CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

MEASUREMENT & CONTROL

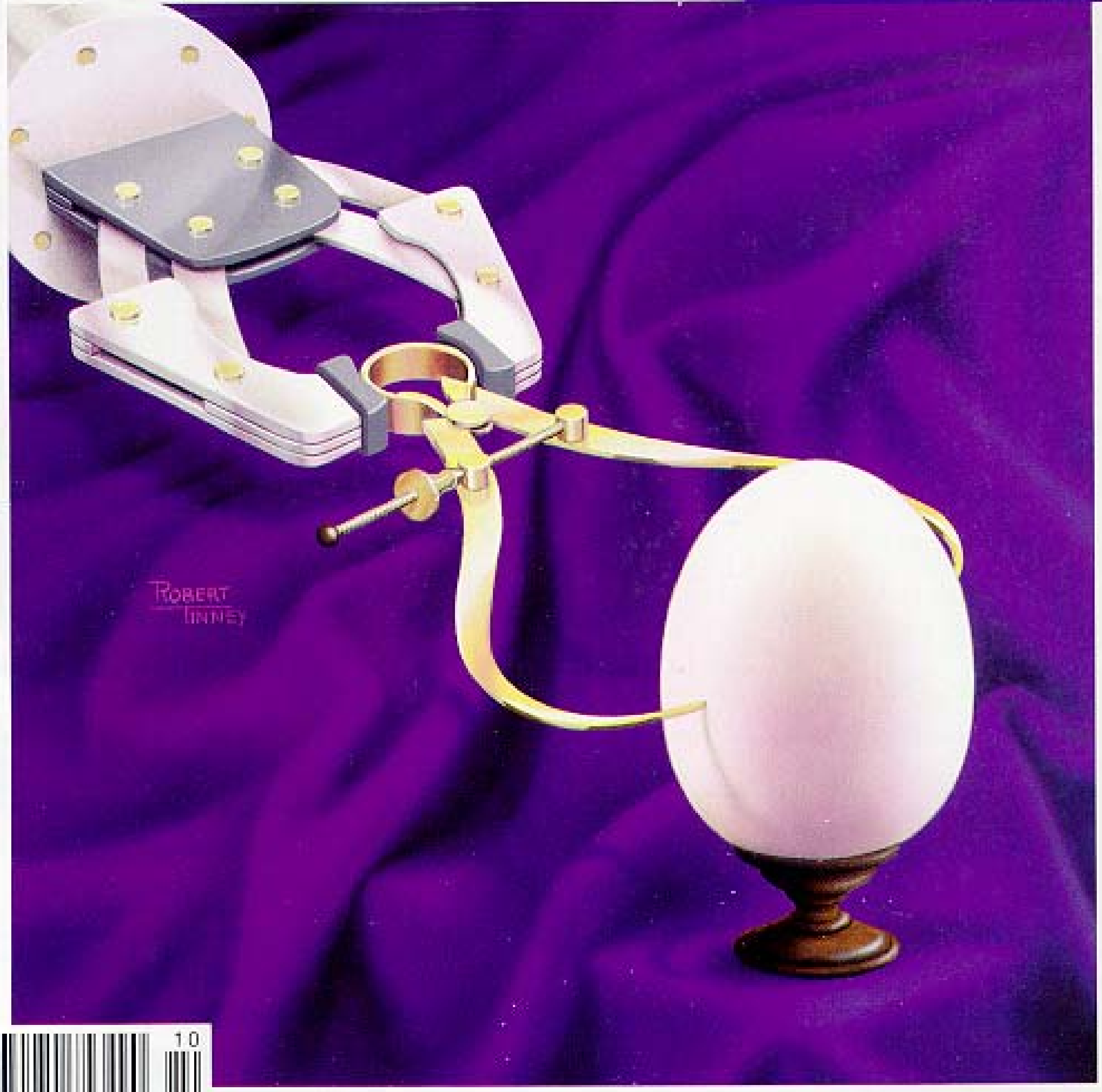
SPECIAL SECTION:
Embedded Graphics & Video

Time Domain Reflectometer

Overlay Text on Video

X-10 Interfacing

October/November, 1992 — Issue #29



$3.95 U.S.
$4.95 Canada

EDITOR'S INK

A Look at Next Year

We're approaching the end of another editorial year, and many of you are anxious to find out what we have in stare for the next year. Before I list the upcoming themes, let me take a moment to encourage you to look at the list with an eye toward writing an article. Does one of the themes come close to what you do in your everyday job? Perhaps you just finished a project that relates to one of the topics. If so, chances are you've learned a few tricks that would surely benefit your fellow readers.

Don't assume that just because you're comfortable with a "basic" concept, everyone else is just as comfortable. Write an article to show off what you know and everyone will benefit from your knowledge. For example, if you work with analog circuits all day long, your expertise can go a bng way in helping someone whose jcb doesn't extend mush past bits and bytes.

What's that? You've never written an article and wouldn't know where to begin? Nonsense. Bounce your ideas off us and we'll be happy to work with you to structure a welt-written article in no time. What's important is that you know ywr stuff and you take that first step by contacting us with your ideas. You may call me at (203) 875-2199, leave me a message on the Circuit Cellar BBS ([203] 871-1988), send me a fax ([203] 872-2204), or put your ideas on paper and send them by mail (4 Park St., Vernon, CT 06066).

Now for next y&s themes. I'll have some specific dates for you in the next issue, but I didn't want to put off telling you about the topics.

Home & Building Automation
Real-Tiie Programming
Communication
Graphics & Vi
Power Control & Conversion
Data Acquisition
Embedded Interfacing
Siinal Conditioning
Signal Processing
Measurement & Control
Programmable Devices
Embedded Control

Nothing catches your eye, but you have some other ideas? That's OK, too. There is often room in each issue for an article that doesn't strictly fit that issue's theme, so we still want to hear about those ideas.

BACK TO THE PRESENT

We're hard at work judging this year's Circuit Cellar Design Contest entries and we'll have the results in the next issue. In the meantime, we have two articles in thii issue written by past Design Contest winners. The MC68HC11-based two-dimensional sensor used with a geranium-planting robot won first place in the General Category last year and the Time Domain Reflectometer won first place in the Cost-Effective Category the year before.

Be sure to keep an eye out in future issues for our new "Design Contest Winner" logo that now marks an author as one of the best.

Ken

CIRCUIT CELLAR INK®

THE COMPUTER APPLICATIONS JOURNAL

FOUNDER/EDITORIAL DIRECTOR
Steve Ciarcia

MANAGING EDITOR
Ken Davidson

ASSOCIATE EDITOR
Lisa Nadile

ENGINEERING STAFF
Jeff Bachiochi & Ed Nisley

WEST COAST EDITOR
Tom Cantrell

CONTRIBUTING EDITOR
John Dybowski

NEW PRODUCTS EDITOR
Hatv Weiner

ART DIRECTOR
Lisa Ferry

STAFF RESEARCHERS:
Northeast
John Dybomki
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 (ISSN 0896-8985) k published bimonthly by Circuit Cellar Incorporated, 4 Park Street, Suite 20, Vernon, CT 06066 (203) 875-2751. Second-class postage paid at Vernon, CT and additional offices. One-year (6 issues) subscription ram USA a-d possessions $17.95, Canada/Mexico $21.95, all other countries $32.95. All subscription orders payable in U.S. funds only, via international postal money order or check drawn on U.S. bank. Direct subscription orders to Circuit Cellar INK, Subscriptions, P.O. Box 3050-C, Southeastern, PA 19398 or call (215) 630-1914.

POSTMASTER: Please send address changes b Circuit Cellar INK, Circulation Dept., P.O. Box 3050-C, Southeastern, PA 19398.

HAJAR ASSOCIATES NATIONAL ADVERTISING REPRESENTATIVES

NORTHEAST
Deb ra Andersen
(617) 769-8950
Fax: (617) 769-8982

MID-ATLANTIC
Barbara Best
(908) 741-7744
Fax: (908) 741-6623

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

MIDWEST
Nanette Traetow
(706) 789-3080
Fax: (706) 789-3082

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

Circuit Cellar BBS—24 Hrs. 300/1200/2400 bps, 8 bits, no parity, 1 stop bit, (203) 871-1988; 9600 bps 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.

