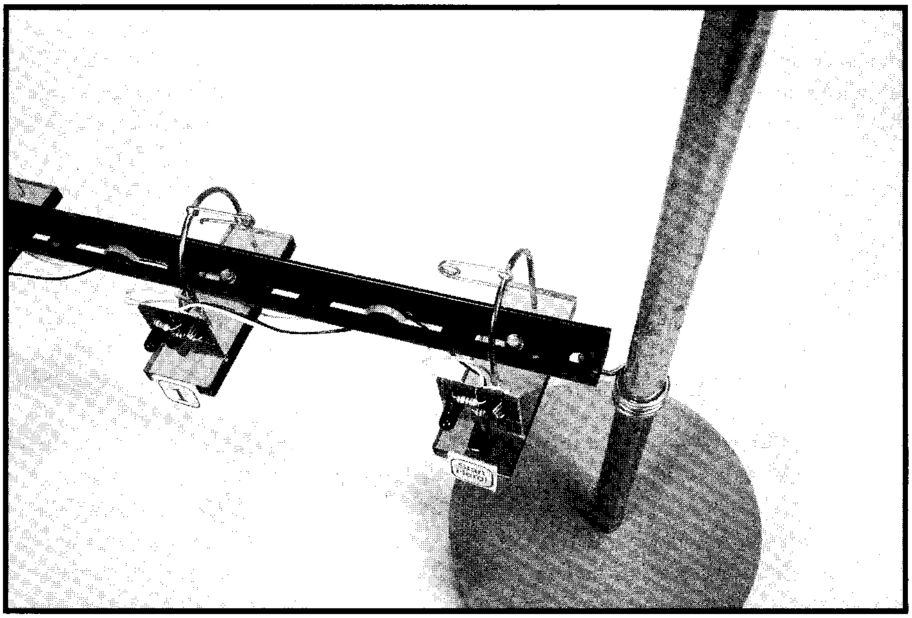


Photo 2—*The ball starts at the fop of the rail and passes several sensors on ifs way down.*

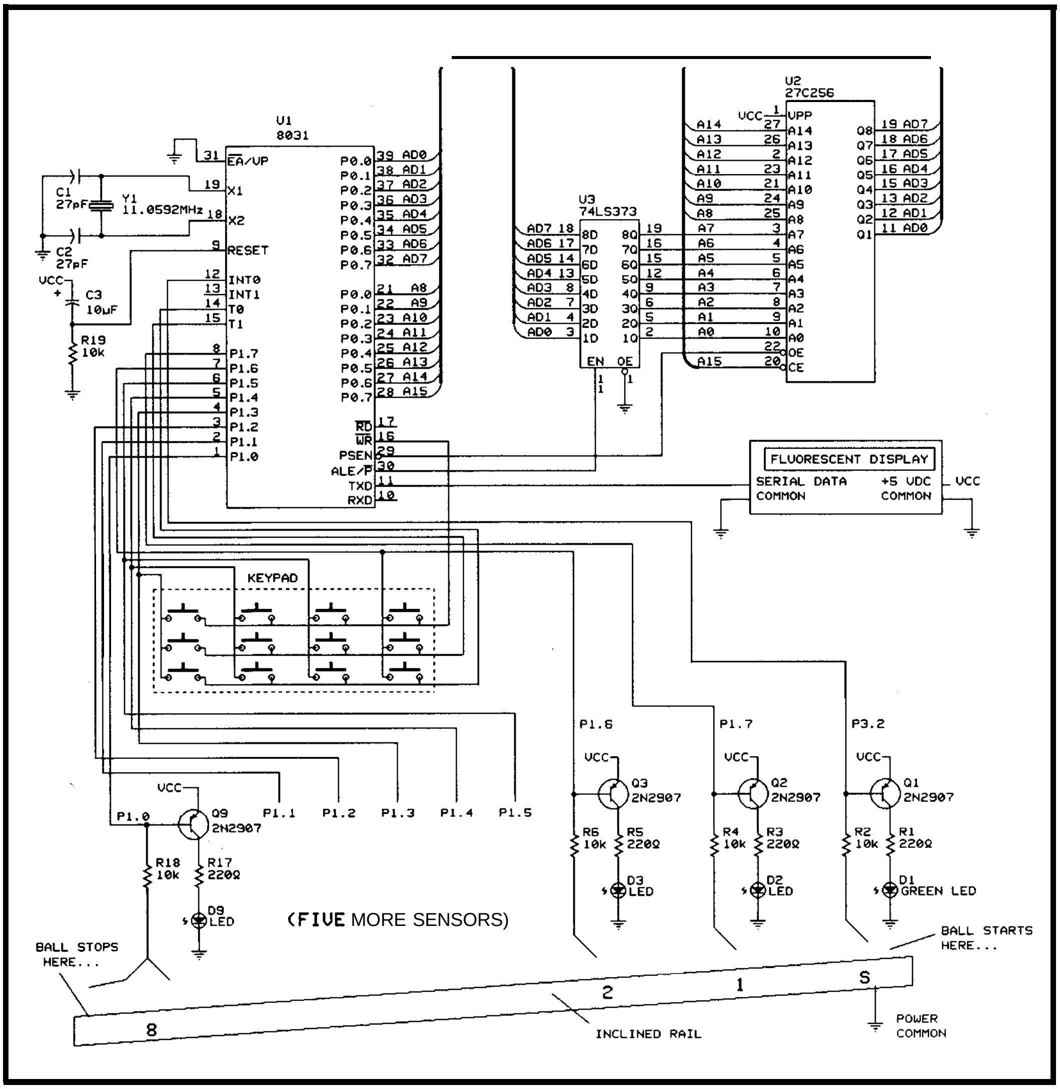


Figure l-Using *mostly* off-the-shelf *computer* hardware, it's mechanical ingenuity and *programming tricks that breathe life into GALILEO.*

for your fingers and the ball between the sensor and the pipe. The sensors are 4" apart, which was easy to measure because the bookshelf bracket has slots on 1" centers. The bottom sensor (see Photo 2) has an additional contact against the ball as it rests against the bumper. This device tells the micro when you've picked up the ball for another run.

Figure 1 shows the circuitry involved in this project. Each sensor sports a small circuit board with an LED and driver transistor. A Cottage Resources Control-R board holds the 803 1 microcontroller and its support circuitry. An IEE FLIP vacuum fluorescent display and a matrix keypad serve for the human interface. A wall-wart transformer supplies regulated +5 volts through a 2.5mm phone jack wired to the Control-R board.

HARDWARE LAYOUT

Perhaps the most interesting hardware feature is the use of just I3 I/O bits to scan a 12-switch keypad, drive the serial display, monitor nine rail sensors, and flash nine LEDs. The trick is that most of the I/O doesn't happen at one time, so several different functions time-share the I/O pins.

For example, each of the sensors includes an LED that can be turned on in either of two ways: when the ball passes under the sensor or when the 8031 drives the I/O pin low. When the firmware knows the ball is at the bottom contact (because that sensor input bit is held low) it can twiddle the remaining eight LEDs by driving the pins as outputs. When it's waiting for the ball to hit the sensors, it sets the pins high so it can watch for the ball to short each one to the rail.

Four of the sensor lines also drive the keypad columns. Normally all twelve keypad switches are open, so they have no effect on either the LED outputs or the sensor inputs. Each time the firmware changes the LEDs, it shuts them all off while it drives one of the three keyboard row outputs low, checks to see if any of the four keyboard inputs remain low, then loads the new output value to light the LEDs.

You can see this switching if you move your eyes very quickly across the rail; the lighted LEDs form rows of dots. This effect is similar to the one you'd get if you swept your eyes across an LED clock; the digits break up because the segments appear out of order. As long as the refresh rate is reasonably high, this switching is not a problem. I picked a 30-ms rate, which is just barely fast enough.

Although it's not shown in the schematic, the Control-R board also includes a MAX232 serial port driver. The FLIP display has a l-bit TTL serial input, so I hard-wired it to the 803 1 serial output pin. The firmware includes a simple debugging kernel that accepts single-letter commands from the serial port and sends the results to both the display and the RS-232 port.

Unlike all the other Micro-C code I've presented in this column, these routines use the polled serial I/O code included with the Micro-C compiler. Only one character comes in at a time and an output interrupt handler isn't needed, so polled I/O works just fine. Even though the serial output makes the LEDs flicker slightly, the output turns out to be little enough that it doesn't cause a problem.

The FLIP display runs at 1200 bits per second and requires 8 data bits with the MSB always 0, no parity, and 2 stop bits. Fortunately, the 8031 can produce that format in 9-bit mode with TB8 set to one. The only restriction is that you must not send a character with a nonzero MSB, which is no problem in this application.

Incidentally, the IEE display doc showed a jumper to select either RS-232 or TTL voltage levels. I spent quite a bit of time fiddling around until I discovered that all the RS-232 level-shifting components were omitted from my display. Of course, the display was an electronic surplus part to begin with, so I should have known better. Moral of the story: check to make sure the hardware is there before you try to use it!

Now, time to use the remaining pair of I/O pins left on the 803 1.

Listing 1-"Source *code" for the* LED program. *The mini interpreter reads* this *code and translates if info* LED *outputs. The program is* simply *a C array* filled with data that the *interpreter recognizes and knows how to handle.*

```
WORD LED_Attract[] = {

   0x0000,

   LED_LABEL(1),

   LED_RATE(1),
   0x0001,0x0002,0x0004,0x0008,0x0010,0x0020,0x0040,
   0x0080,0x0100,0x0100,0x0080,0x0040,0x0020,0x0010,
   0x0008,0x0004,0x0002,0x0001,

   LED_RATE(2),
   0x0002,0x0004,0x0008,0x0010,0x0020,0x0040,0x0080,
   0x0100,0x0100,0x0080,0x0040,0x0020,0x0010,0x0008,
   0x0004,0x0002,0x0001,

   <<< many lines omitted >>>


   LED_RATE(19),0x0002,
   LED_RATE(18),0x0001,
   LED_RATE(26),0x0001,

   LED_JUMP(1),

   LED-END
}
```

ILLUMINATED INTERPRETER

The 8031 in this project spends most of its time doing nothing, as do most computers everywhere. Rather

Listing 2—*LED* control *mini interpreter. This switch statement is the heart of the* mini interpreter. *It processes the data shown in Listing 1 and updates the LED bit pattern sfored in the Pattern C* variable.

```
switch (0xf000 & *(WORD *)pSlot){      /* next pattern opcode */

case LEDOP_JUMP:                      /* branch to labeled target */
   Target = LEDOP_LABEL | (0x0FFF & *(WORD *)pSlot);
   pSlot = pPatBase;                     /* restart from the top */
   while ((Target != *(WORD *)pSlot) &&
          (*(WORD  *)pSlot) != LEDOP_END){
      pSlot++;
   }
   break;

case LEDOP_LABEL:            /* define label location */
   pSlot++;                         /* which we skip... */
   break;

case LEDOP_RATE:                      /* set update rate */
   TickLimit = 0x00ff & *(WORD *)pSlot;
   pSlot++;
   break;

case LEDOP_END:               /* end of display list */
   break:

default :
   if (((WORD)0x8000) > *(WORD  *)pSlot){
      Pattern = *(WORD *)pSlot;          /* set display bits */
      RailCtr = 0;                     /* reset update timer */
      pSlot++;                     /* and step to next slot */
   }
   else {
      putstr("\nBogus opcode:"):          /* invalid opcodes */
      puthex(*(WORD *)pSlot);
      putch(' ');
   }
   break;
}
```

than look unoccupied, the firmware spends its idle moments blinking the rail LEDs to attract attention from passersby. Given the amount of time I spent answering questions, it did that rather well!

Port 1 drives the eight red LEDs and Port 3 Bit 0 drives the green LED at the top of the rail. As you can see in Figure 1, a 0 output turns the corresponding LED on, while a 1 is required to see whether a ball is present at the sensor position.

As I mentioned earlier, the rail's contact sensors and the keypad matrix also use the same port bits as the LEDs. Therefore, I had to write the code to handle all three functions at once, because the keypad must be active while the LEDs are flashing and the sensors report that the ball is still at the bottom of the rail.

For once, though, the display and I/O functions do not require any assembler code. The main loop calls a scan-and-display routine after each timer interrupt, which occurs every 30 ms. That routine checks the keypad and serial input, checks for ball motion, and refreshes the LED bit pattern. If the keypad or serial port produced a command, the mainline code calls a decoder routine to produce whatever output is called for on the serial output to the vacuum fluorescent display.

An Internal RAM variable (the Control-R board has no External RAM) holds the current LED pattern; I used a WORD (unsigned int) variable because there are 9 bits. Because the LEDs do not change every 30 ms, the scanning routine simply rewrites the output from the variable. Producing a new output pattern is a simple matter of updating the variable using an ordinary C assignment statement, and the LEDs will reflect the value in under 30 ms.

Most microcontroller applications I've written have utterly functional outputs; an LED is ON when a condition is present, or OFF to show the converse. In this case, I had to come up with a way to flicker the LEDs in an interesting way that had nothing at all to do with anything else-quite a challenge!

My first thought was simply to shift a few bits around in the variable. Although easy to do, the repetitive patterns were also staggeringly dull. Scratch one idea.

Next I tried an array of bit patterns with some code to copy them into the magic variable every few timer ticks. While this idea allowed some versatility, the array was far too large to be practical and nearly impossible to edit. Scratch another.

Finally, I threw in the towel and did what I should have done in the first place; I wrote a small interpreter implementing a tiny language that knows how to control LED bits. The language has only five instructions: update the LEDs, set the time between updates, jump to a label, and mark the end of the program.

Lest this sound too terrifyingly complex for words, Listing 1 shows the first few lines of the program. The resemblance to machine code is intentional; the program is simply a C array filled with 16-bit values that just happen to be language opcodes.

Listing 2 shows the interpreter's core: a C switch statement called whenever the display update counter times out. The LED-RATE opcode loads this counter, which ticks once every 30 ms. The code produces the next bit pattern based on the current opcode in the LED control program.

I can hear it now: "You mean that's it? One lousy switch can't be a language interpreter!"

Ah, but in fact, that is indeed it. Look at what's going on: think of pSlot as the program counter and the high-order nybble of each word as the opcode. Each opcode corresponds to one of the case clauses. The code after each case changes pSlot, sets the display update counter, or sets a new LED bit pattern as dictated by the opcode-what more could you ask?

No, it’s not a general-purpose language; there are no conditional branches, no arithmetic or Boolean operations, no trig functions, no fancy memory addressing modes (shucks, there aren’t even any variables!). You get to write the code by hand because the language syntax is entirely rudimentary. However, to invoke Einstein on my behalf: “Things should be made as simple as possible, but no simpler.”

When your application needs more features, just add them. Left and right shifts? Define another pair of opcodes and a few lines of code. Variables? An opcode or two, plus static variahles to hold the data. Keep it simple, savants!

A few issues ago, I mentioned a CD Jukebox that used a mini interpreter. I can’t include more of the source, because I completed this project for a client who doesn’t want publicity, but I can include one of the interpreter programs as an example of the power you can put into a very simple language.

I wrote the CD Jukehox code using Avocet C and assembler, but I found the prospect of writing code to control each and every CD player unappealing. I wrote the main program to use data stored in EPROM whenever it needed information about the player, such as the maximum number of CDs, the track layout, and so forth. When the Jukebox needed to play a track, it applied the interpreter to a block of instructions written in my language.

This aspect meant adding a new CD changer had three steps: capture the remote control’s IR signals using an IR Master Controller (see “Build a Trainable Infrared Master Controller” in volume 7 of *Ciarcia’s Circuit Cellar),* create a parameter block defining the changer’s capacity, and write a script to select and play a track. The actual Jukebox code remained unchanged, so you could substitute changers by simply burning a new EPROM. In fact, all the data blocks are at fixed addresses, so you don’t even need an assembler!

Listing 3 shows the control routine for a Pioneer Laser Karaoke player. Even though it isn’t a CD changer, I could use the mini interpreter to handle it without altering the main code. Notice the conditional branches, counted loops, and, yes, even a few variables, all of which are decoded by a switch statement similar to the one shown in Listing 2.

Listing 3—*CD Jukebox interpreter program. The CD Jukebox code included a mini interpreter fhaf knew how to control CD players. This* program drives a Pioneer Laser Karaoke player using *IR* commands stored in a *Master Controller* file.

```
 -- CLD-V500  script   Pioneer Laser Karaoke player

 --- Master Controller key indexes

CLDV500_TRACK     EQU   0          ; variable starts at track 1!
CLDV500_T10       EQU   10         ; add +10 to track number
CLDV500_PGM       EQU   11         ; enter programming mode
CLDV500_STOP      EQU   12         ; stop OR EJECT disk
CLDV500 PAUSE     EQU   13
CLDV500_CLEAR     EQU   14
CLDV500_PLAY      EQU   15

      Reset player to known state
      This verifies that we're not stopped to prevent an eject...

      DB OP_LABEL,'R'
      DB OP_JSTOP,'s'              ; if stopped, don't send a stop!
      DB OP_SEND+VAR_ZERO,CLDV500_STOP
      DB OP_LABEL,'s'
      DB OP_SEND+VAR_ZERO,CLDV500_CLEAR ; clear program
      DB OP JUMP,'z'

      Select track and start playing

      DB OP LABEL,'P'
      DB OP_SEND+VAR_ZERO,CLDV500_CLEAR; clear program
      DB OP_SEND+VAR_ZERO,CLDV500_PGM    ; enter program mode
      DB OP_DECIMAL+VAR_TRACK,0          ; A=10s, B=1s
      DB OP_JZVA,'u'                     ; skip if track < 10
      DB OP_REPEAT+VAR_A,IMM_MINUS1      ; repeat +10 n-1 times
      DB OP_SEND+VAR_ZERO,CLDV500_T10    ; send +10 key
      DB OP_LABEL,'u'
      DB OP_SEND+VAR_B,CLDV500_TRACK     ; send units key
      DB OP_SEND+VAR_ZERO,CLDV500_PLAY   ; start it playing
      DB OP_JUMP,'z'

      exit

      DB OP_LABEL,'z'
      DB OP_END,0
```

Once you get used to the idea that you can write your own language without a lot of trouble, you’ll find the trick coming in handy all nver the place. My only regret is that I wasted so much time fiddling with simple shifts and bit patterns before writing GALILEO’s LED control interpreter!

LONG MATH

Although 16-bit integers are convenient for many purposes, I think you’d agree that 32 bits would be a lot hetter. Some 8051C compilers support long 32-bit integers, but, alas, Micro-C does not. Dave Dunfield points out that, in his experience, you can almost always squeak by with delicate scaling tricks; although longs are on his wish list, they’re not in the immediate future. GALILEO really needed more than 16 bits for some of the calculations, so I had to buckle down and write some code that I’d been avoiding.

To make a long story short, the MATH.* files include some handy arithmetic routines: 32-bit addition and subtraction, signed and unsigned 16 x 16 multiplies that produce a 32-bit result, and an unsigned 32-bit division. They all use a 4-byte Internal RAM buffer, so they should work with any memory model.

Credit where credit is due: the division routine is iust an extended version of the one Dunfield supplies in the Micro-C run-time library start-up code. The bit shifting is not at all obvious and it took me quite a while to convince myself that it really did work the way it had to. Very slick code, indeed!

These routines are not optimized for speed, but they should suffice when you need an occasional value with more bits than usual.

ACCURACYINMEASUREMENT

Obviously, GALILEO is not a laboratory-grade analytical tool, but that was not my intent. Allow me to convert those defects into topics for further investigation rather than endure a torrent of letters and Email.

Apart from the air resistance that everyone agreed to ignore in Physics 101, there is obviously friction against the rail and the sensor contact points. The rail may be warped vertically or horizontally, so the ball's net acceleration is not directly along the rail axis. The contact points are not precisely spaced, so the timings are slightly off. Also, of course, there should be some compensation for the ball's rotational inertia, which soaks up potential energy that would otherwise send it down the rail a little faster.

These effects are most apparent when you reduce the rail's slope. The ball moves irregularly, sticks underneath the sensors, and generally behaves in a nonideal fashion. Increasing the slope reduces the effect of some errors, but they're still in there corrupting the measurements.

Most of those effects can be estimated, calculated, or measured. You may have to add more sensors, change the overall design, tweak the rail, and so forth-and I suspect you'll have a lot of fun in the process!

RELEASE NOTES

The BBS files include the source code and EPROM hex files you need to build the GALILEO rolling-ball timer. Unless you have a serial display handy, you'll have to adapt the code, although you can run it entirely from a laptop's serial port at the loss of a little pizzazz. If you get right to it, you can bring it along to your family's Christmas party and impress the daylights out of your nephews (and your nieces, I hope).

Coming next issue: something completely different. ❑

Ed Nisley is a Registered Professional Engineer and a member of the Computer Applications Journal's engineering staff. He specializes in finding innovative solutions to demanding and unusual technical problems.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

I R S

419 Very Useful
420 Moderately Useful
421 Not Useful

Entry-level Embedded Development

On a Shoestring Budget

The market is seeing more and more neat, tiny, inexpensive processors that are perfect for light-duty embedded applications. Jeff checks out the latest from Motorola and is pleased with what he finds.

FROM THE BENCH

Jeff Bachiochi

“grrrind,” answered the starter as it engaged the already purring engine. I took my hand off the keys so I wouldn’t make the same mistake a third time and they jingled as if giggling at my inexperience. Let’s see: was first gear up and toward me, or was that reverse? “Grrrind,” shouted the transmission. Clutch in-that’s better-clutch out. “Buck, ping, buck, ping,” strained the engine while lurching forward in bronco-like spasms. I was off. It seemed like hours until I reached my destination. I applied the brakes and came to an abrupt halt. “Twang.. twang.. twang.. .wang...ang...ang,″ sung the radio antenna as if top keep beat with the AM radio’s high-fidelity four-inch speaker. Clutch in. Too late. “That’s enough for today,” Dad declared, then quickly added, ″You may leave the car at this end of the driveway tonight.”

My first encounter with an auto was quite a few years ago, but only recently did I acquire my “Learner’s Permit.” However, it didn’t come from the Department of Motor Vehicles. This permit was signed by none other than Motorola Inc. If you read any of the trade magazines, maybe a certain Motorola ad lured you in as well:

> **“For just $50, Motorola’s new 68HC705KICS kit can put you on the road to an economical 8-bit design...Hurry, your learner’s permit expires June 30.”**

I judge inflation by the average cost of one bag of groceries. This standard is rapidly approaching $50, so spending that much on food for thought seemed “logical,” as a Vulcan friend would say. Motorola’s bargain consists of an EPROM-version 68HC705 K-series microcontroller; editor, assembler, debugger, and simulator software; an in-circuit source-level simulator/programmer board; and an armful of manuals.

68HC705K

This 16-pin microprocessor (yes, you’ve read correctly) uses the 6805 CPU (see Figure 1). It has 504 bytes of program space and a whopping 32-byte user RAM. The 705K’s on-chip oscillator will function with crystal, resonator, or R/C components. When you use a 4-MHz crystal, the cycle time is 500 ns.

The 68HC705K has five basic pin functions. Pin 1, *RESET, forces the MPU into a known state. Pins 2, 3, and 5-12 are the general-purpose I/O pins. Pin 4, *IRQ, is an external interrupt input. Pins 13 and 14 are power and ground connections, and pins 15 and 16 are the external component connections for the internal oscillator control.

MEMORY

I’ve provided a memory map in Figure 2. Thirteen of the first 32 bytes are reserved as I/O, status, and control registers. Thirty-two bytes of RAM serve as both user and stack RAM. The stack grows downward, using 5 bytes of stack space for each interrupt, and 2 bytes for each call.

The 496 bytes of user program space are OTP (705K1P) or UV erasable (705K1 S), depending on the chip you use. Eight additional locations hold four jump vector addresses for timer overflow, external interrupt, software interrupt, and reset.

A 64-bit array space is used as a personality table. It is merely an extra 8 bytes of bit-accessible ROM.

CPU REGISTERS

Motorola provides two 8-bit general-purpose CPU registers: *A* and X. While you can use both as scratchpad memory, *A* holds operands and arithmetic results, whereas X is an offset pointer to where *A* can find information.

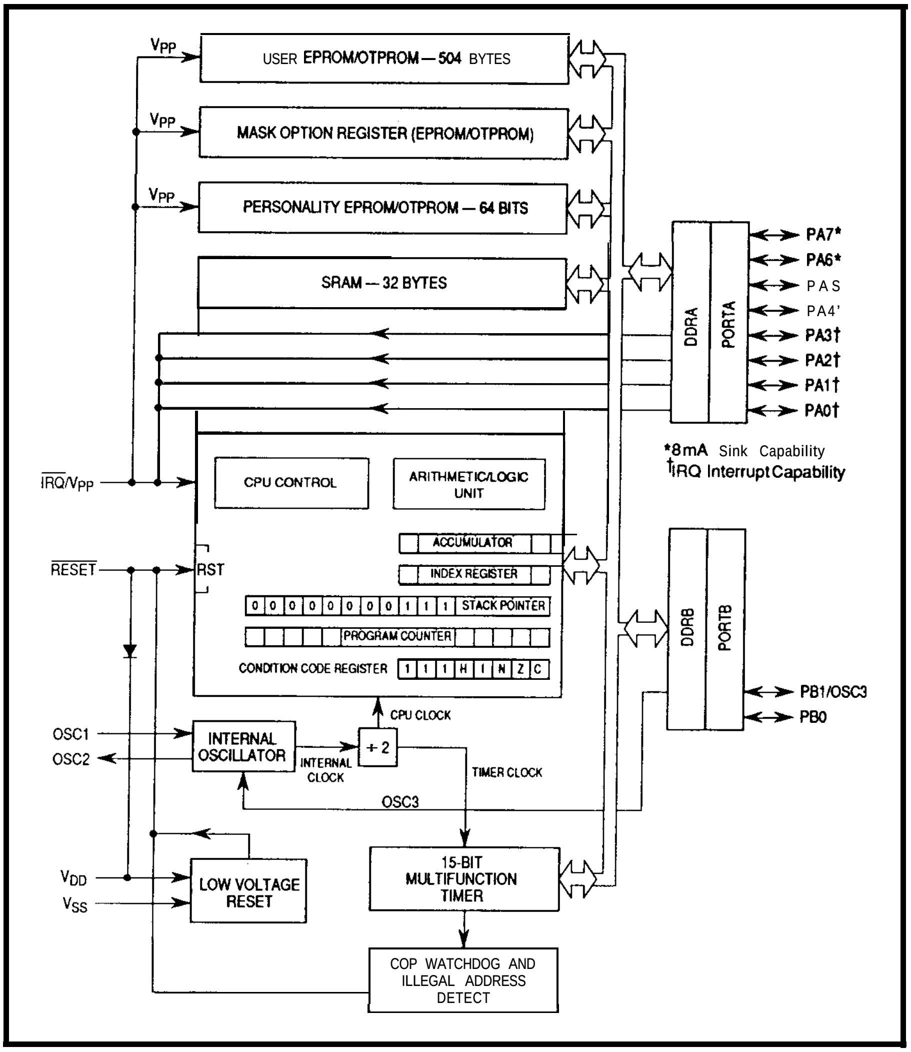


Figure 1—*The 16-pin 68HC705K has 504 bytes of* **program** *space, 32 bytes of user RAM, several I/O pins, and an external interrupt input.*

Only the RS P (reset stack) instruction can alter the next two registers directly. Although the stack pointer is 16 bits, only the first five are used, meaning if the stack drops below the 32 bytes available, it will wrap around and clobber the previously written locations (a catastrophe). The 16-bit program counter contains the address of the next instruction to be fetched. It receives its initial value from the reset vector on reset or power up.

The last register is the 8-bit condition-code register. Five of the 8 bits have significance and are used as flags. The *half-carry* flag indicates a carry between bits 3 and 4 of the *A* register during an ADD instruction. You can use the *interrupt* flag to disable interrupts globally. The *negative* flag indicates a negative by checking for bit 7 set. A manipulation holding a zero result sets the *zero* flag. Finally, the production of a carry or a requirement for a borrow sets the *carry-borrow* flag.

INTERRUPTS

An interrupt can come from a software command, an external interrupt, or a timer overflow. Three bits of the ISC register control external interrupts, and three similar bits of the TSC register timer control overflow interrupts. Interrupt control/status bit functions include an enable/disable bit, a read-only *interrupt-pending* bit, and a write-only *clear interrupt* bit.

RESET

External timing components are unnecessary for reset. The internal timer provides a 4064 clock-cycle delay to stabilize the oscillator. Pulling the • RESET pin low for a minimum of 1.5 cycles accomplishes an external reset. Opcode fetches from any place other than the ROM or RAM locations will cause an illegal-address reset. If enabled, clear the COP watchdog periodically or a reset will occur. If you enable the low-voltage (3.5 volt) detection circuit, VDD falling through the 3.5-volt point will generate a system reset.

Low-power modes available include STOP, WAIT, HALT, and data retention. Using these modes can drop current consumption from the milliampere range to the nanoampere range.

PARALLEL I/O

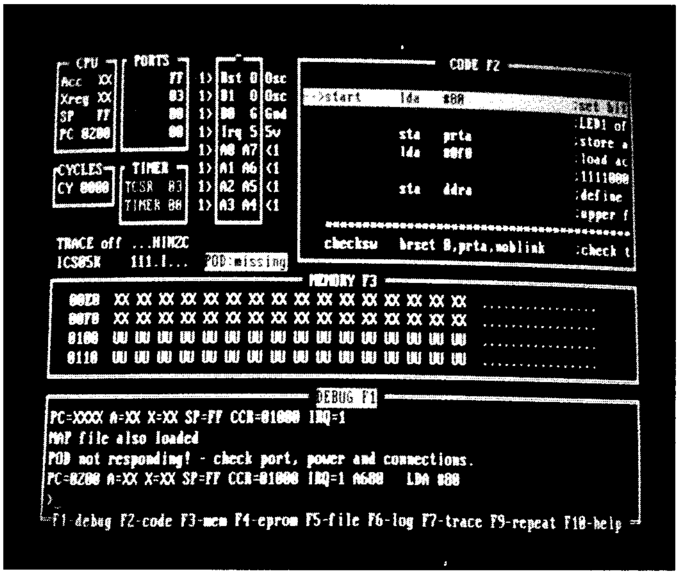
The 10 general-purpose I/O bits divide into two 8-bit ports, *A* and B, both identical in function. *A* uses the full eight bits, and *B* uses the remaining two. Three registers control each port. The data-direction register determines the function of each corresponding bit of the port as an input or output bit. The pull-down register enables a corresponding pull-down for each port bit. The data register holds the I/O data and is a direct reflection of the I/O bits.

TIMER

The CPU clock is derived from the system's internal clock divided by 2. The same internal clock, divided by 2, is further divided by 4 to clock the 15-stage ripple counter. With a 4-MHz crystal, that's a 2-us tick (4 MHz ÷ 2 ÷ 4). A timer overflow interrupt can be generated at the rollover of the first eight stages, every 512 us (2 us x 256). You can program a second interrupt, the real time, for one of four rates: 4.1 ms, 8.2 ms, 16.4 ms, or 32.8 ms (4-MHz crystal). The COP watchdog's time-out periods are eight times greater than the real-time rate: 33 ms, 66 ms, 131 ms, and 262 ms.

PROGRAMMING

The OTP and EPROM devices are both programmed using a 16.5-volt supply attached to pin 4, the ● IRQ input. You use only 3 bits in the EPROM register. Two enable or direct the programming voltage to the EPROM code/personality space or to the mask option register. The third bit enables the internal bus latch used in directing the external program data.



```
 CPU     PORTS                          CODE F2
Acc  XX      FF  1> Rst 0 Osc
Xreg XX      03  1> B1  0 Osc   -->start    lda   #0A            :set bit
SP   FF      00  1> B0  6 Gnd                                    :LED1 of
PC 0200      00  1> Irq 5 5v                sta   prta           :store a
                 1> A0 A7 <1                lda   #0F0           :load ac
CYCLES   TIMER   1> A1 A6 <1                                     :1111000
CY 0000 TCSR  83 1> A2 A5 <1                sta   ddra           :define
        TIMER 00 1> A3 A4 <1                                     :upper f
                                ****************************************
TRACE off ...HINZC              checksu     brset 0,prta,noblink :check t
ICS05K    111.I...  POD:missing
                              MEMORY F3
   0020  XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX  ................
   0070  XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX  ................
   0100  UU UU UU UU UU UU UU UU UU UU UU UU UU UU UU UU  ................
   0110  UU UU UU UU UU UU UU UU UU UU UU UU UU UU UU UU  ................
                              DEBUG F1
PC=XXXX A=XX X=XX SP=FF CCR=01000 IRQ=1
MAP file also loaded
POD not responding! - check port, power and connections.
PC=0200 A=XX X=XX SP=FF CCR=01000 IRQ=1 A600   LDA #00
>_
F1-debug F2-code F3-mem F4-eprom F5-file F6-log F7-trace F9-repeat F10-help
```

Photo 1—*The* simulator, which runs on an *IBM* PC compatible, sets up 10 windows on the screen to display complete *processor status.*

THE 68HC705K'S INSTRUCTION SET

The register/memory instructions include load, store, arithmetic, and logical functions. Although you can pass information between *A* and X directly, no function for passing information directly between memory locations exists, but you can do so with two instructions.