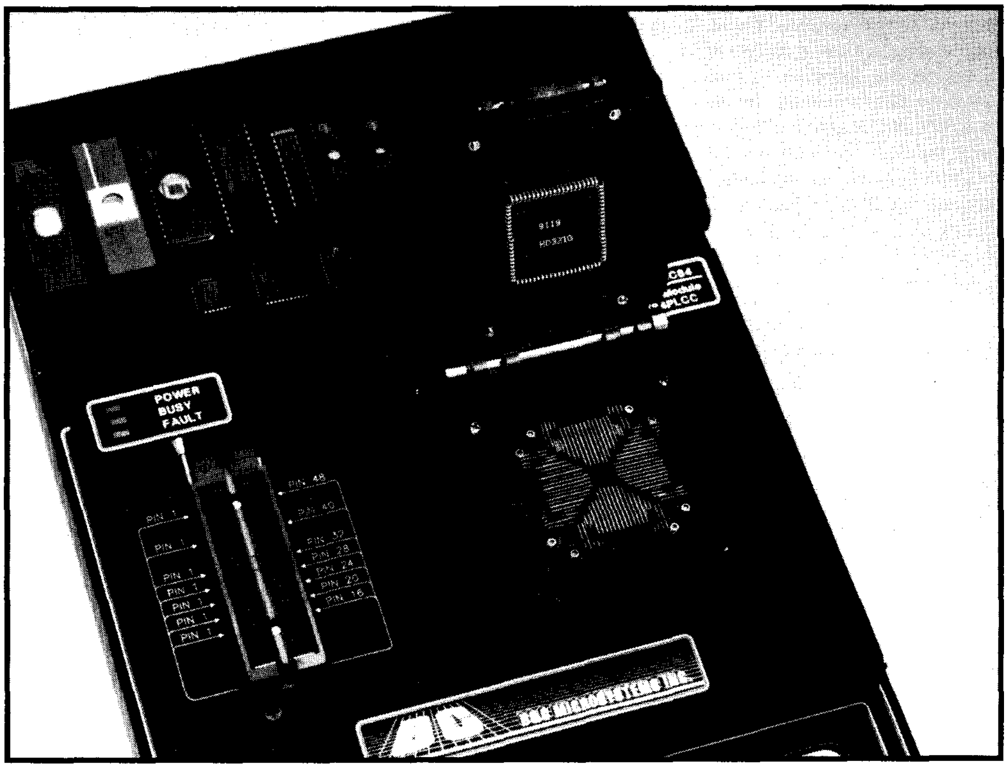
INSIDE ISSUE 29

SPECIAL SECTION Embedded Graphics & Video

NEW PRODUCT NEWS

Edited by Harv Weiner

UNIVERSAL DEVICE PROGRAMMER



The first device programmer in production offering a single universal PLCC socket has been announced by B&C Microsystems Inc. The new socket technology of the Proteus104 accommodates any PLCC device from 20 to 84 pins, reducing PC board traces and capacitance on the programmer header. A 48-pin ZIF socket is mounted alongside for programming 300- and 600-mil DIP devices.

The Protéus104 features fully overvoltage- and overcurrent-protected pin drivers, full digital and analog capabilities for all pin drivers, true system self-calibration of all voltage sources and full diagnostics of all pin drivers, a built-in controller and timer with 250-ns resolution, and state machine testing with rise and skew time of less than 10 ns.

The Protéus104 also features expandable pin driver boards, allowing upgrades to any level within its 24- to 104-pin range. An entry-level version with 24-pin drivers and a 40-pin ZIF socket is the base configuration. For production environments when programming MOS devices, gang (parallel) modules replace the universal module and provide higher throughput to meet manufacturing needs by providing eight or sixteen ZIF sockets. Also available is a memory card gang module (supporting PCMCIA 2.0 Standard] with sixteen sockets.

The programming and algorithm development environment is based on a single executable file with a built-in editor and compiler for fast software development and ease of use. Semiconductor manufacturers use a powerful Algorithm DEvelopment Language (ADEL) in the creation of new device algorithms. ADEL can also make changes to existing algorithms, when revisions are required, and can drastically reduce turn-around time for the user of the devices. An extension of ADEL allows the creation of special test algorithms independent of those officially published ones embedded in the Prottusl04 system.

Communication with any PC, XT, or AT is via a parallel printer port, which has been optimized for the fastest and most reliable performance. The speed limitations exhibited by PC bus-based programmers are absent The parallel port is searched by the programmer and the connection made automatically. All device library software is available through a Bulletin Board System or by floppy diskettes.

The Protéus104 is priced from $745 to $4995.

B&C Microsystems, Inc.
750 N. Pastoria Ave.
Sunnyvale, CA 94086
(408) 730-5511
Fax: (408) 730-5521

#500

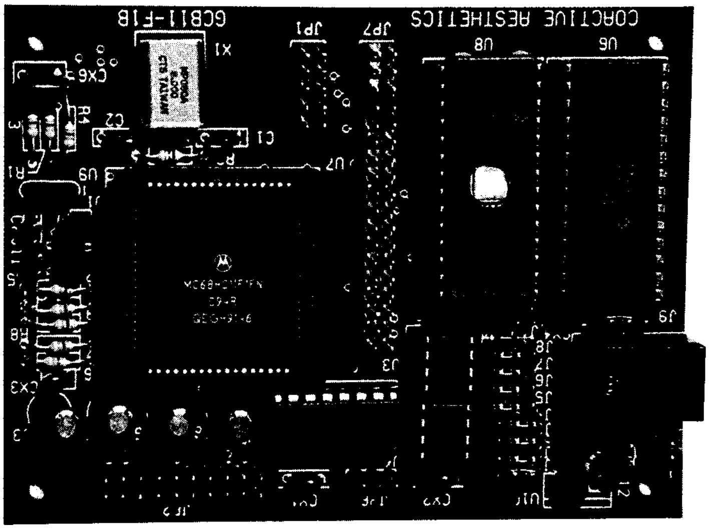
NEW PRODUCT NEWS

LOW-COST NETWORKED MICROCONTROLLER

The GCB11, an 8-bit networked microcontroller hardware and software package, has been announced by Coactive Aesthetics Inc. The 3" x 4" board, based on the popular Motorola MC68-HC 11 F 1 chip, includes 32K of static RAM, 32K of ROM and requires only a +5-volt supply.

The GCB1 1 includes 20 digital I/O lines and eight channels of 8-bit A/D conversion. Communication is provided by a standard RS-232 port and an RS-485 port for a multidrop network with up to 32 nodes. Additional 485 drivers are on board for connecting a PC using the RS-232 to the RS-485 network, without the need for additional circuitry or boards. The boards can be jumper configured as either a master or a slave on the network, and code can be downloaded and debugged across the network.

The GCB1 1 Package comes with a complete set of development tools and network and application libraries. The GNU C cross-compiler operates under PC-DOS and includes support for C++ and Objective C. The GNU Linker is also included. The ROM Monitor-Debugger allows program downloads in S-record format. Break points can be set and single step/step over execution is provided. Read and modify features are provided for memory, CPU registers, I/O registers, EEPROMS, and ports. An assembler and disassembler are also included.



Master-slave packet communications software provides rapid development of distributed control applications at various baud rates up to 115.2 kbps. Application utilities include a PWM motor driver library for transparent support of up to four DC motors, encoder feedback for speed and position control of up to three motors, input signal debounce library, and support for ADC sampling at regular intervals as a background task Source code for all software is provided (C and assembler]

The complete GCB1 1 package sells for $179.

Coactive Aesthetics, Inc.
P.O. Box #425967
San Francisco, CA 94142
(415) 626-5152
Fax: (415) 626-6320

#501

LOW-COST 8051 C PACKAGE

Franklin Software's new Engineer's Evaluation Kit, Release I, is a complete C programming package for 805 1 development at the price of a standard 805 1 assembler. The kit targets engineers and students who have tried earlier C products and found them either too weak and slow or too expensive. The kit allows you to experience the power and efficiency of an advanced C for 805 1 software development and maintenance.

Based on technology developed for Franklin's Professional Developer's Kit, the Evaluation Kit includes a C compiler, a macro assembler, a linker, a powerful symbolic debugger-simulator, C libraries, and utilities. The kit features fast compilation and highly optimized object code. Execution speed is comparable to the best in the industry, and performance is equal to a tightly coded assembler.

The Engineer's Evaluation Kit is an entry-level companion to, and fully upgradable to, Franklin's Professional Development Kit for the 805 1. Priced during introduction at $299 plus an optional support contract, the C kit is an affordable way for engineers to discover the advantages of a complete C environment for 8051 development.

Franklin Software, Inc.
888 Saratoga Ave., #2
San Jose, CA 95129
(800) 283-4080 Ext. 896
Fax: (468) 296-8061

#502

NEW PRODUCT NEWS

PROGRAMMABLE RELAY CONTROLLER PROVIDES ANALOG AND DIGITAL SUPPORT

A family of programmable relay controllers with both analog and digital I/O capabilities for low-end control and sensor interface applications has been introduced by DIP Inc. The Pico PLC units provide digital, analog, PWM, RS-232 I/O, and an auxiliary sensor power supply. Integrated programming and debugging support allow the PPLC to be programmed using a standard data terminal or PC. No specialized programming software is required.

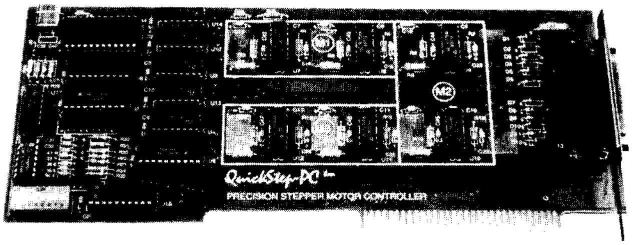
Extending the control language typically found in low-end PLC units with a full set of arithmetic and comparison operators, the PPLC products allow the user to intermix analog and digital operations using ladder-logic-formatted command strings. The unit may be put into a single step or display mode for interactive debugging. File transfers are supported to and from a PC programming unit using standard PC-based communications packages. EEPROM program storage provides for 256 program elements with an average execution time of 50 µs per element.

The PPLC will replace combinations of discrete relays, counters, timers, and comparators in applications such as production monitoring, intelligent sensor interfaces, front-end preprocessing for larger programmable controllers, sequence control, and low-end motion control systems requiring peripheral interfaces. A comprehensive programming guide cuts the learning time to less than two hours.

The base unit (PPLC) provides four inputs (120 VAC), four outputs (120 VAC, 2 amps), two analog channels (O-IO VDC), a PWM output (open collector), an RS-232 serial channel, and a sensor supply voltage of 12 VDC at 200 mA. The entire unit is packaged in a 2.75" x 3" DIN-rail-compatible case with captive screw terminals. A board-level version (PPLC-BRD) extends the I/O to eight inputs, eight outputs, and four analog inputs. Both units sell for $239 in single quantities.

DIP Industrial Products
P.O. Box #9550
Moreno Valley, CA 92552
(714) 924.1730
Fax: (714) 924-3359

#503



PC-BASED MOTION CONTROLLER

MicroKinetics Corporation has introduced Quickstep, a new stepper motor controller containing on-board translators and power drivers for up to three axes all on one card. It is designed for cost-sensitive applications that do not require very high stepping speeds and where the convenience of having the indexers and drivers on one card is important. The Quickstep plugs directly into any 8- or 16-bit ISA bus IBM PC or compatible, eliminating the need for an external enclosure.

Features include programmable acceleration and deceleration, automatic overtemperature protection, end-of-travel detection on all axes, two auxiliary outputs per card, and a shield-open interrupt. The Quickstep operates interactively; no uploading or downloading of programs is required. It provides keyboard control of such functions as Jog, Pause, Abort, and so forth, and drives motors requiring up to 0.9 amps at 12 VDC. A built-in timer assures motor speed consistency regardless of computer speed. The Quickstep requires only one 8-bit slot and features programmable address selection to allow coexistence with other cards that may otherwise cause address conflicts.

The software is very easy to incorporate into any application. The included subroutine libraries support C and QuickBasic, and feature linear and circular interpolation, ramping, keyboard interactive jog, and electronic gearing.

The QuickStep motor controller sells for $389 for a three-axes system. One-axis and two-axes systems are available for less.

MicroKinetics Corp.
1220 Kennestone Cir., Ste. J
Marietta, GA 30066
(404) 422-7845
Fax: (404) 422-7854

#504

NEW PRODUCT NEWS

HARD DRIVE ENCYCLOPEDIA

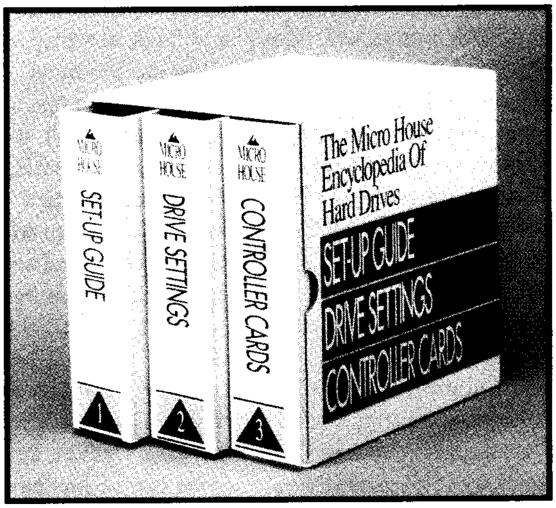
The "Micro House Encyclopedia of Hard Drives" is now available from Jensen Tools. This unique, three-volume support tool provides comprehensive technical help for the installation and upgrade or maintenance of multivendor hard drives. It contains information on hundreds of drives from Alps to Zebec, including many discontinued makes and models. The information is loose-leaf bound in three-ring binders to make page replacement and update easy, and comes with a fast, easy software version on DOS 5.25" diskettes.

Included are separate volumes on setup, controller cards, and drive settings. The information includes everything service professionals need to know about switch settings, cable locations, configuration parameters, power specifications, error codes, and interface basics for ST506/412, SCSI, ESDI, IDE, floppy, and more. Clear drawings detail switch settings and cable connections for both drives and controller cards. Also included are BIOS tables and a complete index of manufacturers with contact information.

The "Micro House Encyclopedia of Hard Drives" (Part #764B880) is featured in Jensen's Catalog Supplement D and costs $150.

Jensen Tools, Inc.
7815 S. 46th St. Phoenix, AZ 86044 . (602) 968-6241

#505



The $595 Solution
to 8051 System Development

PDK51

- *SIBFC-II microcontroller board (8052AH-BASIC CPU) with 16K RAM*
- *BASIC interpreter source on disk*
- *SIBFC-II hardware manual with schematic*
- *Monitor/debugger ROM plus manual*
- *Monitor/debugger hardcopy source listing*
- *UL listed 5V power supply*
- *Programming power supply adjustable from 4V to 26V*
- *KERMIT communications program*
- *SXA51 8051/8052 cross assembler*
- *BTK52 BASIC programmers toolkit with tutorial*
- *160 page BASIC Programming Manual by Systronics*
- *RS232 cable*
- *PDK51 tutorial*
- *Optional: aluminum case--$45, monitor/debugger source -$25, battery-backed real-time clock--$39.*
- *All components are manufactured to exacting standards and warranteed for one year.*

The PDK51is a fully integrated hardware, firmware, and software system designed to help you develop your products quickly and cost effectively.

All you need to use the PDK51 is an IBM-PC/XT/AT or compatible. we supply the rest.

PDK51 PLUS includes everything in the PDK51 plus Vers. 3 of our popular BXC518051/8052 BASIC compiler-$800.

Call Now! 508-369-9556 or FAX 508-369-9549

Binary Technology, Inc.
P.O. Box 541. Carlisle, MA 01741

#103

PRODUCT DEVELOPMENT
* MADE EASY! *

With EMAC's feature packed **Single** Board **Computers** and **easy to use** BASIC **compiler your product/application** can become a reality in no time. **EMAC's** BASIC compiler allows you to develop the software right on **the Single Board Computer. No more time lost linking,** downloading, or burning EPROMs. immediately see the results of a program change. Make modifications or dump memory **right** on the board. Since this BASIC is compiled and written entirely in assembly language, your programs will fly. Full **floating point, trig. functions, windowing, and networking,** ail at your fingertips. But you say you need **multitasking**; how about up to **32** tasks with **priority** scheduling built in. No adding libraries. And this **multitasker** is as easy to use as the BASIC language. If BASIC is not your language of choice, **EMAC** offers **Assembler, ANSI C, and** Forth - you choose. But enough about the software, **software** is nothing without a good hardware foundation with the features you need to make the job easy. Need to drive relays? Take advantage of **8 high** drive d**igital output lines. Have some digital** input lines that are kind of noisy? **Utilize 8 optically coupled inputs.** You can also **take** advantage **of 16 programmabl**e **I/O** lines, 6 **timer/counters,** and 4 serial **485 ports. But wait there's more, a fast, 16 channel,13 bit A/D, opt. 2 channel, 12 bit D/A, and opt. real time clock. Add up to 512K memory, EEPROM, single supply operation and the EPAC 3000 G2 can handle just about anything. The IEPAC 3000 G2 is just $389.00 Qty. 1**.

EMAC, inc.

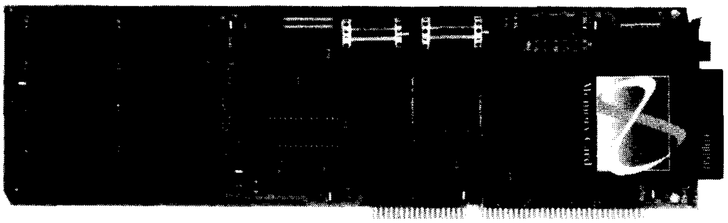
618-529-4525 FAX: 61 8-457-0110

P.O.BOX 2042 CARBONDALE, IL 62902

#104

NEW PRODUCT NEWS

SOLID-STATE DISK SYSTEM FEATURES REMOVABLE MEMORY CARD



The MNC 1150 high-speed solid-state "disk" system features a removable PCMCIA 2.0 memory card slot for ISA and EISA computers. The MS-LXX-compatible system stores files in fast semiconductor memory rather than mechanical disks to achieve the high speed without wear. The on-board BIOS ROM allows instant-on and instant-reset capability.

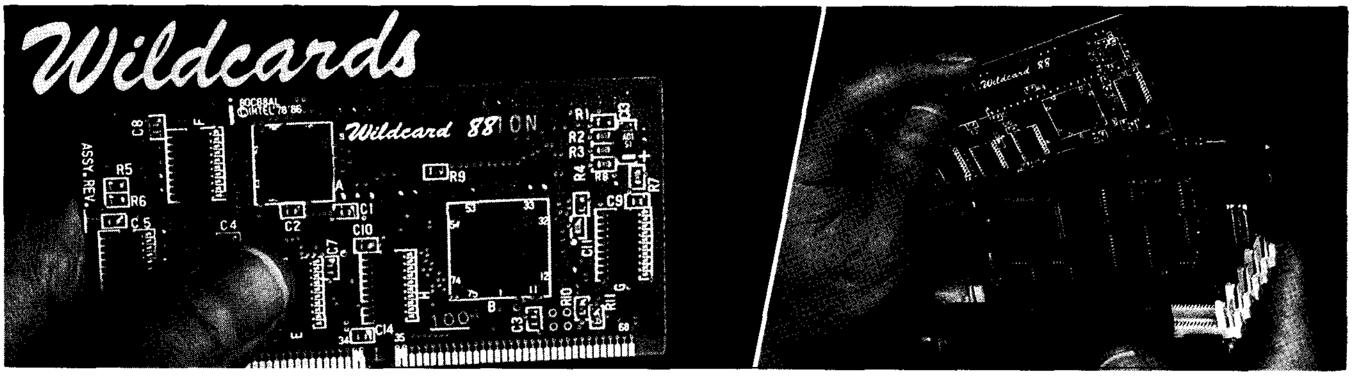
The MNC 1150 offers complete PCMCIA 2.0 compatibility, which can support memory cards up to 64 megabytes with a 16-bit data path. In addition, eight 32-pin JEDEC sockets for SRAM, EPROM, EEPROM, and flash memory chips are provided (for a total storage capacity of 72 megabytes]. It has been designed with CMOS technology using a versatile register-based interface with error sensing and selectable interrupts to ensure data integrity in critical environments.