Bit manipulation is an important control function. You can perform read-modify-write instructions on any register or memory location. These instructions include the increment/decrement, clear, complement, negate, rotate, and shift functions. The *set bit* and *clear bit* instructions can change individual bits in all I/O registers, including RAM.

Program control instructions include jump/branch, software interrupt, return from interrupt, return from subroutine, stop, wait, and no-operation functions. Jumps are unconditional, while most branches test one or more flags within the condition code register. Some branches are more specialized, such as checking the status of the *IRQ input or a memory location's bit position. This instruction is handy for creating your own status registers.

IASM05K DEVELOPMENT ENVIRONMENT

The software included in the 68HC705K kit begins with a nice full-screen editor. It contains full cursor movement and function keys that you may relate to: block moves and copies, find and replace, and even a restore (undo) command for that occasional "Oops." You can open multiple files at a time, each in their own window. This capability is great for checking or editing other modules without leaving your present file.

Without leaving the editor, you can call up the cross-assembler to assemble your present file. An error halts the cross-assembler, marks the error on the editor's screen, and forces you to correct the error right then and there

The diskette comes with a file K1EQU.ASM, which defines all port, register, and bit names with the appropriate equates (although some don't match the book exactly). Constants may use decimal, binary, or hexadecimal notations, indicated by a suffix *T, Q,* or *H.* It supports conditional assembly as well as macros. You can produce object files in either Motorola S19 or Intel Hex formats, and it can create a load map file for use in source-level debugging.

This editor package winds up with a communication module that will use COM1 or COM2 for I/O. All parameters-baud, parity, word length, and number of stop bits-are selectable. The communications window, like the editing window, makes beginning and finishing your session all from within one program easy. The simulator/programmer makes the communication module unnecessary, although it is no doubt useful as a stand-alone product.

SIMULATOR

You can enter the simulator directly from the editor, loading up the object and map files automatically. Although you can run the simulator without the emulator/programmer pod attached, you will not be able to see how your target hardware reacts with your software. Also, programming a device without it is, well, unlikely.

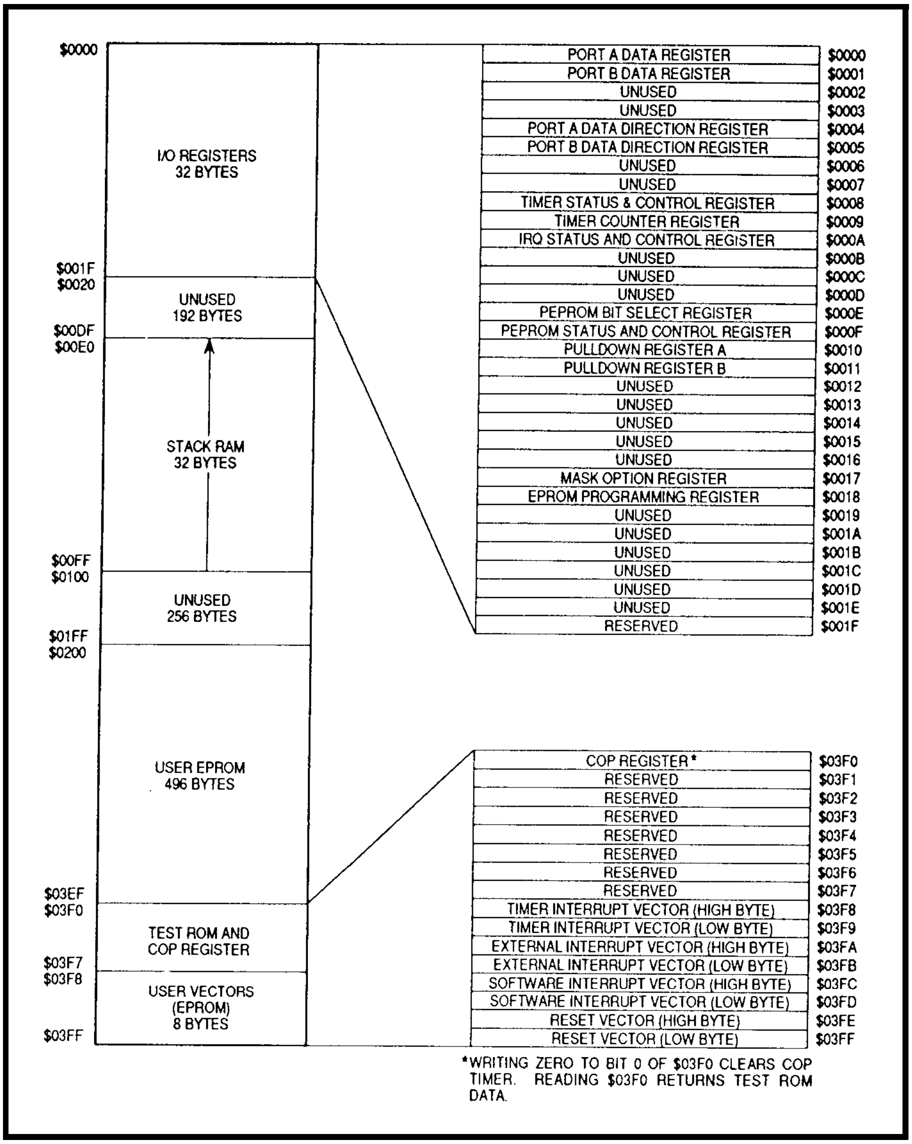
The simulator sets up ten windows into the 68HC705K (see Photo **1).** The CPU window displays the current contents of *A, X, SP,* and PC. The cycles window indicates the status of the timer's least-significant byte. The *trace* window indicates trace status. The *port* window shows the status of the port registers and their associated direction (*I* or 0). The *timer* window reflects the status of the timer control and status registers. The *CCR* window displays the status of the condition code bits.

The *chip visualization* window depicts the chip itself. Arrows on the pins indicate the programmed data direction of the pin. Values outside the arrow show output or input levels. Note that if you use the kit's in-circuit hardware, the inputs come from and the outputs go to your actual target system. The *code* window displays your source code and indicates where the PC is presently pointing. The *memory* window displays 64 consecutive memory locations and may be moved within the memory map.

The *debug* window is where everything happens. This window supports command input and message display during simulation. Commands include: set or remove breakpoint **(B R),** start program at specified address *(GO),* set input (INPUTA or **INPUTB), set** memory **(MM),** simulate IRQ input (**I RQ),** execute script file *(SC***R I P T),** single-step **(ST E P),** enable/disable trace **(TRACE),** and the all important **PRO GRAM** (must have the hardware attached for this one).

THE IN-CIRCUIT EMULATOR/ PROGRAMMER

You're probably saying, "Enough already! OK, so you get lots of flashy software. What about the hardware!" Well, the hardware is fairly straightforward. Motorola's emulator/programmer hardware, a 4-inch printed circuit board, interfaces to your IBM-compatible PC through the parallel port.



$0000
I/O REGISTERS 32 BYTES
$001F
$0020
UNUSED 192 BYTES
$00DF
$00E0
STACK RAM 32 BYTES
$00FF
$0100
UNUSED 256 BYTES
$01FF
$0200
USER EPROM 496 BYTES
$03EF
$03F0
TEST ROM AND COP REGISTER
$03F7
$03F8
USER VECTORS (EPROM) 8 BYTES
$03FF
PORT A DATA REGISTER $0000
PORT B DATA REGISTER $0001
UNUSED $0002
UNUSED $0003
PORT A DATA DIRECTION REGISTER $0004
PORT B DATA DIRECTION REGISTER $0005
UNUSED $0006
UNUSED $0007
TIMER STATUS & CONTROL REGISTER $0008
TIMER COUNTER REGISTER $0009
IRQ STATUS AND CONTROL REGISTER $000A
UNUSED $000B
UNUSED $000C
UNUSED $000D
PEPROM BIT SELECT REGISTER $000E
PEPROM STATUS AND CONTROL REGISTER $000F
PULLDOWN REGISTER A $0010
PULLDOWN REGISTER B $0011
UNUSED $0012
UNUSED $0013
UNUSED $0014
UNUSED $0015
UNUSED $0016
MASK OPTION REGISTER $0017
EPROM PROGRAMMING REGISTER $0018
UNUSED $0019
UNUSED $001A
UNUSED $001B
UNUSED $001C
UNUSED $001D
UNUSED $001E
RESERVED $001F
COP REGISTER* $03F0
RESERVED $03F1
RESERVED $03F2
RESERVED $03F3
RESERVED $03F4
RESERVED $03F5
RESERVED $03F6
RESERVED $03F7
TIMER INTERRUPT VECTOR (HIGH BYTE) $03F8
TIMER INTERRUPT VECTOR (LOW BYTE) $03F9
EXTERNAL INTERRUPT VECTOR (HIGH BYTE) $03FA
EXTERNAL INTERRUPT VECTOR (LOW BYTE) $03FB
SOFTWARE INTERRUPT VECTOR (HIGH BYTE) $03FC
SOFTWARE INTERRUPT VECTOR (LOW BYTE) $03FD
RESET VECTOR (HIGH BYTE) $03FE
RESET VECTOR (LOW BYTE) $03FF
*WRITING ZERO TO BIT 0 OF $03F0 CLEARS COP TIMER. READING $03F0 RETURNS TEST ROM DATA.

Figure 2—*The 68705K's I/O registers, RAM, user ROM, and configuration ROM all fit within the firsf 1K of address space.*

The programmer circuitry consists of a low-insertion-force test socket for the 68HC705K and an on-board DC-DC converter, which generates the + 16.5 volts necessary for programming the 68HC705K from a 5-volt source (that you supply).

The emulator consists of a preprogrammed 68HC705J and ribbon cable for plugging into your target system's processor socket. The 68HC705J runs at full speed, providing target outputs and reading target inputs. It communicates with the simulator at a speed relative to your system's clock speed through the parallel port, providing output data to the target system and retrieving input for the simulator. Even if the simulator cannot run in real time, this combination is extremely useful in exercising the code with actual hardware (except for time-dependent routines).

The hardware has an LED and push-button switch attached to two of the emulator's I/O pins. This equipment is enough to allow a user to test the software-hardware combination by running a "blinky" program, also provided on the diskette.

CONVERSION PROJECT

After looking at this device, I wasn't sure if I could do anything practical with it, not to mention the

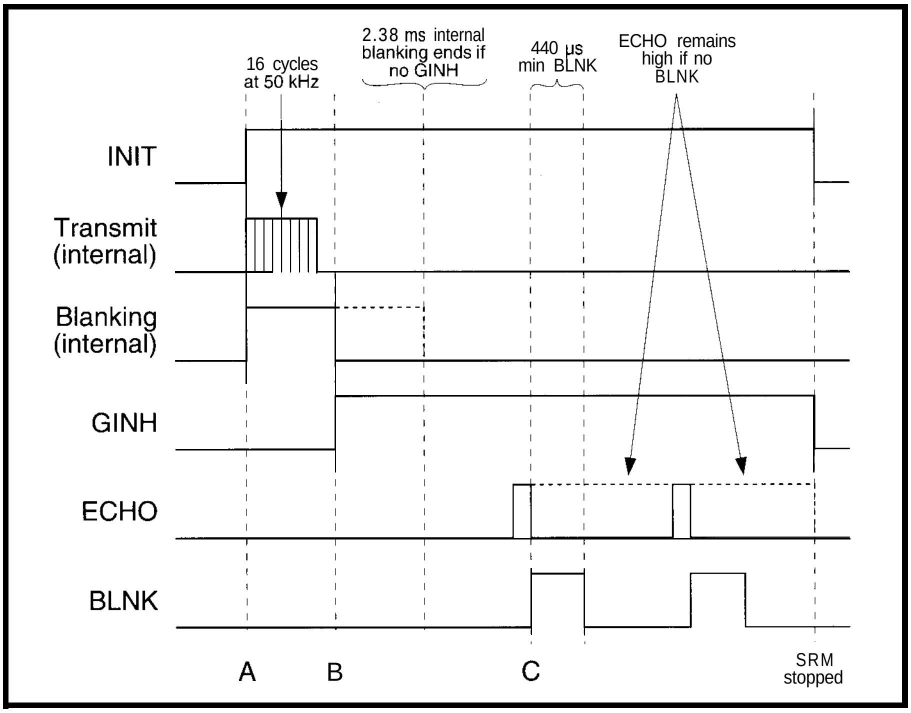


Figure 3—*Point* (a) shows the *timing for a normal ultrasonic echo; point (b) demonstrates how to cancel the internal blanking to catch an object closer than 1.33 feet; and point (c) shows how to use the BLNK input to capture multiple echoes.*

conversion project I had in mind. However, as I outlined what was necessary and which shortcuts I could take, possibilities began to develop.

Distance measurement using an ultrasonic transducer has always fascinated me. Polaroid has used one for years in their automatic cameras. Stanley marketed an instrument for measuring room dimensions, although seeing them on the surplus market now raises a few questions. Was it inaccurate? Was the public just not ready for such technology? Or was this a complex solution for a simple task? From a surveyor's point of view, it was an accuracy problem. Joe Average has enough trouble with a tape measure, not to mention another gizmo with a perpetual dead battery. I suspect a wall-to-wall carpet estimator may have found it useful. After all, they just round up to the nearest yard, anyway!

The embedded application I have in mind for this Motorola part is to convert the time travel of ultrasonic waves into readable text, acceptable by most computers. With a touch of software and the 68HC705K, you can make it an addressable module that will convert its "How long did it take?" gate into a serially transmitted "How far away is it?" string. The Stanley estimator could have had a more industrious life if only its umbilical cord had not been cut so short.

SONAR RANGING

I remember, as a child, counting the number of seconds between seeing a lightning flash in the distance and hearing the sound of thunder. That number divided by five supposedly indicated the number of miles to where the lightening bolt had hit the ground. There was nothing scientific about it back then; it was just a game. Today, some things have changed, but the velocity of sound traveling through air at STP is still 1087.1 ft./sec. (STP = standard temperature and pressure, that is, 0°C and 14.70 lb./in.^{2}).

Counting seconds between lightning and thunder is the principle behind the sonar ranging module (SRM). This device was first produced by Texas Instruments for Polaroid's autofocus cameras. The circuit consists of a 50-kHz oscillator, gated to drive an ultrasonic transmitter for 16 cycles. A receiver then listens for the 50-kHz burst. As time goes by, the gain of the receiver cranks up, attempting to make up for the loss in signal strength due.to dispersion. Any object within the path of the ultrasonic beam will reflect back a bit of the burst. The receiver indicates an acquired echo by raising its output.

The user has control of three functions: INIT, BLNK, and GINH. Raising the INIT input will transmit the 16-cycle packet, and lowering it cancels the transmit/receive cycle. The second input, BLNK, will cancel the current echo indication and start listening for more. It must be held high for 440 µs so all 16 cycles can pass and not be recognized as another echo. The final input, GINH, will cancel the internal blanking, permitting the acknowledgment of echoes for objects closer than 1.33 feet. Transmitter ringing may be detected as legal echoes, depending on how the transducer is mounted. Internal blanking creates a safety zone to guard against these false images. In Figure 3, point A shows the timing for a normal echo, point B demonstrates how to cancel the internal blanking to catch an object closer than 1.33 feet, and point C shows how to use the BLNK input to capture multiple echoes. The SRM has only one output: ECHO.

If sound travels 1087.1 feet in 1 second, then it also travels only about 1/1000 of a foot every microsecond, something a microprocessor can handle. The 68HC705K's timer tick is 2 us, or about 1/500 of a foot. Accumulating these ticks from the rising edge of INIT to rising edge of ECHO, will indicate the amount of time the 16-cycle packet of 50 kHz takes to travel out to an object, reflect back, and be acknowledged. From this elapsed time, you can calculate a distance.

DISTANCE

Documentation indicates the range of the SRM is 6 inches to 35 feet. Understand that objects closer than 1.33 feet will cause erroneous readings, unless the internal blanking is inhibited using the GINH input. Even this can cause bad readings, depending on how the transducer is mounted. Also, the receiver's amplifier reaches maximum gain at about 38 ms. For the transmitted packet, that's 19 ms out and 19 ms back, or about 19 feet.

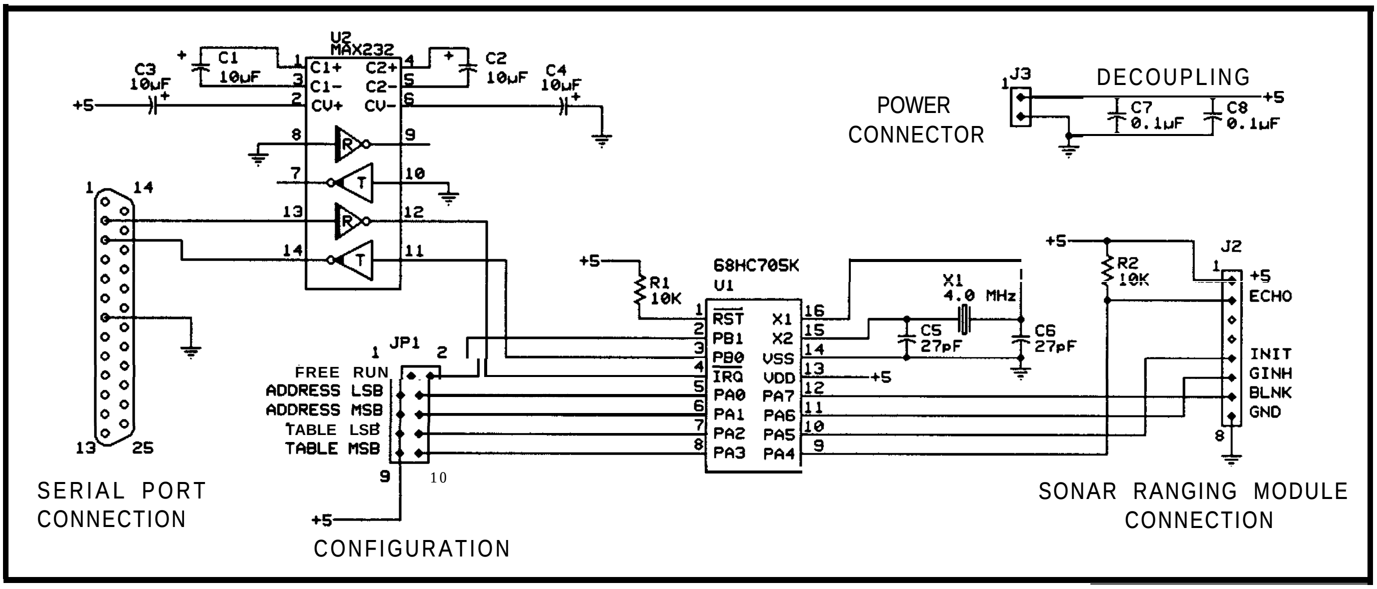


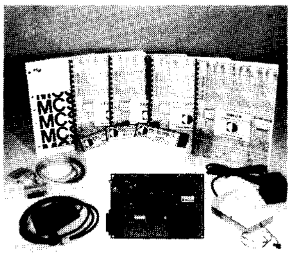
Figure *4-A smart serial interface to an ultrasonic transducer can be built with a single 16-pin 68HC705K processor and a* handful *of support components.*

Objects farther than I9 feet are amplified by maximum gain only, while their signals continue to deteriorate with distance.

Distance is calculated by multiplying the time by the speed of sound (total distance = time x 1087.1 ft./sec.). Halve the total distance to find the distance to the object. For example, if the time is 22 ms, then the actual distance is 11.9 feet (0.022 sec. x 1087.1 ft./sec. ÷ 2).

INTERFACING

If full control of the SRM is unnecessary, use only one digital output (to INIT) and input (from ECHO). For full control, you need three digital outputs (to INIT, GINH, and BLNK) and one digital input (from ECHO). You may choose to give your processor the job of timing and converting the time to distance by directly interfacing to the SRM. I want



to deal with converted information only, and leave my processor free for other tasks. To do this, I want to use the 68HC705K as an embedded processor.

Figure 4 shows the circuit used to interface the SRM to your PC (or any device with a serial port). My prototype and the Motorola development board are shown in Photo 2. The interface to the PC is through the RS-232 serial port using a simple three-wire connection. The 68HC705K receives serial transmissions from the PC at the *IRQ input. The '705K uses one I/O pin to send conversions back to the PC, leaving nine remaining. Optionally, a diode in the RS-232 transmit side allows tying multiple circuits together to one serial port. While this wire ORing is not allowed under RS-232 rules, the PC's serial port can only enable one circuit at a time to respond, so it avoids collisions.

Each circuit is addressable as 0, 1, 2, or 3. The address assignment requires two I/O pins, leaving seven. Four I/O pins are used as the SRM interface (three out and one in), leaving three. Two of the last three I/O pins are used to select the conversion factor. You can set up the converted output for English or Metric units. The final I/O pin selects the conversion mode. When using only one SRM, the autosend mode will continually spit out conversions at the rate of ten per second. You can use the manual mode to initiate a conversion or read back the results from one of the four addressable circuits on the RS-232 daisy chain.

COMMUNICATIONS

You can initiate conversions in one of three ways. Sending an S command code will start a normal conversion, accepting an echo anytime after the internal blanking (1.33 feet).

The B command code will start a normal conversion. However, after it receives an echo, the BLNK input is used to reset it and continue listening for more (I store up to eight).

The I command is much the same as the B command, except the GINH output eliminates the internal blanking. This allows echoes immediately, not limiting the echo to a minimum of 1.33 feet.

The four input commands all have the same three-character format: #c<cr>, where # is a single-digit decimal address, c is the command, and <cr> is a carriage return.

The A command retrieves all the echoes saved by one of the first three commands. The output format is #####c<cr>, where ##### is a five-digit decimal number, c is the conversion indicator, and <cr> is a carriage return.

GO WITH THE FLOW

You can break down the code written for 68HC705K into five simple routines (see Figure 5). The first three are parts of the main loop, while the remaining two are interrupt routines.

The main loop is a state machine beginning with a WAIT statement. Execution continues only after a character is received. If the character falls into the legal #, c, or <cr> byte for the present state, then the present state is increased. If the final state has not been reached, a jump is made back to the WAIT statement. However, if any illegal characters are received, the state is cleared before looping back to the WAIT. If the final state is reached, execution continues in one of the two final sections.

If the command is an S, B, or I, then execution drops into a routine that takes one or more echo measurements by toggling the SRM's inputs and keeping track of the time expired

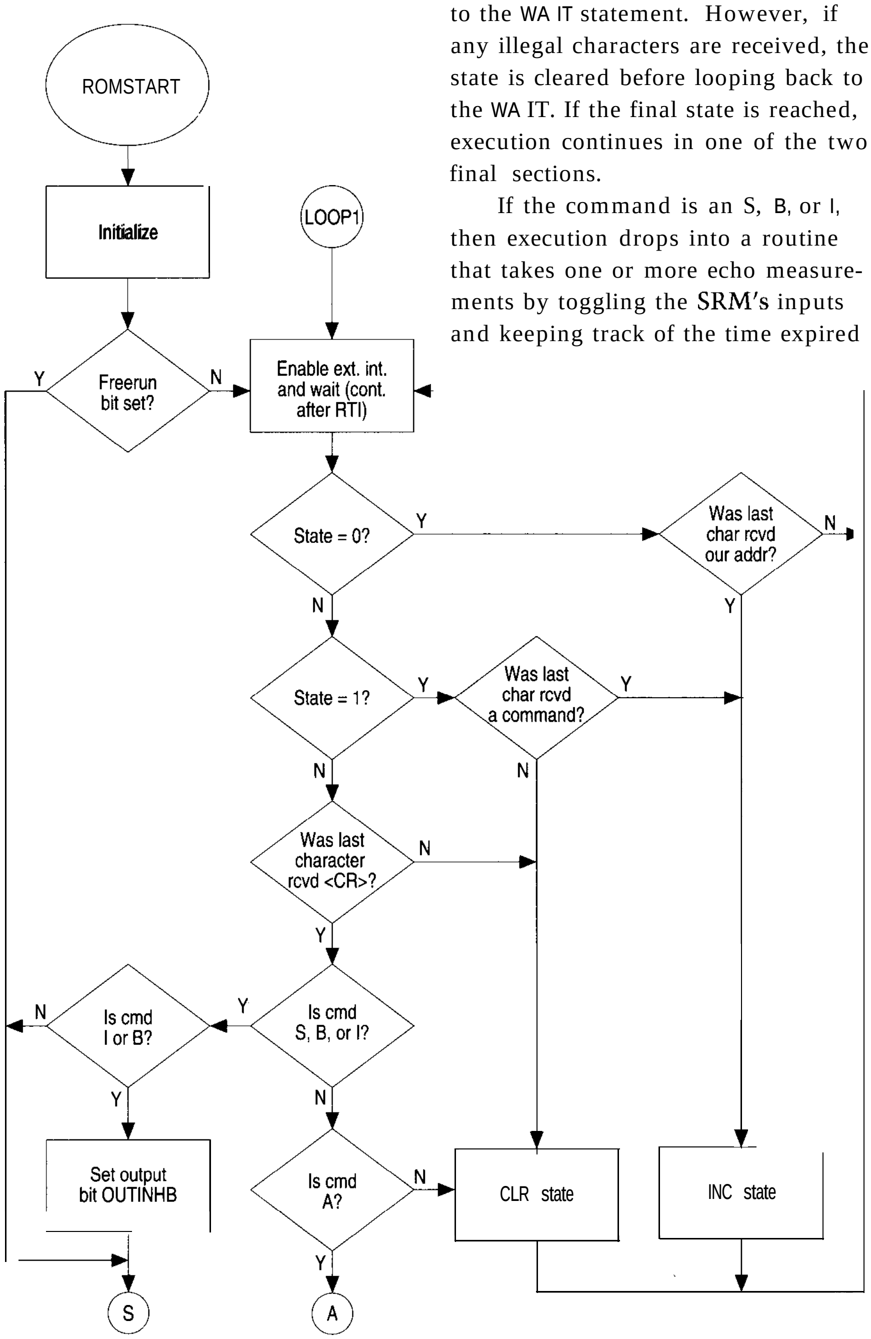
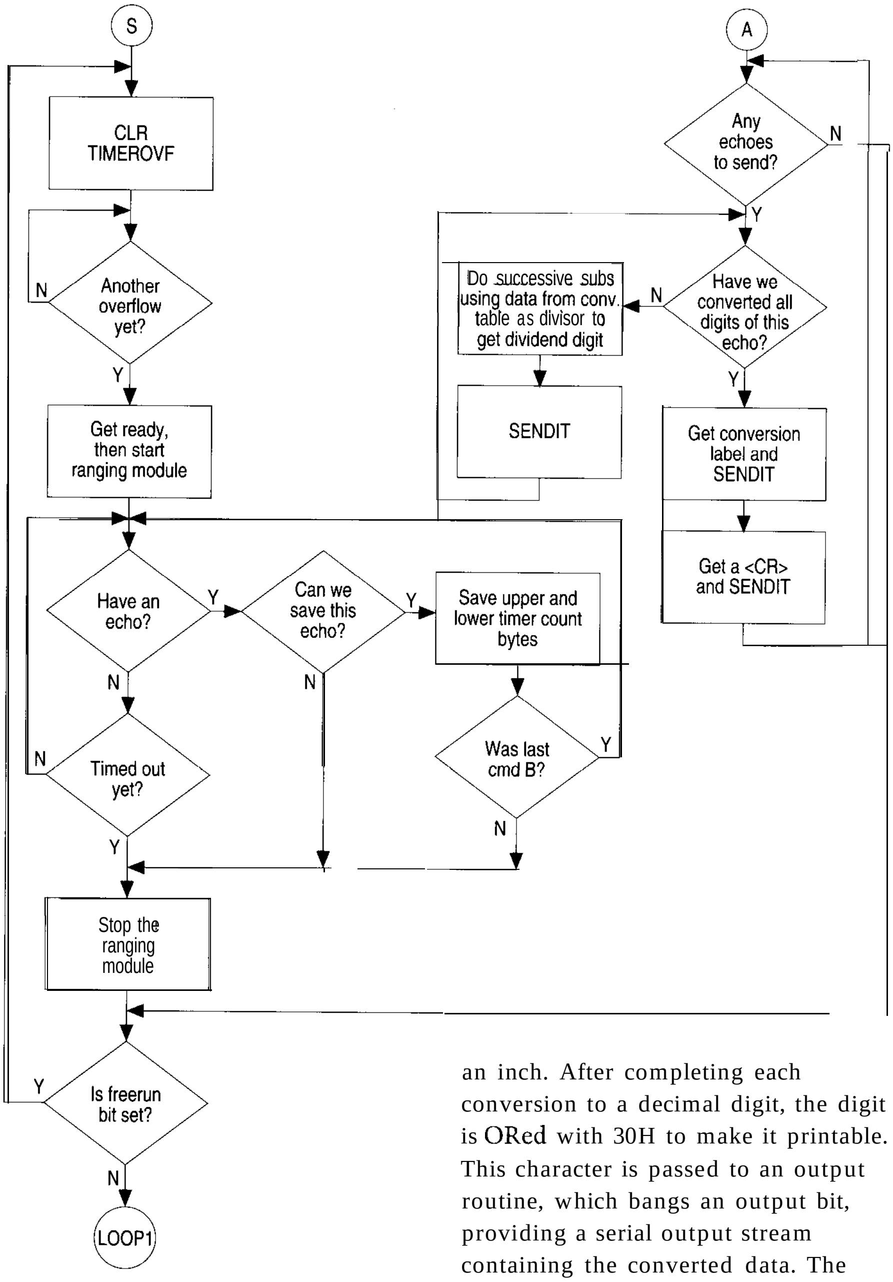


Figure 5a—*The* main loop of *the* ultrasonic support *code is a simple state* machine that *supports four user* commands (continued on next page).


S
CLR TIMEROVF
Another overflow yet?
N
Y
Get ready, then start ranging module
Have an echo?
Y
N
Can we save this echo?
Y
N
Save upper and lower timer count bytes
Was last cmd B?
Y
N
Timed out yet?
N
Y
Stop the ranging module
Is freerun bit set?
Y
N
LOOP1
A
Any echoes to send?
N
Y
Have we converted all digits of this echo?
N
Y
Do successive subs using data from conv. table as divisor to get dividend digit
SENDIT
Get conversion label and SENDIT
Get a <CR> and SENDIT

until the echo output goes high, signaling an object. The timer's least- and most-significant bytes are saved in RAM (up to eight echoes), and a jump is made back to the main loop.

If the command is an **A,** then the last routine is entered. Here, any echoes saved are converted from counts (time) into units of length [distance). Successive subtractions are used to provide a dividend for each place: tens of thousands, thousands, hundreds, tens, and ones. Input jumpers select a conversion divisor table used to convert the count into distance, either in millimeters, thousandths of a foot, or hundredths of an inch. After completing each conversion to a decimal digit, the digit is ORed with 30H to make it printable. This character is passed to an output routine, which bangs an output bit, providing a serial output stream containing the converted data. The five-digit decimal number is appended with a letter, signifying the conversion table used, and finally a <cr>. After all the recorded echoes are sent, control again returns to the main loop.

The alternate free run mode starts the SRM and reports the first echo encountered. This mode requires no user input and repeats continuously.

I chose the external interrupt, 'IRQ, as the serial input bit because it is capable of breaking out of a wait state and has individual branch instructions devoted to both polarities. A start bit will trigger an *IRQ and jump to the external interrupt routine. Because the 68HC705K has no hardware serial port, reading the serial bit stream is done in software. The code pauses one-half bit time to align itself with the center of each serial data bit. The polarity of the *IRQ pin is sampled after pausing one bit time for each of the next nine. The bits are rotated into a register, rebuilding the data byte. The first bit (the start bit) gets dropped, leaving only the 8 data bits. The stop bits following the data byte are ignored. This step allows extra time to process the input on-the-fly without the need of an input buffer.