The PCMCIA 2.0 card socket is accessible through the card bracket, so memory cards can be loaded and unloaded while the computer system is operating. Dual-battery backup is provided for the JEDEC sockets when SRAM is used. On-board flash memory programmer and support software is provided. Power requirements are less than 1 amp at 5 volts and up to 250 mA at 12 volts (when programming flash memory).

Evaluation units are available for $295 with zero memory.

MNC International, Inc.
2817 Anthony La. South
Minneapolis, MN 55418
(612) 788-1099
Fax: (612) 766-9365

#506



FEATURES

DESIGN CONTEST WINNER

FEATURE ARTICLE

Brian Farmer

Planting Geraniums by Robot

Build an MC68HC11-based 2-D Sensor

What do you do when you need a sensor for an application nobody has thought of? Build your own. This unique sensor helps a robot prevent damage to delicate plant cuttings when handling them.

“Opposites attract,” so the saying goes. Maybe it’s true with people, but definitely not with robots and flowers. Even though the latter are about as opposite as can be, they don’t attract. Matter of fact, they are about as incompatible as you can get. Robots are inclined towards uniformity, and flowers are inherently diverse. That’s where my sensor comes in, providing a necessary ingredient for a marriage that would otherwise never work, or technically speaking, assisting a robotic system to accommodate nonuniformity.

Robotic systems aren’t anything new. They’ve been around industry for years, improving productivity and quality and doing many hazardous or uncomfortable tasks for workers. For example, the automotive and electronics industries use robots to perform repetitive and precise tasks. This form of automation is implemented in predictable and structured environments [1] with defined spatial configurations.

Agricultural robotics ventures beyond the predictable to the unpredictable environment. No formal methodology exists for designing automation systems that work with biological products that are of variable

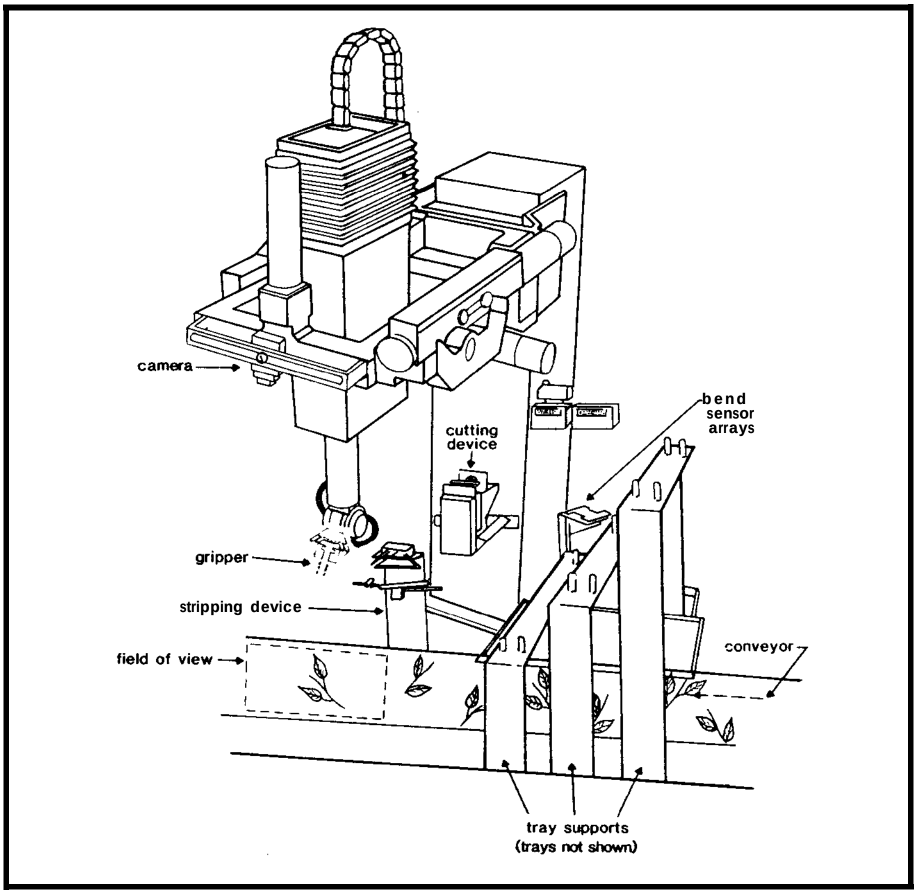


Figure I-The robot's job is to pick plant cuffings off a conveyor belt, trim leaves and the stem, and insert the cutting into a plug of peat. The sensor helps tie robot compensate for the natural bend in the plant's stem.

size, shape, color, orientation, and stress or strain relationships [2].

This scenario presents quite an inviting challenge, and who can resist one of those? Not many. Fundamental research is widely underway to establish methods for the design of cognitive machines [robots) that can autonomously or semiautonomously operate with contingencies.

THE ROBOT

Most robots are capable of repeating a preprogrammed sequence of operations; however, for robots to operate effectively in a changing or uncertain environment, the machine must be equipped with sensors. Therefore, a key activity in robotics research is examining methods for coordinating information from various sensors to control robotic actions [3].

At the Georgia Station Mechatronics Laboratory, research is investigating pragmatic techniques that allow robotic systems to accommodate the significant variability present when handling and processing living plant materials. This accommodation of variability is contrary to a typical industrial application where an attempt is made to eliminate any inconsistencies.

Dr. Ward Simonton developed a robotic workcell for the labor-intensive preparation of geranium cuttings for propagation to be used as a case study. A conveyor brings geranium cuttings into a robot work envelope. Using machine vision, the supervisory computer locates a cutting and classifies all primary plant parts. Knowledge of plant structure is used to grade and determine an appropriate processing strategy.

Control is passed to the robot and end effector controllers. The robot grasps a single cutting from the conveyor with a gripper, moves it to a pneumatic device for leaf removal (if necessary), takes it to a pneumatic cutter for stem trimming, and inserts it into a plug of propagation medium [in this case, peat). Fixtures and sensors mounted within the workspace assist the processing (Figure 1).

THE CHALLENGE

The problem the robot had early on was it damaged a significant number of cuttings. Its strictly vertical insertion motion into the plug did not regard the bend in the geranium stem. It could neither recognize different relative angles of stem growth frequently found on cuttings nor the change in stem orientation during workcell preparation.

For the reliable insertion of cuttings into plugs, the direction and degree of bend at the base of the stem had to be determined. Then, the robot arm had to make the appropriate moves to align the end of the stem with the hole in the peat plug accurately.

Company/item	Model Name/No.	Range	Resolution
Type: LVDT			
Schlumberger Ind.	ACR/15	0.030 m	∞
Type: Ultrasonic			
Polaroid Corp.	Design Kit	0.27 m-10.7 m	0.003 m-3 m
Contaq Tech.	UDM-RMU	0.15 m–9.14m	0.0002 m
Ultrasonic Arrays	DMS-100	0.005 m-6.61 m	0.0002 m
Cosense	ML-1 01	0.02 m-2.74 m	0.00005 m
Type: Optoelectronic			
Banner Engineering	MP-6	3.05 m	0.013 m
Banner Engineering	SBF5 w/IR2.53S	0.025 m	∞
Frost Controls	Edge Sensing Sensor	0.025 m	0.0025 m
Keyence	2-D, LS-3033 & LS-3100	0.03 m	0.000002 m

∞ : Analog output; resolution primarily limited by measuring device.

Table 1—Numerous sensors are available that might have helped in this project, but none fit the project criteria.