The timer interrupt is simply a rollover of the 8-bit TCNT register every **512** µs **(2** µs **x 256** counts). By incrementing a RAM register every interrupt, you can expand the timer from **512** µs **(8** bits) to 128 ms **(16** bits). This value is well within the 80 ms needed for the ultrasonic wave propagation time.

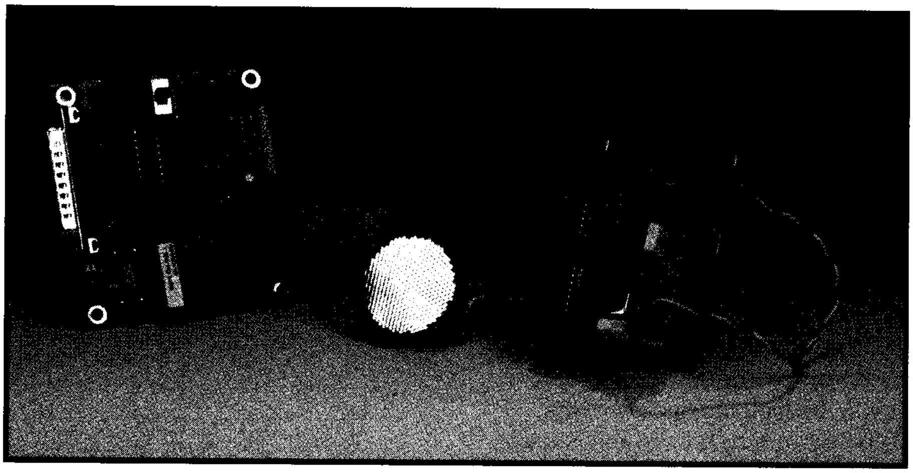


Photo **2**—*The T.I. ultrasonic transducer **interface** board plugs on fop of the profofype containing the 68HC705K and serial interface. The Motorola development **board** plugs info the profofype in p/ace of the processor during code development.*

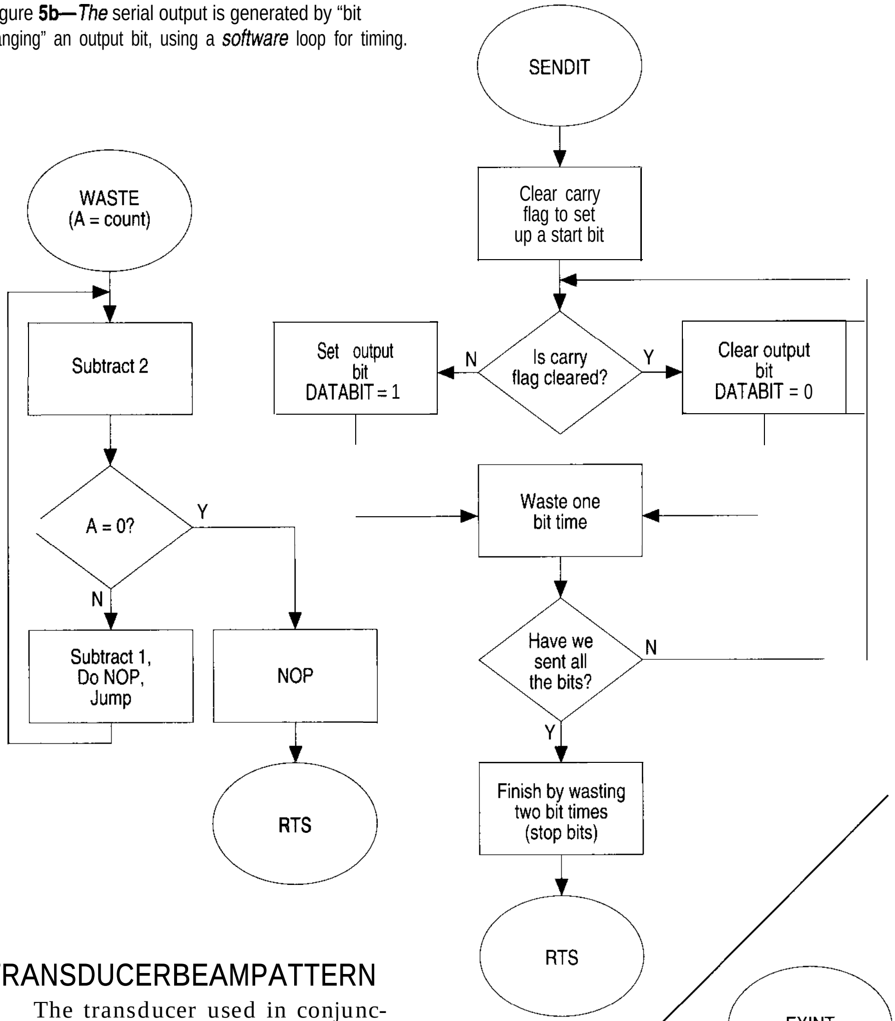

SENDIT
Clear carry flag to set up a start bit
Is carry flag cleared?
N
Y
Set output bit DATABIT = 1
Clear output bit DATABIT = 0
Waste one bit time
Have we sent all the bits?
N
Y
Finish by wasting two bit times (stop bits)
RTS
WASTE (A = count)
Subtract 2
A = 0?
Y
N
Subtract 1, Do NOP, Jump
NOP
RTS

Figure **5b—***The* serial output is generated by "bit banging" an output bit, using a *software* loop for timing.

TRANSDUCERBEAMPATTERN

The transducer used in conjunction with the SRM is made up of a stainless steel housing, which is physically (electrically) connected to a 24-carat gold-covered Kapton Film diaphragm. This terminal is usually connected to the SRM's ground, keeping the housing at a safe ground potential. The hot lead connects to a spring clip, holding a stainless steel disk against the back of the film diaphragm. This connection forms a large disk capacitor with one stiff and one flexible plate. Although the transducer current is small, the few hundred volts produced during a transmission burst by the step-up transformer can be rather uncomfortable if touched.

Typical beam patterns are normally given for objects of 1 ft.^{2}, held perpendicular to the beam's point of origin. In applications where the reflected surface is large and perpendicular, the echoes are simple and predictable. However, when surfaces are curved or at oblique angles with multiple surfaces, echoes are picked up not only from direct reflection, but also from indirect reflection. The indirect reflections give three or more paths to the echo where the overall distance traveled is more than out and back. These echoes suggest the object is farther away than it actually is.

My own test results show the pattern of detection for this transducer (using a spherical object) to be a cone of almost 15" off of the transducer's perpendicular center line.

KEEPING YOUR DISTANCE

Texas Instruments has produced the SRM for many years. Polaroid, probably TI's largest OEM, is now manufacturing their own. Polaroid has also designed an environmental transducer able to exceed the SAE J1455 1/88 specifications for heavy duty trucks.

This feature may suggest to you applications such as collision avoidance, docking, and automatic guidance. How about motion detection, proximity, height, or size estimation? The sonar ranging module has possi-

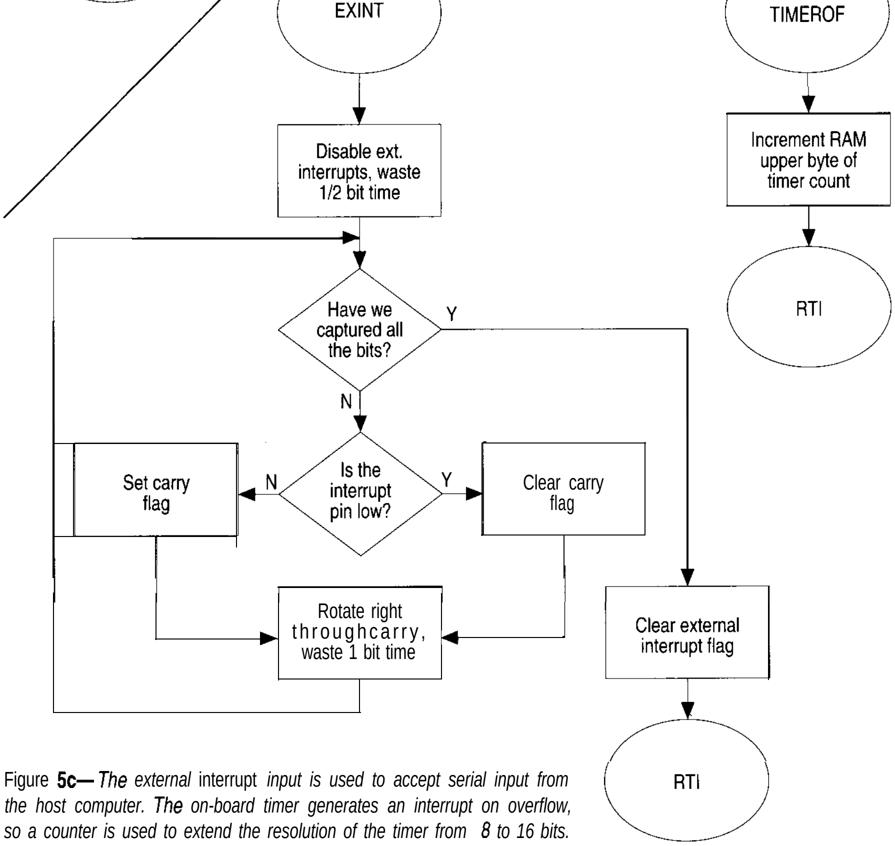

EXINT
Disable ext. interrupts, waste 1/2 bit time
Have we captured all the bits?
Y
N
Is the interrupt pin low?
N
Y
Set carry flag
Clear carry flag
Rotate right throughcarry, waste 1 bit time
Clear external interrupt flag
RTI
TIMEROF
Increment RAM upper byte of timer count
RTI

Figure **5c—** *The external* interrupt *input is used to accept serial input from the host computer. The on-board timer generates an interrupt on overflow, so a counter is used to extend the resolution of the timer from 8 to 16 bits.*

bilities where other sensing devices just can't measure up.

Oh yes, I talked to the Motorola rep. Although the 68HC705K developer's package is no longer available for $50, you can purchase one for $160. If you consider what you're getting, it's still a great deal. The EPROM version of the chip costs about $13 and the OTP version about $4. I hope I've convinced you that this kit *is* a cost-effective solution if it fits your needs. If not, maybe next time. ▣

Jeff Bachiochi (pronounced "BAH-key-AH-key") is an electrical engineer on the Computer Applications Journal's engineering staff. His background includes product design and manufacturing.

SOFTWARE

Software for this article is available from the Circuit Cellar BBS and on Software On Disk for this issue. Please see the end of "ConnecTime" in this issue for downloading and ordering information.

SOURCES

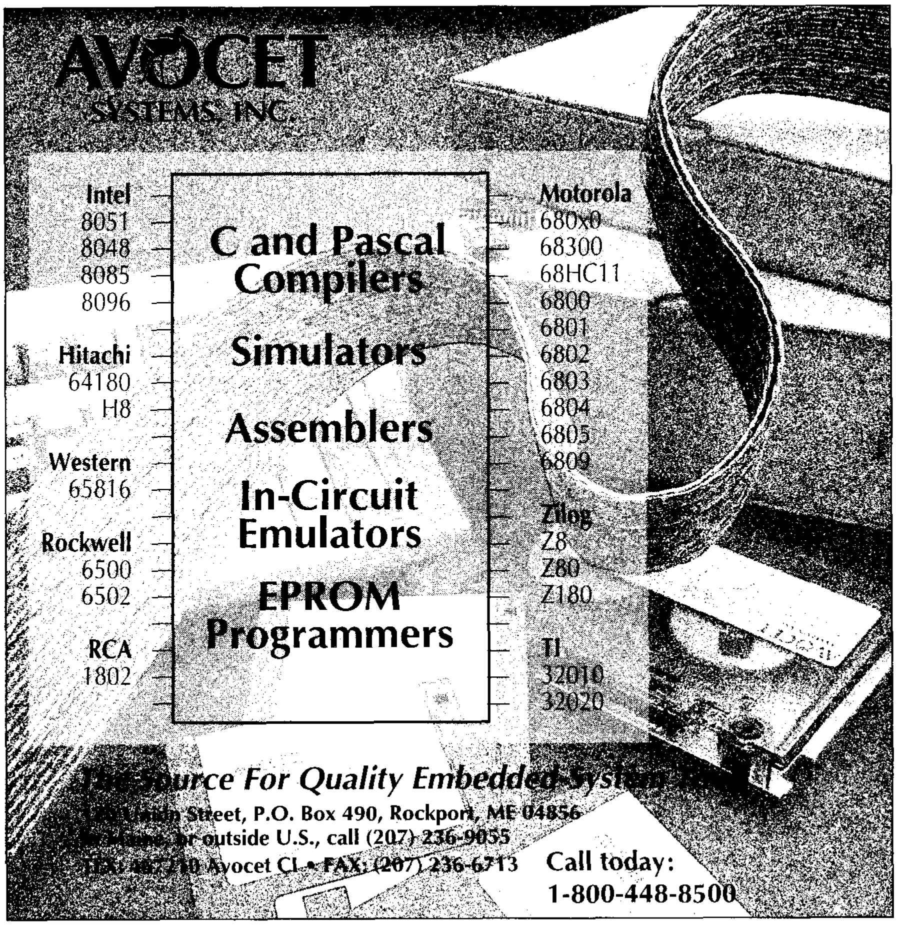
Motorola, Inc.
Semiconductor Products Sector
Microprocessor and Memory
Technologies Group
6501 William Cannon Dr. West
Austin, TX 787358598
(512) 891-2000

TI01 Sonar Ranging Module, $79
Micromint, Inc.
4 Park St.
Vernon, CT 06066
(203) 871-6170
Fax: (203) 872-2204

Ultrasonic Ranging System
Developers Kits, $99 to $295
Polaroid Corp.
I19 Windsor St.
Cambridge, MA 02 139
(617) 577-4681
Fax: (617) 577-3213

I R S

422 Very Useful
423 Moderately Useful
424 Not Useful



SILICON UPDATE

Hot Chips IV

Silicon Sizzlers

Tom Cantrell

Tom's cooking again with some new chips fresh off the grill. Check out the latest megaprocessors from DEC, AT&T, and Intel, plus some research going on at Stanford University.

The chips weren't all that were hot at the recent Hot Chips IV conference. As the temperature soared into the nineties on the Stanford campus, many of the attendees were clearly suffering. Given their nocturnal lifestyle, I kept expecting the chipheads to collapse and writhe under the sun's relentless rays like Count Dracula.

The heat was only fitting because the latest hot chips are just that-hot! For instance, consider the DEC Alpha microprocessor.

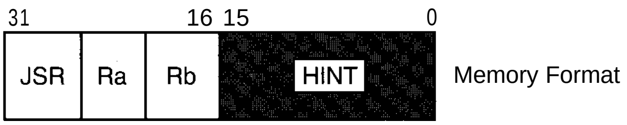
Alpha is DEC's response to their competitor's RISC-based workstations, which have successfully hammered the VAX in recent years. As the newest architecture on the block, Alpha reflects the latest trends in the computing community.

One basic advance is the shift from 32-bit to 64-bit address/data. As John Mashey of MIPS points out, 32 bits isn't enough even to track our national debt, while 64 bits could handily express it-even in pesos!

Yes, other applications demand 64 bits, such as (fittingly) the simulation of multimillion transistor chips. Furthermore, history is on the side of 64-bit proponents because every previous memory address limit has ultimately proven style cramping. Nevertheless, I think 32 bits will be enough for mere mortals for some time to come.

Another recent trend is the move towards "hints" in which the application programmer has to (or gets to, if you're an optimist) give the compiler and chip some help. Compilers and chips have been advanced about as far as they can in terms of guessing a programmer's plans so as to implement them more speedily. One form of hint Alpha offers is a FETCH instruction, which allows a programmer to start loading data into the cache before it is actually used.

Branches, the bane of pipelined machines, continue to receive a lot of attention. The solution on the newest chips, including Alpha, is to incorporate some form of branch prediction hardware. Interestingly, branch prediction is in direct conflict with yesterday's favorite ***delayed branch*** technique. Branch prediction accepts branch delays as a given and attempts to schedule useful instructions during the gap. If the compiler is unable to find any useful instruction, it inserts



disp <15:14>	Meaning	Predicted Target <15:0>	Prediction Stack Action
00	JMP	PC + {4* disp <13:0>}	–
01	JSR	PC + {4* disp <13:0>}	Push PC
10	RET	Prediction stack	Pop
11	JSR_COROUTINE	Prediction stack	Pop, push PC

Figure I-One *method the newer chips are starting to use to speed up branches is the* **encoding of** *"hints" coded info the instruction. The programmer usually knows when writing the program which way a branch will most often go, so the processor can start preparing* **for if ahead** *of time.*

NOPs to keep the pipeline in sync. DEC calls schemes like delayed branch *First Implementation Artifacts* (i.e., it worked well at the time, but proved problematic later). The gotcha is that software compiled for a machine with branch delays won't work if the branch delay is reduced.