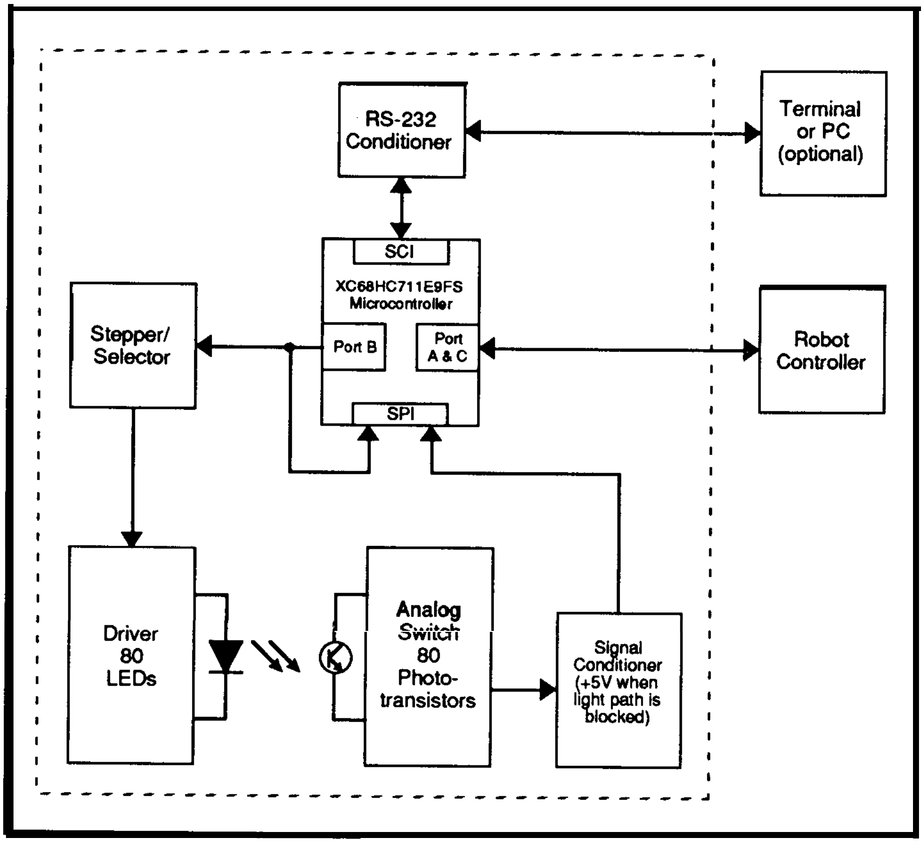


Figure 2—*The external robot controller starts the whole measurement process. Results may be optionally displayed on a terminal screen.*

I considered several commercially available sensors for the stem bend measurement, including **LVDTs,** ultrasonic transducers, and optoelectronics (refer to Table 1). With the LVDT, physical contact between its armature and the object sensed requires force, resulting in deflections and *incorrect* measurements due to the elasticity of the geranium stem.

Ultrasonics (Contaq Technology) can make noncontact measurements; however, it needs relatively precise insonification of the ultrasonic beam on a specific area of a small object. Standard collimating tubes do not normally ensure a measurement focused on a 1- to 2-mm object section. In addition, precision ultrasonic sensors are costly (Ultrasonic Arrays & Cosense).

I then turned to commercial optoelectronic sensors. Discrete measurements from an arrayed "curtain of light" would be possible with an emitter-receiver pair for binary blocked/unblocked detection. I considered an b-channel Sensor

Multiplexer Module (Banner), but it has fairly poor resolution (5-12 mm]. Other light curtain devices offered by Banner and Sick Optoelectronics simply provide presence detection or the edge measurement of a surface.

A 1-D C-shape sensor is available from Frost Controls that measures an object's size using an analog signal proportional to the area blocked, and it also can be constructed to detect position. However, this unit is 12.5 mm thick, which is the minimum displacement between measured dimensions. Due primarily to cost and complexity, I did not consider the various types of laser-based position measurement systems (Keyence, $14,400) for this application [4].

Evidently, no commercially available position measurement device fit the needs of this application, so I developed a sensor and coupled it to an MC68HC11 microcontroller [5,6,7]. The pairing of the measuring element and the inexpensive microcontroller yielded an externally controlled programmable sensor that reported measurement results through a parallel/serial interface.

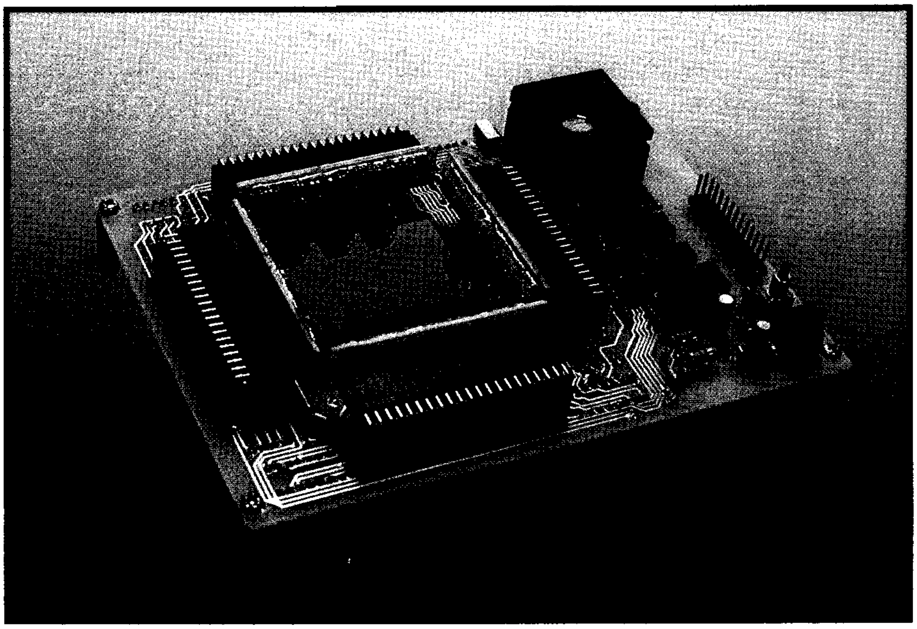


Photo I-The *sensor* uses *pairs* of *LEDs* end *phototransistors* b *sense* the *position of a plant cutting inserted in the hole* in *the middle.*

DESIGN CRITERIA

The primary design goal for my sensor was that it detect the direction and magnitude of bend on a geranium cutting stem's base relative to the robot's grasp. To make this determination, the robot is directed to position geranium cuttings in a fixed location in the workcell, then to measure

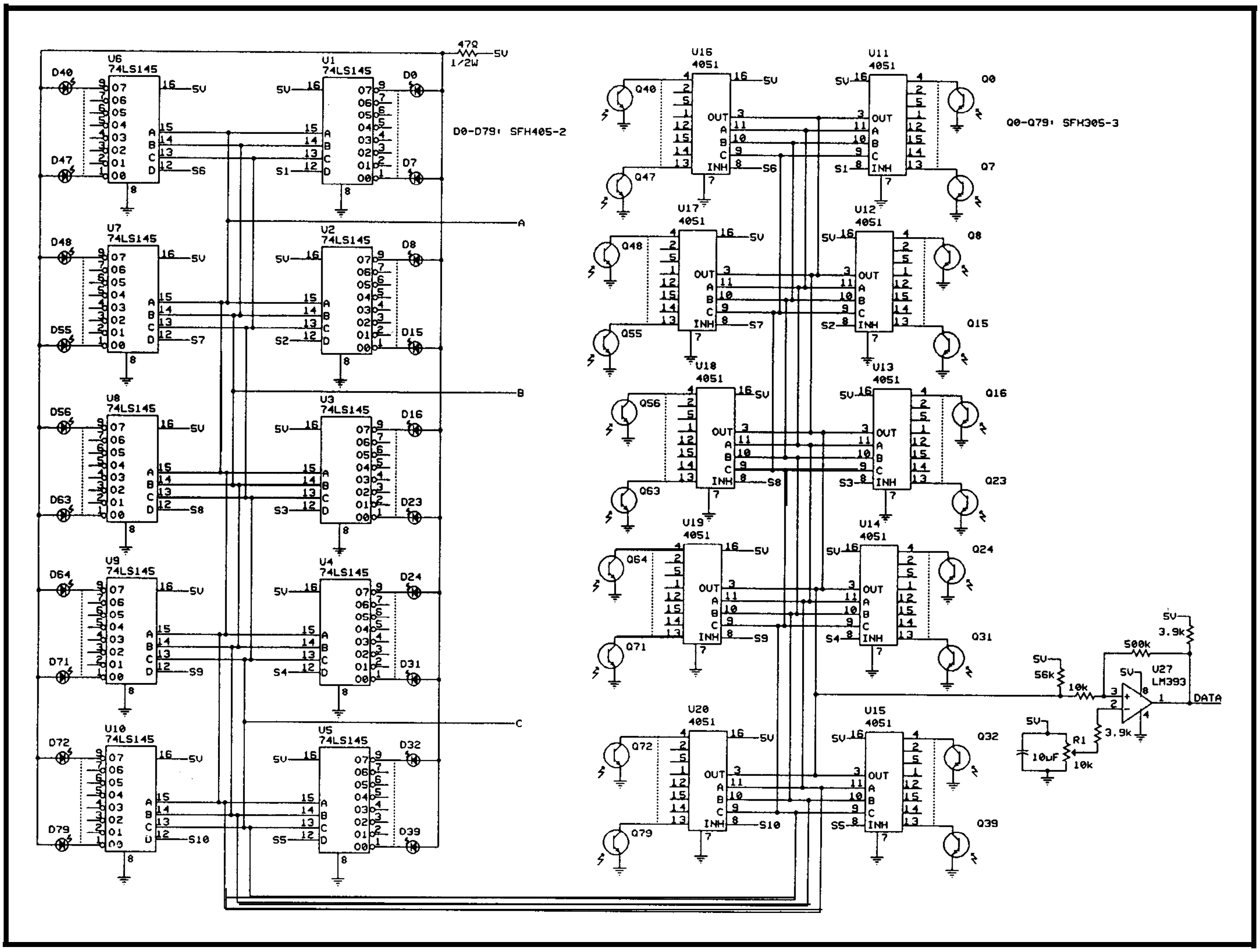


Figure 3a—*The bulk of the sensor is made up of the LED/phototransistor pairs. One at a time, each LED is lit and its companion phototransistor is checked to find out if the light path is being blocked.*

quickly the stem's position with a sensor. The bend of the stem is referenced from the robot's grasp point to the bottom of the stem, and the information is used when inserting the cutting into the plug. Geranium cutting stems are frequently 5-10 mm in diameter with bends of O-O.6 radians. Before a stem has been inserted into the plug, it generally extends below the robot's fingers 15–20 mm.