Going beyond passive branch prediction, Alpha also applies the hint concept to branches. After all, nobody knows better than the programmer which way a branch is likely to go. As shown in Figure 1, the jump opcode includes space for 16 bits worth of hints. Two of these bits are used to differentiate the type of jump (i.e., regular JMP, JMP to subroutine, RET from subroutine, etc.), which itself is a type of hint. The remaining bits specify the predicted target address, allowing on-chip hardware to get things started ahead of time.

Clearly, superscalar techniques are in vogue, having been adopted by DEC and nearly everyone else. Exploiting multiple function units (integer, float, branch, and load/store) means Alpha can issue two instructions at once. The advantages of superscalar depend on the restrictions placed on allowable instruction pairs and the ability of the compiler to schedule within those restrictions. Judging by the proliferation of superscalars, dual-instruction machines clearly make sense. However, the jury is still out on machines handling more than two instructions because the benefits decline and the cost skyrockets.

Oh yeah, back to the heat wave. With 1.68 million transistors packed in a 431-pin PGA running at 200 MHz, the Alpha dissipates an amazing 30 W!

LORD OF THE CHIPS

The AT&T "Hobbit" microprocessor isn't as "hot" as Alpha because it is targeted towards "personal communications" (i.e., battery powered) applications and thus needs to be "cool." Hobbit deserves mention since it marks the return of some CISCy concepts that many thought were buried on the ash heap of computer history.

Size Comparisons of Cross Compiled Hobbit PCC

Machine	Bytes	Relative
VAX	72K	1.0
Hobbit	74K	1.0
M68000	86K	1.2
R3000	128K	1.8
IBM 370	134K	1.9
SPARC	141K	1.9

Figure 2—*AT&T's Hobbit processor achieves a code density almost as good as a VAX and better than its competitors.*

The Hobbit is derived from a circa 1980 effort at Bell Labs to create a machine called CRISP, which stands for C Reduced Instruction Set Processor. Measured against RISC dogma, the CRSIP/Hobbit is fatally flawed by memory-to-memory architecture in which instructions operate on memory (versus RISC load/store in which instructions only operate on registers) and variable-length (2-, 6-, and 10-byte) instructions (RISC calls for fixed-length instructions to speed up and simplify instruction decoding).

In case you've forgotten, there is virtue associated with the CRISP/Hobbit CISC features—namely, improved code density (see Figure 2). Beyond the obvious reduction in system memory cost and power usage, improved code density gets more mileage from an instruction cache (more instructions fit) and bus bandwidth (more fetched per access).

The Hobbit block diagram (see Figure 3) shows a few other unique features. Notice that by the time instructions are put in the instruction cache, they are already decoded into the internal 192-bit control word format. Thus, the decoding step can be skipped for cached instructions.

Also, notice the adoption of a stack cache. This great idea seems obvious and I'm surprised it isn't universally used. After all, the benefit of a cache is due to the locality of the reference, and a stack is almost always accessed in an orderly manner (i.e., the few entries near the top are where the action is). According to AT&T, the tiny 256-byte stack cache hits up to 80% of stack references.

P5 NONANNOUNCEMENT

Let's face it. The Hot Chip IV organizers didn't schedule the P5 stuff last by accident. With the heat and a big lunch under my belt, I was tempted to bail out on afternoon sessions such as "Chip Pair Creates Self-Timed Network Fabric for Paragon Parallel Supercomputers" and "GaAS VLSI Enhancement through Utilization of Global Optical Free Space Smart Interconnects." Nevertheless, the P5 bait worked and I stayed glued to my seat.

When the handouts were held back until immediately prior to the session, I suspected this wouldn't be a

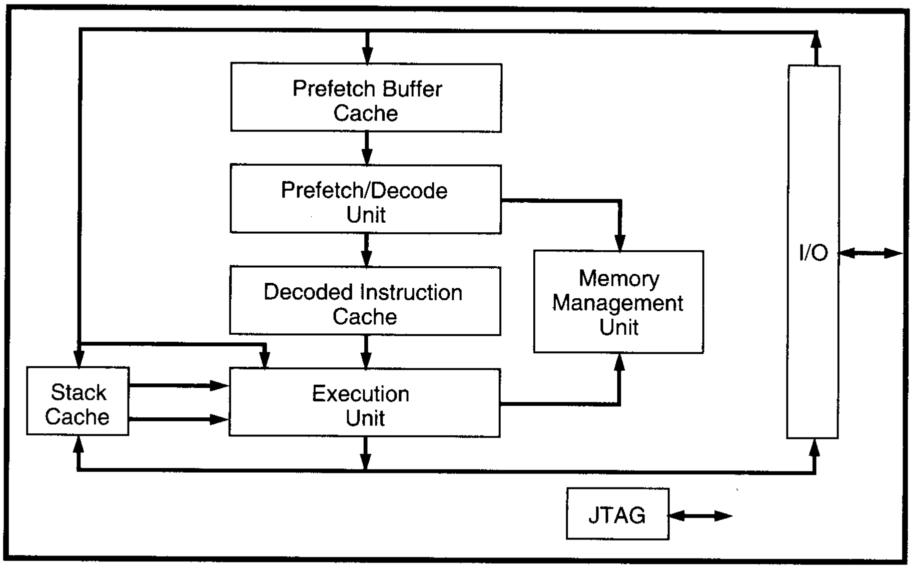
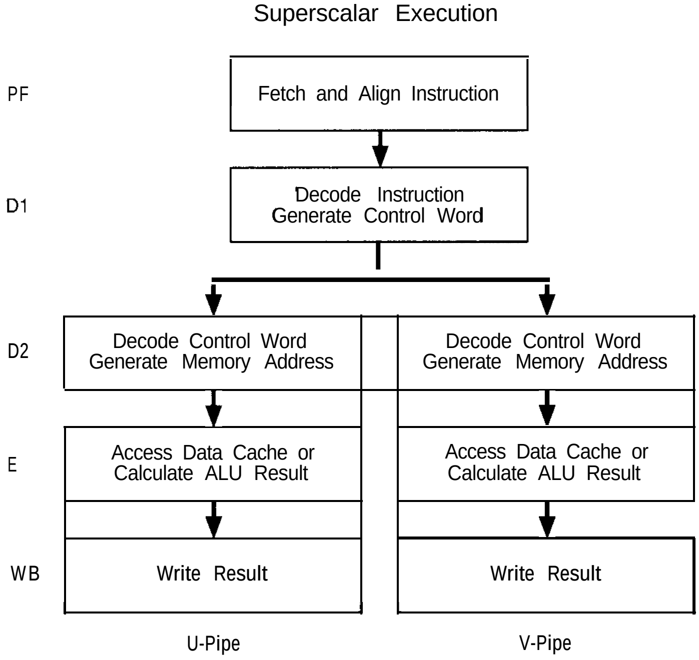


Figure 3—*The AT&T Hobbit automatically decodes instructions by the time they reach the instruction cache, so the decoding step can be skipped for cached instructions.*



Instruction Issue Algorithm

```
Decode Two Consecutive Instructions: I1 and I2
If the Following Are All True
    I1 Is a "Simple" Instruction
    I2 Is a "Simple" Instruction
    II Is Not a JUMP Instruction
    Destination of I1≠ Source of I2
    Destination of I1≠ Destination of I2
Then Issue I1 to U-Pipe and I2 to V-Pipe
Else Issue I1 to U-Pipe
```

"Simple Instructions Are Generally ALU or MOV Operations, Including Reg-Reg, Imm-Reg, Mem-Reg, and Reg-Mem Formats, and JUMPS

Figure 4-*The mysterious Intel P5 remains a mystery when it comes to details. Information released so far indicates if is to be a superscalar processor with dual instruction pipelines.*

typical presentation. This trick caused something of a mad rush-nothing like festival seating at a heavy metal concert mind you, but still disconcerting enough when you're accustomed to the usual mild-mannered behavior of computer types. The panic turned out to be unwarranted because the document was remarkably undetailed.

Perhaps the most interesting point to be gleaned from the materials (see Figure 4) is that the P5 is indeed superscalar. I found this information rather surprising because conventional wisdom dictates that CISCs (if the 'x86 isn't, what is?) aren't amenable to issuing multiple instructions. In particular, aligning and dispatching variable-length instructions is problematic. Some experts speculate that, like the Hobbit, the P5 stores predecoded and prealigned, rather than raw, instructions in the cache.

The adoption of superscalar techniques in the P5 raises the issue of compatibility. Of course the chip can run existing PC binaries, but achieving top performance will require recompiling applications with a compiler that schedules instructions per the superscalar issue restrictions. Thus, I imagine we'll all be deluged with P5-optimized software upgrade offers.

The talk got off to a rather bad start when the speaker revealed he was being forced to read a "legal statement." The intelligentsia greeted this news with boos and hisses, although I noted the indignation failed to propel anyone from the room. The aforesaid party of the second part then read said statement from the party of the first part to the party of the third part, which I translated to "you clone, you die."

After the session, there was a "Q" period in contrast to the "Q&A" session you might expect. Generally the answers were along the line of "That's a good question, so I can't answer it."

I'm sure valid legal reasons for all this bunk exist just as they probably do for not calling P5 the '586 (yes, Intel is really having a "name that chip" contest). The question is whether a sanitized presentation is better than no presentation at all.

YOU'VE GOT SOME NERVE

One of the most interesting talks was "Silicon-based Nerve Interfaces" by Gregory Kovacs of Stanford University. Forget the mice and bit-mapped displays, now we're talking *human* interfaces.

This work is being funded by the VA to serve an obvious and deserving real-world application-improving prosthesis technology for amputees. The idea is to connect electrically with the nervous system, allowing both control of and, very important, feedback from an artificial limb.

The passive neural interface basically consists of some multiplexer/amplifier logic and a piece of silicon with a 32 x 32 array of holes through which the nerves pass. The size of the holes turned out to be critical. Too small and the nerve will fail to regenerate through the hole well, too large and isolating the nerves of interest becomes difficult. Doctors implant the device using a coupler that somewhat resembles those used for fiber-optic connections (though much smaller since the neural interface chip itself is only about 50 µm x 50 µm).

Effectively, Kovacs and others are trying to make the equivalent of a break-out box for the *human bus*. The long-term implications of this type of research are quite profound indeed.

Have a math test today? No sweat, just plug a *Pointdexter 200 math coprocessor* into your forehead.

Big track meet coming up? Good thing your *Zronman Floating Foot Accelerator* arrived. One in each hip should do the trick, directly controlling your leg muscles. You can even dial in the amount of "boost," taking care to insert enough wait states lest you "crash."

Nervous about that presentation to the board of directors? Not a problem if you preprogram your *Dog & Pony Pitch-giver IC* the night before. The latest model even has a *Random Joke Generator* to make sure you leave 'em laughing.

Kind of neat and kind of scary. Fortunately or unfortunately, depending on your point of view, we're a long way from this brave new world. It may be possible to grab or inject a few interesting signals into a nerve bundle or two, but integration with the higher level brain is questionable because the intelligence is "distributed" and not conveniently brought to one "edge connector."

Furthermore, healthy people may hesitate because, at least for now, installing a nerve interface requires cutting a perfectly good nerve.

Any volunteers? ❑

Tom Cantrell holds a B.S. and M.B.A. from UCLA and has been in Silicon Valley for more than ten years working on chip, board, and systems design and marketing. He can be reached at (510) 657-0264 or by fax at (510) 657-5441.

SOURCES

Digital Equipment Corporation
146 Main St.
Maynard, MA 01754
(SOS) 493-5 111

AT&T Bell Labs
1247 S. Cedar Crest Blvd.
Allentown, PA 18103

Intel Corporation
3065 Bowers Ave.
Santa Clara, CA 9595 1
(408) 987-8080

Gregory Kovacs
Center for Integrated Systems
Dept. of Electrical Engineering
CIS 130
Stanford, CA 94305

I R S

425 Very Useful
426 Moderately Useful
427 Not Useful

PRACTICAL ALGORITHMS

John Dybowski

Denominations of Time

Time marches on, and nothing knows that better than the microprocessor in your latest project. There are many techniques for dealing with time; which one you use depends on your hardware.

Microcontrollers are asked to perform many diverse and at times unexpected functions. Consider the operations performed by some of the contrivances driven by embedded controllers. Unlike most general-purpose computers, these instruments operate and dispense the consequences of their labor in the domain of real time for the most part. Because of this fact, many of us take for granted modern microcontroller elements such as fast interrupt servicing, good bit-manipulation capabilities, and integrated-timing subsystems. These attributes are especially important when operating on small chunks of time where fast operation in general and fast response to real-time events in particular are mandatory. They exist in order to yield the required precision in the time measurements performed because the controller must be capable of literally running circles around the event to be measured.

Another side to this story centers on time as a human being perceives and counts it. Perhaps contrary to popular logic, operating on very small repetitive events falls more in line with our controllers' counting and processing capabilities than does dealing with what we have so little trouble coping with: the larger denominations dealt by real-time clocks. I'll be discussing the time of real-time clocks, but for now let me take a look at the short end of it.

THE CONTROLLER HEARTBEAT

One of the most fundamental and mundane timekeeping chores a microcontroller performs is the generation of a timebase to function as a point of reference for all of its recurrent operations. Generally, a microcontroller achieves this step by running a timer at a frequency that activates an interrupt service routine (ISR) on expiration, which acts as the system heartbeat for all time-critical activities. Depending on the time interval required for this interrupt and the timer's capabilities, you can maintain the free-running nature of this timer by either operating the timer in autoreload mode or performing the reload function via software from within the timer interrupt handler.

A periodic timer can do all kinds of useful things with a minimum of software overhead. For example, it can count response and abort times for communications protocols, schedule events such as keyboard and sense point scanning, count time delays, and debounce switches.

I have yet to encounter an application that can perform adequately without a solid interrupt-driven timebase. Often, they can tolerate the absence of other system peripherals if a timer is available. For example, you can use your timer as a reference to create a software real-time clock or, with a little careful coding, perform functions like background serial communications using a timer interrupt without resorting to a hardware UART.

PERIODIC EVENTS

Often, you have to measure accurately events that exist in time, such as duty cycle, velocity, or the frequency of a pulse train. Looking at what these measurements constitute, you can see they all involve the measurement of rudimentary functions, such as pulse widths and periods. These measurements can be as simple or as complex as you care or can afford to make them. Before looking at some details, I'll first discuss a few different ways you can use controllers to count time.

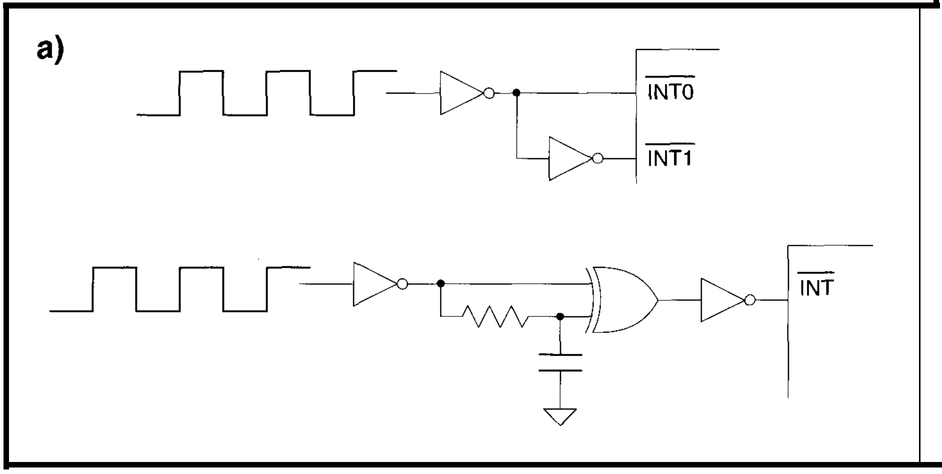
When the application can tolerate dedicating a controller to the task of time measurement, the simplest way to apply one is to shut down all the interrupts and implement a dedicated timing loop in software. You can use this approach only when all other system operations can be put aside for the duration of the measurement.

The lead-in to the measurement loop consists of clearing the timer register and setting up some miscellaneous variables, such as pointers and counters, because the process normally consists of collecting a number of samples for later manipulation. Now, the program loop increments a register and watches an input bit for a change of state. Once it detects a transition on the input signal, it stores the count and starts a new measurement. This process continues until the desired number of samples is collected.

The disadvantage to this approach is that the actual time measurement consumes all of the CPU horsepower to the exclusion of any other processing tasks. Worse, the timer resolution degrades to the time required to execute the measurement loop, which consists of multiple instruction steps. However, if you're stuck using some of the more primitive controllers, this method may be your only choice.

A better approach is to use one of the controller's on-chip timers to time the event. Here, the timer is initially zeroed and the controller again sits on the bit waiting for a change to occur. As before, the timer count is recorded when a change of state is detected before the timer is reinitialized and another sample can be taken. If you're using a 16-bit counter on an 8-bit controller, which is frequently the case, you will usually stop the timer before it is read to avoid overflow errors between the low and high timer bytes. You will want to ensure that the bytes read from the timer belong together.

You can attain better resolution with this method because the timer can be clocked at a much higher rate than what is attainable using the software timer approach. If the application can tolerate dedicating all of the controller's resources to the measurement process, this approach isn't all that bad because you can avoid problems with interrupt response latency altogether, provided the controller has good bit-manipulation capabilities and can respond to the transition quickly. You can obtain very good response times using controllers with built-in bit-branching functions. Here, you can just sit on a jump-bit-type instruction and fall through immediately when the state changes. However, leaving yourself an escape path is smart in case the expected transition never occurs. To provide this path, enable the overflow



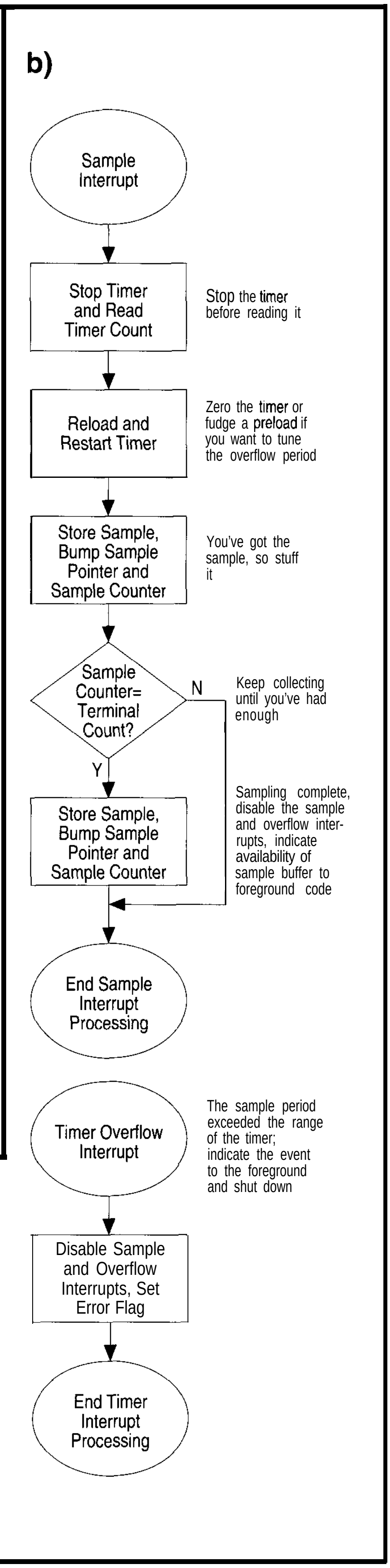

b)
Sample Interrupt
Stop Timer and Read Timer Count
Stop the timer before reading it
Reload and Restart Timer
Zero the timer or fudge a preload if you want to tune the overflow period
Store Sample, Bump Sample Pointer and Sample Counter
You've got the sample, so stuff it
Sample Counter= Terminal Count?
N
Keep collecting until you've had enough
Y
Sampling complete, disable the sample and overflow interrupts, indicate availability of sample buffer to foreground code
Store Sample, Bump Sample Pointer and Sample Counter
End Sample Interrupt Processing
Timer Overflow Interrupt
The sample period exceeded the range of the timer; indicate the event to the foreground and shut down
Disable Sample and Overflow Interrupts, Set Error Flag
End Timer Interrupt Processing

Figure l-a) When you *need an* interrupt on *both rising and falling edges, a sing/e, unmodified interrupt input just won't cut if. A pair of interrupt inputs does the trick, as does a little extra circuitry. In the second circuit, each edge-whether rising or falling-will generafe a very shorf pulse, the durafion of which depends on the RC fime constant. b) On each interrupt, a counter is incremented and compared with a terminal count. When the terminal count is reached, the timer has timed out* interrupt to yank you out of the sample loop when the timer wraps around. You can tune this overflow time by using a preload value when initializing the timer instead of starting the count at zero, but you'll have to make corrections for this value when you get around to processing the samples.

You can free up some processing power for other tasks by detecting the bit transition under interrupt control. Initiate an ISR in response to a signal transition that reads and stores the timer value, then reinitialize the timer. As before, adding overflow detection by enabling the overflow interrupt function is easy. You can use this feature to mask the transition interrupt and terminate sampling should the pulse train cease.

Using interrupt-driven sampling usually works well, provided the controller doesn't carry too much burden of overhead in its interrupt controller. Controllers such as the 8051 with its two-level interrupt priority do particularly well in this regard, whereas parts like the 80 188 can be real hogs unless you're willing to relegate this function to the nonmaskable interrupt that bypasses all the mumbo-jumbo of the interrupt control unit.

The significant thing here is the dependence of the measurement's accuracy on the repeatable timely response to the interrupting event, because the timer is actually read from the ISR. Most important of all, make sure that the interrupt response time is consistent. In this case, even if the interrupt response time is long, all of the interrupting events will be subject to the same delay, and the delays will tend to offset each other. Figure 1 shows some ways to interface external signals to the controller's interrupt pins.

Before looking at a more sophisticated system for dealing with time measurements, I'd like to mention a technique that you can use if you find your measurement task a little beyond the capacity of your present controller. Rather than using a timer, use a shift register to record the patterns of ones and zeros, referenced to a shift clock generated by an external oscillator.

What you want is an 8-bit shift register with tristate parallel outputs, a built-in output latch, and a reset function. Tie the outputs directly to the controller's data bus and generate the output control signal by gating the controller's read strobe with a chip select. Use the reset pin to hold off sampling until you are ready to start collecting data. When you start sampling, count the oscillator's clocks with some additional hardware or use the controller's timer in count mode to interrupt the controller and to issue a latch pulse every eight clocks. You must then read the shift register's output latch before eight more clocks occur or lose your sample. Once you've acquired the samples, count ones and zeros in the sample buffer to convert the bit pattern to durations for the high and low states.

This method works when you have to handle two or more different incoming bit streams and the controller's existing timer architecture does not permit the direct reading of the on-chip timer without the need for stopping it. It's all pretty cut and dry because you know that the interrupts will be coming at fixed intervals. You don't have to deal with the random interrupts that would occur when sampling the transitions themselves.

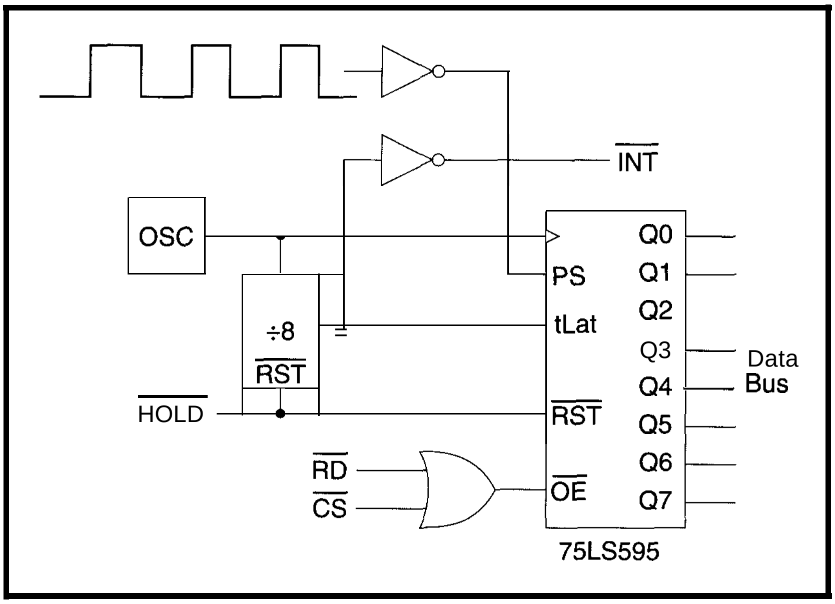


Figure Z-A simple shift register can be used to accumulate timer ticks, reducing the processor overhead by generating fewer interrupts.

Using this technique, you may be able to squeeze the controller to get the extra performance needed. The penalties of going this way are the increased memory requirement to store the samples and the need for the subsequent conversion, but sometimes you just want to stick with a $2 processor. Figure 2 shows the general idea behind this approach.

CAPTURING TIME

Modern microcontrollers address the need for accurate time measurements requiring a minimum of CPU intervention by implementing a timer capture system. In general, such a system is composed of a prescaler, a free-running timer representing physical time 16 bits wide, and multiple independent capture functions that record the time at which a transition is detected on a control pin. These capture functions all operate on the same master timer, but in effect provide the user with as many timers as there are capture functions.

Usually programmable to be either rising, falling, or both edges, a transition on the capture pin transfers the time count to the capture register and can also generate an interrupt if the function is enabled. A timer overflow that occurs when the timer counts from FFFFh to 0000h sets the overflow flag and allows software to extend the timing capability beyond the 16-bit limitation of the master counter. As with the capture event flag, the overflow flag can be interrogated via software or cause an interrupt to occur if it is not masked.

Once times for successive capture events on an incoming signal have been recorded, software can determine the duration of the occurrence by subtracting the first captured count value from the second. This procedure is straightforward if the duration between captures is less than the full counter overflow period. Note that no special considerations exist if the time periods being measured are known to be less than the time between successive overflows. That is to say, you can use simple 16-bit arithmetic to determine the time duration even if an overflow, from FFFFh to 0000h, occurs within the time period; the underflow from 16-bit arithmetic behaves just like the overflow from a 16-bit counter. If the events to be measured exceed the full overflow period, software will have to keep track of overflows to extend the counter's range, which tends to complicate matters significantly, especially when dealing with captures and overflows on an interrupt-driven basis.

The capture system is a big improvement over the other methods I have described because you don't have to stop the timer in order to read it. However, most important is that the time at which the event occurred is saved. Although the software may take an undetermined amount of time to respond to the event, it can determine exactly when the event occurred. Another advantage is that, because all capture events are referenced to the same free-running timer, you can determine timing relationships among various input signals in a straightforward manner.

The value read from the capture register corresponds to the most recent edge seen at the pin because the latching action of the capture function occurs every time an edge is detected. This agreement could mean that the value in the capture register may not be the value that caused the capture flag to be set in the first place if the edges occur more rapidly than the controller can process them. In some cases, as with unbuffered switch chatter, you can handle these undesirable extra captures by inhibiting further captures with software until after the current capture has been processed.

PERIODS AND PULSE WIDTHS

To measure a period, time the interval between two edges of the same polarity. The period can be converted to a frequency by obtaining the period's reciprocal. A better method would be to count pulses for a known period of time. The count at the end of the sample period equals the sample time times the frequency. From this information, you can derive the frequency as *frequency = number of cycles / sample time.* An alternate method for calculating frequency would be to accumulate a sample time for a known number of cycles; as before, *frequency = number of cycles / sample time.*

Measuring a pulse width is almost identical to measuring the period except that the interval between two

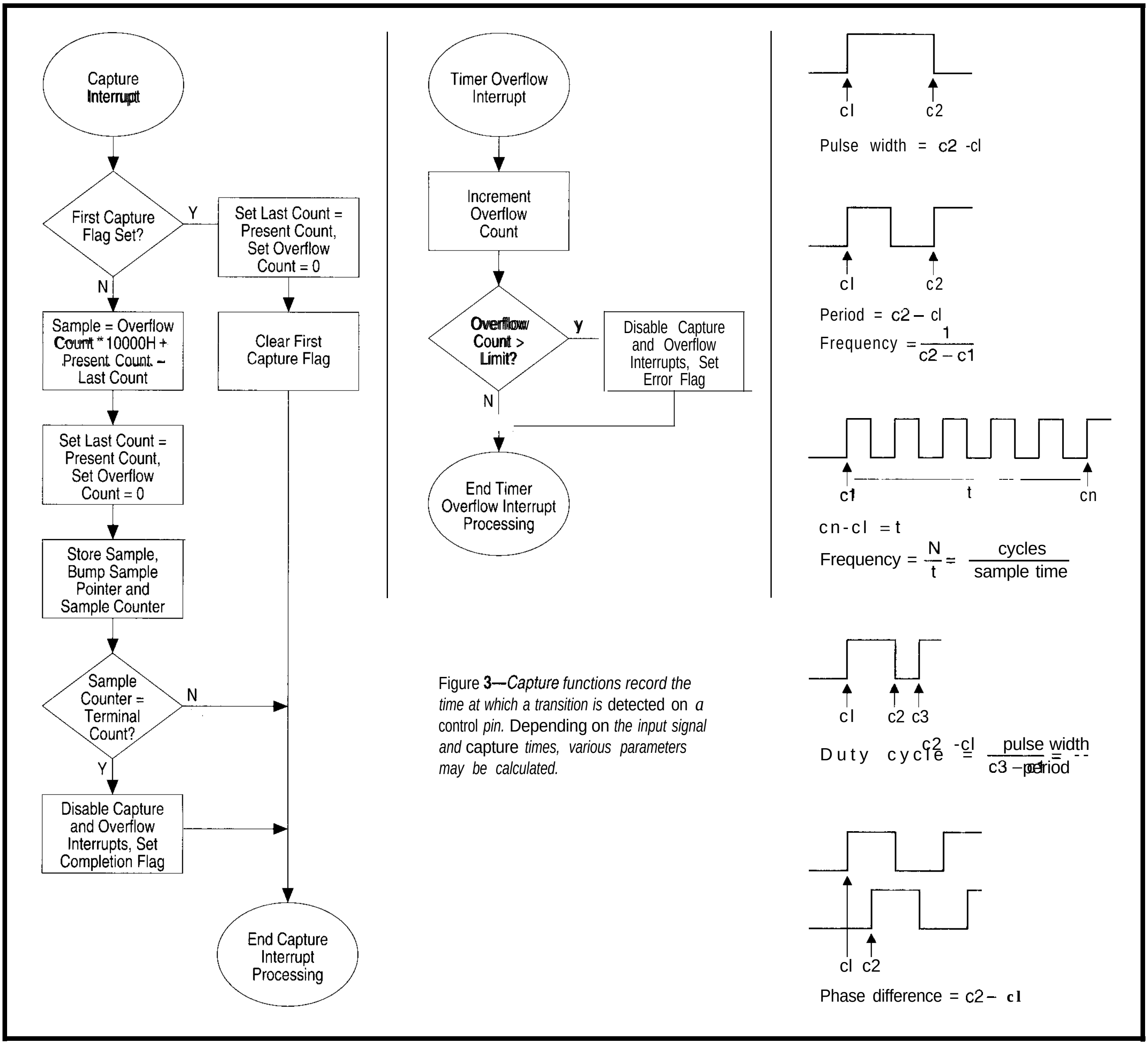

Capture Interrupt
First Capture Flag Set?
Y
N
Set Last Count = Present Count, Set Overflow Count = 0
Sample = Overflow Count * 10000H + Present Count - Last Count
Clear First Capture Flag
Set Last Count = Present Count, Set Overflow Count = 0
Store Sample, Bump Sample Pointer and Sample Counter
Sample Counter = Terminal Count?
N
Y
Disable Capture and Overflow Interrupts, Set Completion Flag
End Capture Interrupt Processing
Timer Overflow Interrupt
Increment Overflow Count
Overflow Count > Limit?
y
N
Disable Capture and Overflow Interrupts, Set Error Flag
End Timer Overflow Interrupt Processing
cl
c2
Pulse width = c2 -cl
cl
c2
Period = c2 – cl
Frequency = 1 / c2 – c1
c1
t
cn
cn-cl = t
Frequency = N / t = cycles / sample time
cl
c2 c3
Duty cycle = c2 -cl / c3 – c1 = pulse width / period
cl c2
Phase difference = c2 – cl

Figure 3—*Capture functions record the time at which a transition is detected on a control pin. Depending on the input signal and capture times, various parameters may be calculated.*

edges of opposite polarity is measured. Using a combination of period and pulse width measurements, you can calculate the signal's duty cycle as *duty cycle = pulse width /period.*

Finally, you can measure phase differences by timing the duration between the edges of the polarity of interest for two signals. Figure 3 illustrates some measurement techniques based on a timer capture system.