Thus, the design criteria for the sensor included a l- to 2-mm resolution, a 25-mm range in two dimensions, a l-mm maximum displacement between measured dimensions, and a 100-ms maximum response time. Object size was assumed to be within the range of 4-12 mm. Also, I needed to keep component cost under $500. I pursued a small, noncontact sensing method because of the infrequent need for cleaning and the absence of any force applied to the plant stem, which could affect the measurement or disturb the cutting within the gripper.

THE SENSOR

In 1989, I developed the first sensor for this application [4]. It was based on two lo-element infrared LED emitter and phototransistor receiver pairs (Siemens LD260/ BPX80) [8] physically aligned as a 25-mm x 25-mm square. Emitter-receiver pairs were not multiplexed. The entire first dimension was activated and latched, and the same was repeated for the second dimension. Both dimensions were read by a 6809 microprocessor through a parallel PIA interface.

The minimum object size measured was 5 mm in diameter or width, and average position measurement errors in the x and y dimensions were 2.4 mm and 2.0 mm, respectively (these values are within the range of expected accuracy due to the 2.5-mm resolution). Response time was 115 ms, repeatability was excellent, and the unit insertion operation increased in performance from 80% to 98% for 204 cuttings. The sensor was connected to a single-board computer, the Wintek 6809 Control Module.

While the position sensor performed well, I needed to improve resolution and range to achieve the design goal for a minimum object size measurement of 4 mm and to overcome the occasional shortcomings of the 25-mm range. The resulting project won first place in the General Category of the third annual Circuit Cellar Design Contest (see Photo 1). It uses multiplexed emitter-receiver pairs (Siemens SFH405-2/SFH305-3) with a smaller component width, a smaller

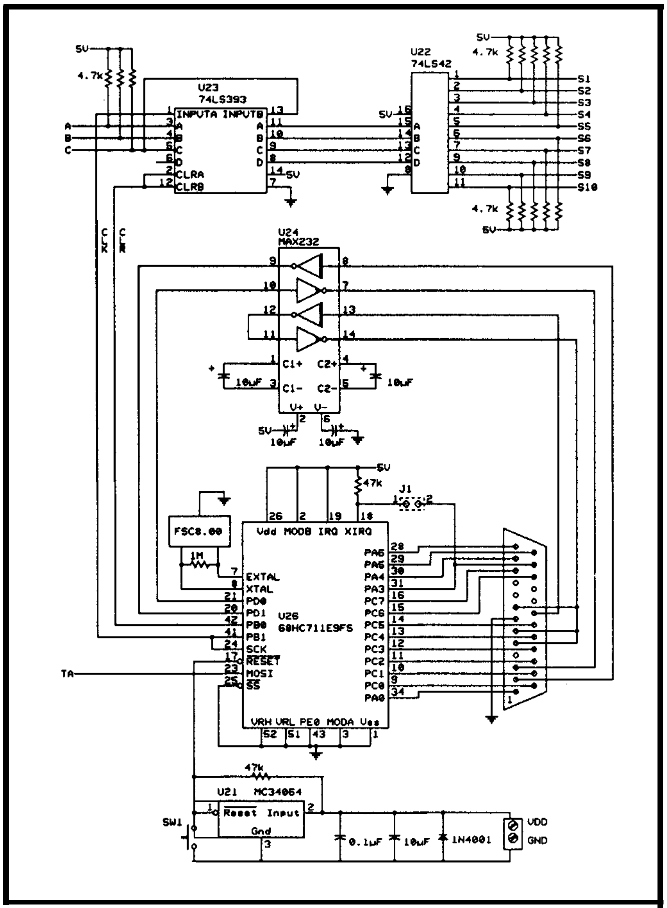


Figure 3b—*An MC68HC11 processor oversees the entire measurement process. The processor's on-board memory and peripherals keep external support circuitry to an absolute minimum.*

emission-acceptance angle, a larger 50-mm x 50-mm sensing area, and an MC68HC11 microcontroller unit (MCU).

The entire measurement process [refer to Figure 2) is requested by an external robot controller to the smart sensor. Once initiated, the MCU controls and reads the sensor through a three-line interface: *CLR*, *CLK*, and *DATA*. The MCU then calculates the center of the object inserted into the square based on absence of light at individual receiver elements in two dimensions. The object position is coded using 6 bits for each dimension.

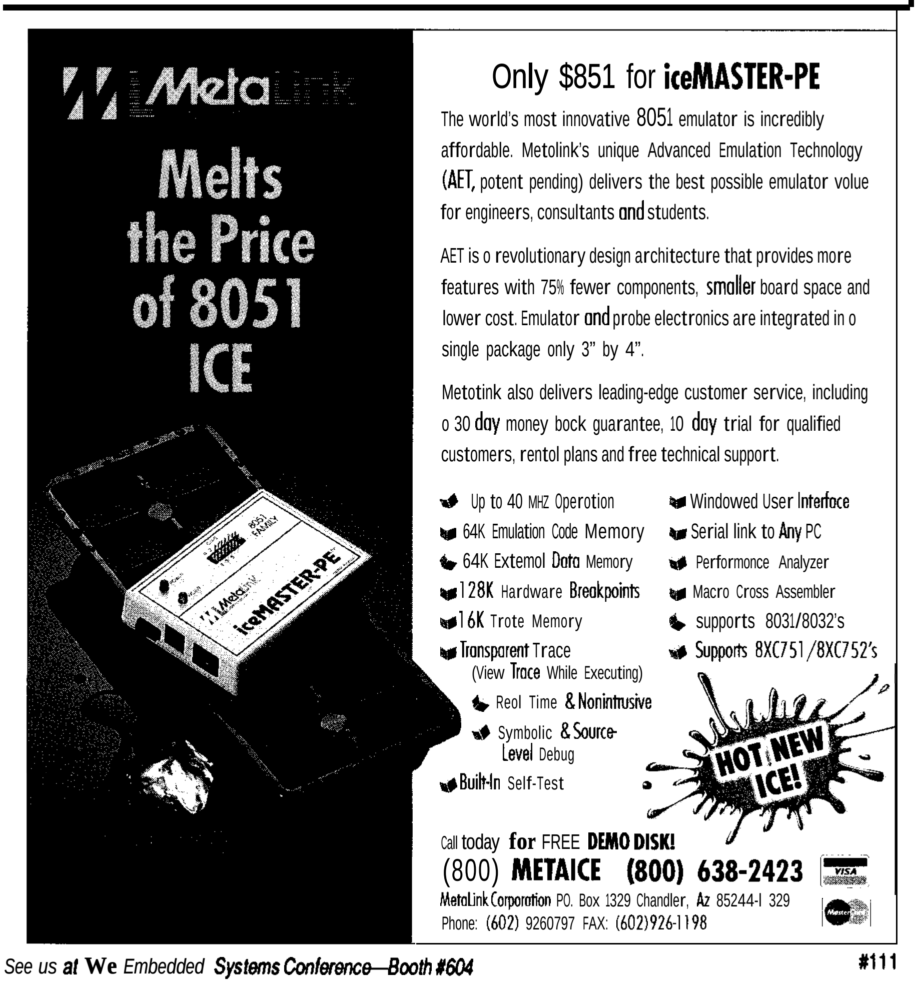
Once a GO request is received, the computer sequence is as follows [refer to Figure 3 for the schematic): The MCU resets U21/U22, turning on the pair D7 and Q7 and latching their condition into the synchronous serial peripheral interface (SPI) port. It then sequences down the line along the horizontal (x) axis to D32 and Q32, storing each pair's condition in on-board RAM in single-byte groups. The scan continues in the y direction without interruption to D72 and Q72.

Notice the reverse sequencing process: 7, 6, 5, 4, 3, 2, 1, 0, 15, 14 ,... 39, 38, 37, 36, 35, 34, 33, 32, 47, 46 ,... 73, 72. This feature allows the highest to the lowest pair to match with the MSB to the LSB in on-board RAM and also follows the high-to-low order SPI format. Only one pair is energized at any one time.

If any of the beams within these 80 pairs is obstructed, the output of U26 will go high for the respective pair according to the threshold set by R1. A high or low will be loaded into the SPI port for all pairs blocked or unblocked, respectively [9]. When the MCU provides the stepping-clocking sequence for the final pair, one extra count is clocked in order to deactivate all pairs to lessen power drain during idle time. Both sets of serialized signals are read from on-board RAM, and object position is calculated and sent to the parallel/serial output.

THE SOFTWARE

My MC68HC 11 configured with pin *PE0* to ground and jumper J1 allows RS-232 sensor-to-PC software



Listing 1—*A demonstration program shows what's involved in dealing with the position sensor.*

```
*****
* 68HCll Register Equates
*****

PORTC    EQU     $1003
PORTB   EQU      $1004
DDRC     EQU     $1007
DDRD     EOU     $1009
SPCR     EQU     $1028
SPSR     EQU     $1029
SPDR     EOU     $102A

*****
* Buffalo 3.2 Eauates and Jump Table adr
*****

USTACK  EOU      547               (0033-0047)
WARMST  EQU      $FF7C             Warm start

*****
* RAM VARIABLES
*****

         ORG      so001            Lower half of RAM

DA10     RMB      1                U10 & U20 pairs
DA9      RMB      1                U9 & U19 pairs
DA8      RMB      1                U8 & U18 pairs
DA7      RMB      1                U7 & U17 pairs
DA6      RMB     1                 U6 & U16 pairs
DA5      RMB      1                U5 & U15 pairs
DA4      RMB     1                 U4 & U14 pairs
DA3      RMB      1                U3 & U13 pairs
DA2      RMB      1                U2 & U12 pairs
DA1      RMB      1                U1 & U11  pairs

         ORG      so100            Upper half of RAM

*****
* START
*****

 * Register Initialization

         LDS      #USTACK          Init stack pointer
         LDAB     #$00             Zero B for counter
         LDX      #$000A           Set X as counter
 * SPI initialization
         LDAA     #$02             Prepare CLK/SCK hi
         STAA     PORTB            and CLR lo
         LDAA     #%00000010
         STAA     DDRD             SS*,SCK,MOSI,MISO,RxD-In,TxD-Out
         LDAA     #%01001100       SPI on as slave,
         STAA     SPCR             CPHA=1,CPOL=1
         LDAA     SPSR             Clear any
         LDAA     SPDR                possible SPIF
• PBO is 74LS393's CLR. PB1 is 74LS393's
* CLK and SCK for SPI.
• Initiate scan
         LDAA     #$01             First falling edge
         STAA     PORTB               of SCK indicates
*                                     start of xfer
         CLR      PORTB            Scan initiated with
*                                     D7/Q7 pair activation
         JSR      WAIT1            Wait a while
         LDAA     #$02             D7/Q7's condition
         STAA     PORTB            xfered thru MOSI
```

(continued)

Listing 1-continued

```
        INCB                    Record data count
* Loop  through scan
LP1     CLR     PORTB           Next pair activated
        JSR     WAIT1           Wait a while
LP2     LDAA    #$02            Next pair's condition xfered
        STAA    PORTB
        INCB                    Record data count
        CMPB    #8              One byte yet?
        BNE     LP1             No, go back for more
        CLR     PORTB           Yes, activate next pair
LP3     LDAA    SPSR            SPDR full with 1 byte?
        BPL     LP3             Keep looking for flag
        LDAA    SPDR            Load into on-board RAM
        STAA    0.X
        JSR     WAIT1           Wait a while
        LDAB    #$00            Reset data counter
        DEX                     Count down
        CPX     #$0000          All data in?
        BNE     LP2             No, go back for more
        LDAA    #$0C            Yes, SPI off. Resets/ends
        STAA    SPCR            transfer. And...
        JMP     WARMST          go back to Buffalo prompt ">"
*
* Subroutine
*
WAIT1   LDAA    #$30            Counter
WAIT1A  SUBA    #1              Count down
        BNE     WAIT1A          Finished?
        RTS
```

debugging and development with Motorola's BUFFALO program. Listing 1 provides a demonstration program used to acquire and store data.

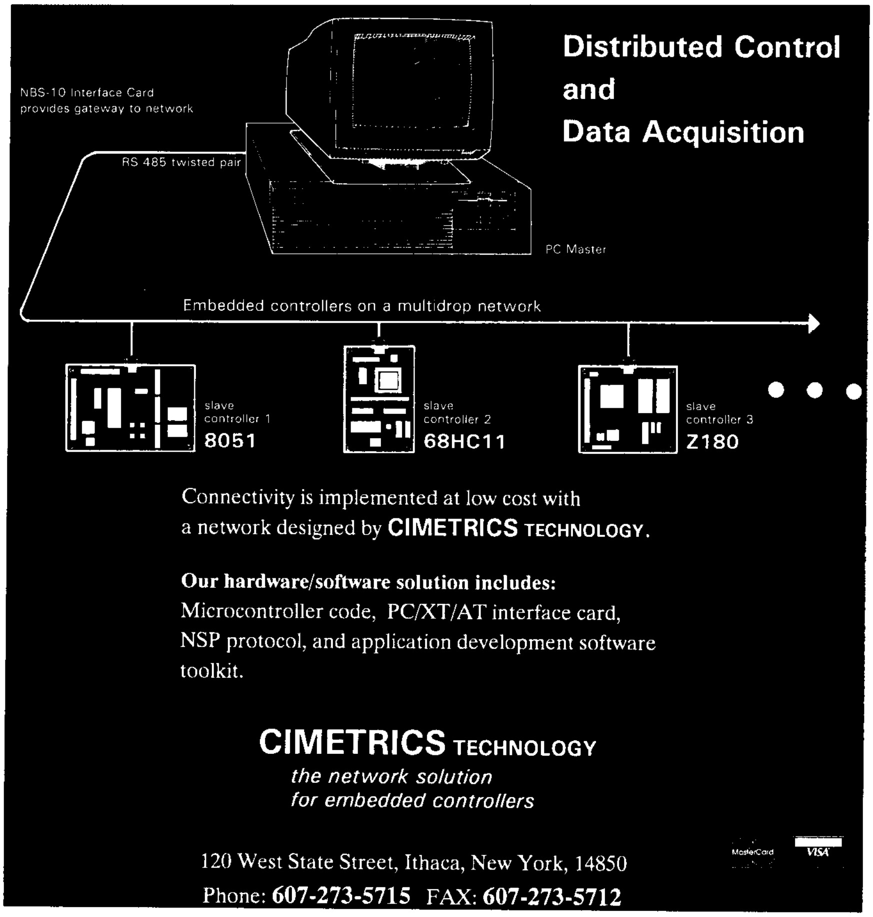
The code is implemented in the MCU's on-board RAM as a subroutine while under control of the BUFFALO program, with a warm start back into BUFFALO after program execution occurs. Again, acquired emitter-receiver pair data is displayed on a PC's screen from on-board RAM with a BUFFALO command. The most important software detail involves the *SCK* and *DATA* timing. *SCK* (*CLK*) speed is dependent on the phototransistors' rise and fall time.

Although SPI rates as high as 2.1 megabits per second are possible, Siemens specifies the receiver with a 6 µs maximum rise and fall time. Still yet, testing yielded 129 µs due to the effect of coupling with the multiplexer, resulting in an optimum period of 135 µs (7.41 kHz, which isn't one of the four selections in master mode). SPI polarity and phase were initialized for stepping on the falling edge of the clock, and data latching on the rising edge.

Calculating position requires determining the nature of the data from the arrays. Data is classified as reliable or unreliable. Unreliable data refers to noise (any pattern other than one consecutive group of blocked elements) or signal absence (all elements unblocked or all elements blocked), with appropriate codes transmitted for each type of condition. Reliable data is processed by a simple algorithm to locate the object center.

As I noted earlier, the microcontroller uses six parallel bits per dimension to code the calculated object position within the array of the sensor. In addition, raw position, position codes, and error codes are sent to the optional terminal for user verification if desired. The 26-pin dual-row header on the circuit board provides RS-232 interfacing when a 25- to 9-pin adapter is used. Nine of the 25 pins are used to avoid circuit contention with the parallel output on the same header.

Eventually I will want robustness: the ability to handle unreliable data. I



hope to implement a form of fuzzy logic in the future to deal with the tendency of the data rather than rigid rules of traditional logic [10].

PERFORMANCE OF THE SENSOR

I conducted several tests to determine operational limits of accuracy, repeatability, and speed. Tests showed a minimum of 0.22 mm for object size measurements. Average position measurement errors in the x and y dimensions were 0.07 mm and 0 mm, respectively. Measurement time was 11 ms (demonstration program). Finally, the sensor had excellent repeatability and a greatly reduced power drain (including a single +5-volt supply] compared to the first circuit.

The component cost was $476.76. An insertion performance of over 98% of cuttings is projected. The sensor has added some other benefits as well. It can detect the proper grasp of a cutting by the robot before attempting insertion into the plug. Also, it has the capability to abort a particular cutting cycle if no object is detected. ▣

REFERENCES

1. **W. L. Whittaker,** G. **Turkiyya,** M. **Herbert, "An architecture and two cases in range-based modeling and planning," IEEE, 1987.**
2. **W. Simonton, "Automatic plant handling and processing in a robotic workcell,"** ***Transactions of*** **the ASAE, 1990.**
3. **W. Simonton, "Issues in robotic system design for transplant production systems,"** *International Symposium on Transplant production Systems,* **Yokohama, Japan, 1992.**
4. **W. Simonton, B. Farmer, "Sensor for two-dimensional position measurement of small objects,"** ***Transactions of the ASAE,* 1992.**
5. **S. Ciarcia, "Why microcontrollers?",** ***BYTE,*** **August 1988.**
6. **S. Ciarcia, B. Brown, "Using the Motorola MC68HC11", Circuit** *Cellar INK,* **Issue 18, December/ January 1991.**
7. *M68HC11 Reference Manual,* **Motorola Inc., 1989.**
8. *Optoelectronics Data Book,* **Siemens Components, Optoelectronics Division, 1990.**
9. **S. Ciarcia, "Let your fingers do** the talking", August 1978, ***BYTE.***
10. **M. AbdelRahman, "Artificial Intelligence expands sensor applications",** *Electrical Construction & Maintenance,* **December 1991.**

Brian Farmer was formerly an Electronics Engineer with the University of Georgia Experiment Station and occasionally does consulting through Power Tech Inc. He is now a full-time student in the University of Arkansas's Electrical Engineering Graduate program with a teaching assistantship.