LINEAR TIME

Viewing the phenomenon of time as a cyclical happening is easy to do. This view is also justifiable when working with small components of time because many of the elements that you work with are repetitive in character. On a more substantial scale, this recurrence seems to abide as well. First morning arrives, then evening, then morning again. You reach the end of one week, then a new week begins. However, at some point you've been forced to realize that time is in fact a linear entity.

You are moving from the past into the future. Irrefutably. Unmercifully. I'm sorry to remind you if you were comfortably forgetting that time marches forward (although perhaps it is something worth keeping in mind), but denying it can prove troublesome, particularly when working with controllers in applications that must operate in time as you perceive it. Next month, my column will have a new name, but I'll continue with a discussion of the currency of real-time clocks.

John Dybowski is an engineer involved in the design and manufacture of hardware and software for industrial data collection und communications equipment.

IRS

428 Very Useful
429 Moderately Useful
430 Not Useful

CONNECTIME

conducted by Ken Davidson

The Circuit Cellar BBS
300/1 20012400 bps, 24 hours/7 days a week
(203) 871-1988—Four incoming lines
Vernon, Connecticut

LEDs are everywhere. Take a good look around some time and notice just how many places they're used. LED technology continues to improve, with new colors and brighter devices showing up every day. In our first thread, we fake a look at some of the latest developments.

Following LEDs, we look at some issues related to adapting or building a high-frequency frequency counter.

Msg#:60794
From: PAUL PETERSEN To: ALL USERS

The latest issue of ECN arrived today and two of the items caught my eye:

MarkTech International is offering 1500-mcd and 2500-mcd LEDs in yellow and orange; 20-mA current. That's incredibly bright! Probably blinding. An industry breakthrough they say; I believe it.

LEDtronics has a rainbow LED. Create any color of the rainbow, they say, with this 3-in-1 that combines red, green, and blue in a 4-lead T1-3/4 package. No price shown.

Msg#:60928
From: WILLIAM VONNOVAK To: PAUL PETERSEN

Radio Shack has a 5000-mcd red LED that's good to play with. It's blinding, but the beam is narrow. They make good brake light replacements.

The LEDtronics all-color LED sounds interesting, but I think they still have problems with the intensity of the blue chip (max of -6 mcd?). I like the idea of white LEDs.

Msg#:61011
From: JAKE MENDELSSOHN To: PAUL PETERSEN

AND has a 13,000-mcd LED {AND190CRP} that they are selling through Allied Electronics {800/433-5700} for less then $5.00 each. It is so bright, it hurts.

Msg#:61019
From: BRIAN KRAUT To: JAKE MENDELSSOHN

Is there really a need for a 13,000-mcd LED, or are the just trying to see who can make the brightest one?

Msg#:61084
From: WILLIAM VONNOVAK To: BRIAN KRAUT

What do you mean, is there really a need for 13-cd LEDs? How about:

- solid-state traffic lights (you'd only need one set per intersection instead of two if they never burn out)
- super efficient area lighting and emergency lighting
- brake lights

DC/DC couplers (LED/photovoltaic cell)
large-area video screens

I want 50-cd LEDs! In red, green, yellow,and blue. (OK, maybe I can wait on the blue.)

Msg#:61105
From: PAUL PETERSEN To: WILLIAM VONNOVAK

Your comment about super-efficient LEDs piqued my interest. Do these really bright LEDs get hot? Heat sinks and all that stuff? How much current is typical for one of them?

Msg#:61169
From: WILLIAM VONNOVAK To: PAUL PETERSEN

It's odd-they all seem to use 2030 mA. Some have slightly higher forward drops, but not enough to account for all the brightness differences. None of them (that I've seen) need heat sinks.

I used to think LEDs were around 70% efficient. Then they started coming out with these 13-cd LEDs, and they're definitely not 300% efficient. So I'm not sure what the actual efficiencies are. One thing to keep in mind is that brightness is not the same as total light output-it depends greatly on the lens, whether it's diffused or not, and so forth.

Msg#:61393
From: JAKE MENDELSSOHN To: PAUL PETERSEN

The LED is rated to emit 13,000 mcd with a 20-mA current. However that is assuming a continuous current. If you pulse the current, you can put a much greater load through the LED and get a much greater light output. I am using these LEDs to make a strobe light and I put one full amp through them (for only 1 ms, of course). The strobe is NOT as bright as a zenon flash tube, but it is bright.

Msg#:61402
From: BRIAN KRAUT To: JAKE MENDELSSOHN

Never thought of using one as a strobe before. Except for the brightness, it sounds like an LED would be the ideal strobe. Easy to control, simple circuitry, and an incredibly fast maximum strobe rate. I've been planing on building a strobe and I'm glad you gave me this idea before I wasted my time with a conventional strobe. Thanks. By the way, how bright is 13,000 mcd in comparison to incandescent bulbs?

Msg#:61 a79
From: JAKE MENDELSSOHN To: BRIAN KRAUT

Since I started this strobe discussion, let me make a few points very clear.

1. These ultra bright LEDs are extremely bright for LEDs but they are NOT anywhere nearly as bright as your standard zenon strobe tube. As a small strobe light that you aim at the target at close range (<1 foot) they are great, but they will never function as disco lights.

2. The next question is how bright are they and how do they compare to incandescent bulbs? The LEDs have their output given in millicandela, most incandescent bulbs have the output listed in lumens, and the output of IR LEDs is given in milliwatts. How do all these rating compare? I didn't know the answer and after making a few calls, I found out that a lot of other people also didn't know the answer. After doing some research, here is what I came up with:

The facts:
1 candela = 1 international candle = 1 lumen/steradian
1 sphere = 4π steradians
1 lumen = 0.0015 watts
1 steradian = 3282 square degrees

The ultra bright (#190) LED from AND emits 13,000 millicandela (13 candela) at 20 mA of current. The LED is rated for up to 50 mA for continuous operation and the chart in the data book shows that the output at 50 mA will be more than doubled. Therefore, let's assume that the 50-mA output is 30 candela.

A 40-watt incandescent light bulb emits 480 lumens (as listed on the box). This is 480 lumens radiated in all directions (i.e., a sphere). Divide 480 lumens/sphere by 4π steradians/sphere yields 38 lumens/steradian. Since 1 candela = 1 lumen/steradian, this means that a 40-watt bulb is a 38-candela source.

An IR LED (CLED 155) from Senisys emits 10 milliwatts (0.01 watts) at 50 mA of current. From a chart in the data book, we see that this LED has a maximum continuous current of 100 mA and will yield 1.8 times the 50-mA output or, in other words, 0.018 watts. Divide this by 0.0015 watts/lumen yields an output of 12 lumens. This output is spread over a cone with a 20-degree radius (see data book). The base area of this cone is πR^2 and thus 1256 square degrees. Therefore, the output of the IR LED is 12 lumens for the 1256 square degrees. Dividing this by 3282 square degrees/steradian yields 3 1 lumens/steradian, which is 3 1 candelas.

Conclusions: Although there is a difference between brightness (candela) and total light output (lumens), because the LEDs emit the light in a narrow cone and the incandescent bulb emits its light in all directions, it does seem that the AND #190 red LED, the Senisys #155 IR LED, and a 40-watt incandescent bulb all appear to be equally bright light sources.

By the way, the above analysis is based on continuous operation, but both the LEDs can be driven with much higher currents if they are pulsed. For example, the data book says that the Senisys IR LED can draw up to three amps if pulsed, and I have driven the AND red LED with one amp. There is no data given on how much brighter a one-amp pulse is then the normal 20-mA current, but it must be significant.

The best way to avoid reinventing the wheel is to adapt an existing design to your special requirements. Sometimes, though, it can be more trouble than it's worth. Here, we look at some alternatives.

Msg#:58070
From: GUY RESH To: ALL USERS

I'm trying to figure out a way to "inexpensively" make a frequency counter board that I can interface to either a 68HC 11 MCU or the parallel port of a PC. I'd like to use a full 32-bit word, which should allow more than enough precision (10 MHz or thereabouts will do, but I might as well use four 8-bit bytes). What I was wondering was if anyone knew of a chip (or group of chips) that could do this as simply as possible. I've got the concepts down, but since the projects I've seen display the results on an LCD display (and don't display all digits at the higher frequencies], I thought it might be even easier (and cheaper) to simply have a group of counters somehow connected to a set of latches that I could multiplex with a 4-way decoder of sorts to "select the proper byte" and roll out the current value assuming a particular latch bit was set as "available and waiting." Sampling would only need to occur every second. Any part numbers for the above chips? My '86 edition of IC Master isn't much help, and I want to use as few chips as possible.

Msg#:58236
From: PELLERVO **KASKINEN** To: GUY RESH

This may not be quite what you are looking for in the area of interfacing, but have you taken a look at the Harris (former Intersil) parts ICM7208 and 7207? The first one is a 7-digit counter and the second one is a matching timebase for the counter. I think the output from the counter is in seven-segment or other multiplexed form, hence not necessarily easy to tie in with a PC..

Msg#:58247
From: GUY RESH To: PELLERVO KASKINEN

Yep, I've got the specs for the entire line of Harris/ Intersil counters, which do work only with segmented (BCD) displays. Trying to convert one of those BCD output (multiplexed at that) versus simply using a few fast 4-bit or 12-bit counters with a timed latch/reset is more than I want to mess with so I've opted for the latter. Chip count should be manageable, but locating a **l-Hz** time source with a decent resolution is my latest dilemma, though I've got a lot of sources to check into yet.

For some reason, I was thinking that I was going to get l-Hz accuracy at 20+ MHz and still keep the project within my budget. I've woken up since then and relaxed my requirements into a more realistic design. Thanks for the help.

Msg#:58493
From: PELLERVO KASKINEN To: GUY RESH

Actually, converting the 7-segment format to something else is not that impossible. I think it is one of the applications I saw as an example for using (and programming) PALs. Does not take too much input or output pins like some of my needs always have appeared to do. But I have to back off the Intersil suggestion for another reason: You mention 20 MHz. That clearly would require a different front end. If I remember correctly (don't have the data book at home), the CMOS parts in question can handle some 5 to 10 MHz.

As to the current sensing, a current transformer is the easy and cheap way out, unless you indeed need more than about 1000 amperes. Beyond that they tend to become a bit more esoteric (read expensive). The explanation is the number of wire turns in the secondary-you simply run out of space for the winding in all normal iron cores.

Msg#:58545
From: GUY RESH To: PELLERVO KASKINEN

Yeah, the hi-freq is the limiting factor since I want an "all-in-one" lab instrument that isn't only a DSO, but also frequency counter, DMM _and_ current probe. It's no wonder HP gets the $$$ they do for their high-end stuff; now if only someone could do it "all" for under $2000 (100 MHz and all, though 50 will do fine, I guess). The PAL approach sounds like a neat idea, though I'd have to get someone to burn it for me since I've only got an EPROM burner from days past.

Msg#:58423
From: DAVE TWEED To: GUY RESH

Some time ago, I had the idea of building a "universal" counter using a single-chip micro. I tend to use the 8051 and friends, but the 'HCl 1 should work just as well. The main idea is this: Do as much of the work as possible in software. The microprocessor should be perfectly capable of counting at up to 10 kHz or so all by itself, so all that is required is a pair of external prescalers and some gating logic.

The idea of a "universal" counter is simply that you have two counter chains that are gated on and off together, then you divide the number from one chain by the number from the other chain and display the result. If you want to measure frequency, you feed a known clock into one input and the unknown into the other input. The gate time is not particularly critical-longer times give you more resolution, but the accuracy of the reading is determined only by the accuracy of your "known" frequency.

So, the plan is this: take two 16-bit counters; each one is fed from an external jack and they are gated by a common

CONNECTIME

control line from the CPU. Another control line from the CPU clears both counters. Connect the MSB of each counter to an external interrupt line on the CPU (the 805 1 has two external interrupt inputs, which is convenient) so that each time it overflows, a software counter inside the CPU can be incremented. In other words, you have two 32-bit (or more) counter chains, the first 16 bits of each are implemented in hardware and the remaining bits are implemented in software. The gate time can be controlled by the CPU's on-board timer.

The measurement cycle proceeds as follows: the CPU clears both counters (including the overflow counts), then turns on the gate. Interrupt routines handle the overflow counts until the timer indicates that the gate should be closed. The CPU closes the gate, then reads the values out of each of the hardware counters, combines them with the overflow counts, does the division and either displays the result or makes it available to a PC through some kind of interface. The 8051 has a hardware serial port that would work well for this.

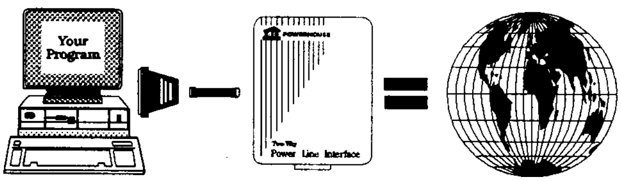
We invite you call the Circuit Cellar BBS and exchange messages and files with other Circuit Cellar readers. It is available 24 hours a day and may be reached at (203) 871-1988. Set your modem for 8 data bits, 1 stop bit, no parity, and 300, 1200, or 2400 bps.

ARTICLE SOFTWARE

Software for the articles in this and past issues of *The Computer Applications Journal* may be downloaded from the Circuit Cellar BBS free of charge. For those unable to download files, the software is also available on one 360K IBM PC-format disk for only $12.

To order Software on Disk, send check or money order to: The Computer Applications Journal, Software On Disk, P.O. Box 772, Vernon, CT 06066, or use your VISA or Mastercard and call (203) 875-2199. Be sure to specify the issue number of each disk you order. Please add $3 for shipping outside the U.S.

431 Very Useful 432 Moderately Useful 433 Not Useful



STEVE'S OWN INK

Evolution

Want to know what the worst-kept secret in the publishing business has been for about the last three months? The *Computer Applications* Journal is increasing from 6 times to 12 times per year. The first public indication was supposed to have been the subscription cards in the last issue. Because of the lead times involved in subscription servicing, the only way to keep subscribers from being caught in a "gray zone," similar to buying retail the day before a sale, was to change subscription terms earlier than the actual physical metamorphosis. To better understand how this affects your subscription, Publisher Dan Rodrigues has a more detailed explanation on page 96.

Apparently the only ones keeping the secret was us, however. Recent communications with authors, advertisers, and readers included statements like, "I heard you're going monthly," or, "Jumping off the cliff, huh?"

After careful consideration, I can only ascertain that the blabbermouth must have been me. A few times in the last 18 months, during conversations on the BBS, I might have suggested such a scenario. Recent surveys suggest that fully half of our readership frequent the Circuit Cellar BBS, so my secret answer was probably read by 50,000 people. Even so, I never said this month exactly, but success dictates specific evolution and it's apparently been obvious to more than just us.

So what does all this mean to you the reader? Well, if Ken and I thought we could get away with it we'd slip in six more home automation issues a year.

All kidding aside, the reality of providing 12 issues is that we'll present twice the technical information that we have up to now. Of course, more room also means that we can cover some more esoteric subjects and projects that couldn't fit in six issues. At the same time we guarantee that, as always, what we present will be relevant and perform as described.

Publishing monthly takes a special commitment from the staff. I can tell you from personal experience that presenting monthly projects is no easy feat. As tough as it might seem, Jeff "From the Bench" Bachiochi, Ed "Firmware Furnace" Nisley, John "Embedded Techniques" Dybowski, and Tom "Silicon Update" Cantrell will be expanding their presentations to every issue. Cost-effective embedded processors will still be covered in depth but we are also making an effort to significantly increase project coverage of new processors and 80x86 embedded applications and software.

If that isn't enough, we'll also be adding a new column on patents. Because of the entrepreneurial nature of our readership, there are always questions when designers sell their ideas or become manufacturers themselves. Our patent column will point out important patents that might affect your designs as well as illuminate issues involved in the patent process.

We keep thinking of new material to add every day. Of course, if the present is any indicator and you continue to support us as well as you have in the past, who knows what the next evolution will bring?

Steve