My ***thanks to the clear thinking and wise supervision of Dr. Ward Simonton.***

I R S

401 Very Useful
402 Moderately Useful
403 Not Useful

114

#1

DESIGN CONTEST WINNER

FEATURE ARTICLE

Brian Kenner
& John Wettroth

The Design of a Time Domain Reflectometer

As anyone who's tried to track down an elusive network cabling problem knows, improper termination and bad cables are common culprits. Let this time domain reflectometer make your life a little bit easier.

a few years ago, a friend of mine needed a portable network cable tester. He does LAN management and found that cable problems were the cause of a very large number of network instabilities. I wasn't surprised, having also managed several large networks at one point in my life. I had grown weary of having a system fail whenever someone decided to rearrange the office and inadvertently disconnected the network while moving a computer.

Certainly, no replacement exists for building networks with some level of *fault tolerance,* but doing so is not always physically possible or economically viable. The type of cable tester my friend suggested was not going to solve network errors originating from the "designer office worker" anyway.

Nevertheless, my friend reasoned that if a truly fault-tolerant network wasn't possible, providing the means to quickly locate the cable fault was the next best thing. He proposed I build a network cable tester: a Time Domain Reflectometer (TDR).

Our portable TDR is based on the Intel 87C51 microcontroller. We designed it primarily to measure the length and termination impedance of coaxial cables commonly used in computer networks, such as Ethernet, Arcnet, and so forth.

The TDR uses time domain reflectometry techniques to measure length and calculates the actual termination impedance based on the amplitude of the reflected waveform. It has a measurement range of approximately 1800 feet and handles 50- and 75-ohm characteristic impedance cables (primarily RG-58 and RG-59 used in *thin-wire* networks).

The hardware combines an 87C5 1 microcontroller with a 4-line 20-character LCD display, an infrared three-key touch-screen interface, a low-power PAL, a dual 8-bit DAC, a very high-speed comparator, and a handful of digital and analog "jelly-beans" to bring it all together.

The software consists of approximately 3K of 805 1 assembly language, which includes a flexible menu system, a custom integer math package, and the actual measurement code. A 9-volt battery powers the unit, which features automatic shutoff, and it is packaged in a 2" x 4" x 6" plastic enclosure. We also kept the bill of materials below $200.

THEORY OF TDRS

The TDR capitalizes on the traveling waveform's characteristic of reflecting some portion of its power as it passes through the interface of two dissimilar materials. A detailed understanding of why these reflections occur requires more space than I have. However, I will cover the theory necessary to build the TDR, which centers on how electromagnetic waves propagate down a transmission line.

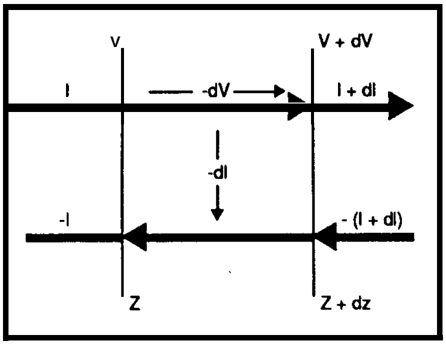


Figure I-Some *vectors* and *differential calculus help show how the time domain reflectometer works.*

Transmission lines result from the interaction of wire sections that make up any circuit. In most cases, the interaction between the wire in the signal and return paths is insignificant relative to the effect of the physical components that form the system. However, in the case of network cabling, the distributed reactance (both inductive and capacitive) is substantial with respect to the data rates involved. Figure 1 and some differential calculus help show how these reactive elements sum.

In an infinitesimally small element of the transmission line, ***dz,*** voltage and current are defined by

$$V-(V+dV)=-dV=(Ldz)\left(\frac{\partial I}{\partial t}\right)$$

and

$$-dI=(Cdz)\left(\frac{\partial V}{\partial t}\right)$$

where ***L*** is the series inductance and C is the parallel capacitance of the cable conductors measured in inductance and capacitance per unit foot. If I assume a **lossless** transmission line, I can combine the above equations to form a pair of linear differential equations representing the variation of voltage and current in the conductors at position z. These equations are

$$-\left(\frac{\partial V}{\partial z}\right)-L\left(\frac{\partial I}{\partial t}\right)$$

$$-\left(\frac{\partial I}{\partial z}\right)-C\left(\frac{\partial V}{\partial t}\right)$$

$$\frac{\partial^2 V}{\partial z^2}--\frac{\partial}{\partial z}\left(L\frac{\partial I}{\partial t}\right)--\frac{\partial}{\partial t}\left(L\frac{\partial I}{\partial z}\right)-LC\frac{\partial^2 V}{\partial t^2}$$

They have a solution of

$$V=F_,(z_o-vt)+F_2(z_o+vt)$$

where v and z_o must be

$$v=\frac{1}{\sqrt{LC}}$$

$$z_o=\sqrt{\frac{L}{C}}$$

z_o and v play very important roles in the **TDR's** capability to make its measurements. z_o is the characteristic impedance of the cable and is used to determine the magnitude of the waveform (and possibly phase) reflected at the interface of the two conductors. The parameter v, defined as

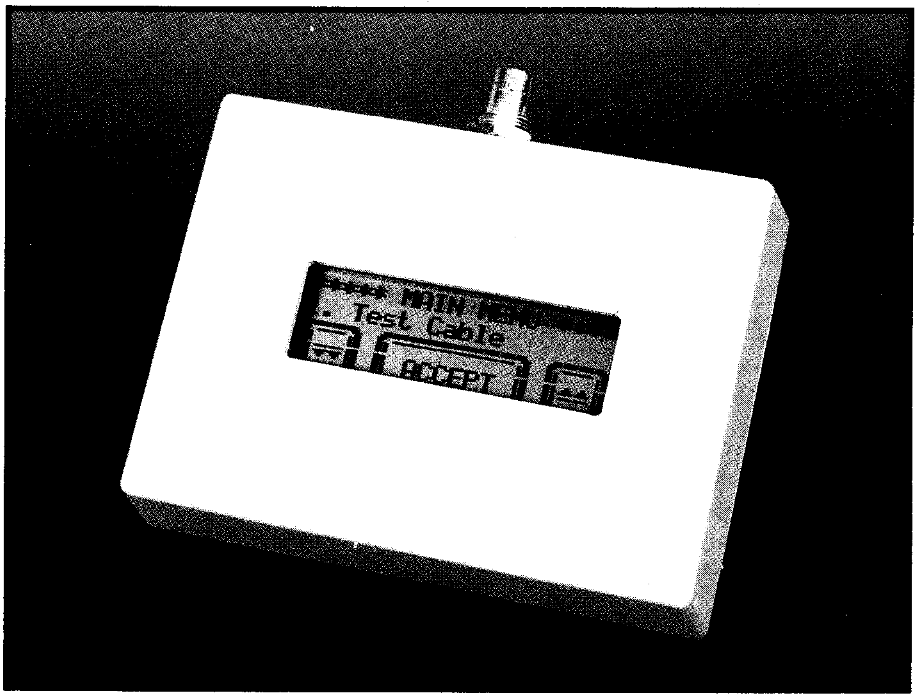


Photo I-The *time domain* ***reflectometer*** *uses a* ***4-line*** *by* ***20-character*** *display for* ***all operator*** *input and* ***output. Soft*** *keys and* ***touch*** *screen* ***circuitry*** *make for a vety* ***flexible interface.***

$$v=\frac{1}{\sqrt{LC}}-\frac{1}{\sqrt{\varepsilon\mu}}$$

is the velocity of the electromagnetic wave in the cable and is used to compute the physical distance to the discontinuity.

In a vacuum, the dielectric constant (E) and permeability (μ) are equal to 8.854 x 10^{-12} F/m and $4\pi \times 10^{-7}$ H/m, respectively, so the speed of propagation becomes c, or the speed of light.

Defining the equations for capacitance and inductance per unit length of the two most commonly used types of cable is also helpful. For coaxial conductors

$$C=\frac{2\pi\varepsilon}{\ln\left(\frac{b}{a}\right)},\quad L=\frac{\mu\ln\left(\frac{b}{a}\right)}{2\pi}$$

For parallel conductors

$$C=\frac{\pi\varepsilon}{\ln\left(\frac{2d}{a}\right)},\quad L=\frac{\mu\ln\left(\frac{b}{a}\right)}{2\pi}$$

Using these equations, I can define Z_o in terms of the physical and electrical properties of the cable: parameter a, the diameter of the inner conductor; b, the diameter of the outer conductor in coaxial cables; and ***d,*** the distance between conductors in parallel wire cables. The equationfor a coaxial conductor is

$$Z_o-\frac{1}{2\pi}\left(\frac{\mu}{\varepsilon}\right)^2\ln\left(\frac{b}{a}\right)$$

and for a parallel conductor is

$$Z_o-\frac{1}{\pi}\left(\frac{\mu}{\varepsilon}\right)^2\ln\left(\frac{2d}{a}\right)$$

The TDR makes its basic measurement by injecting a pulse into the cable and subsequently measuring the amplitude and phase of any reflected pulse. The reflected pulse's amplitude and phase indicate the impedance of the new material. The time elapsed between pulse injection and the detection of the return pulse is then used to compute the round-trip distance to the cable discontinuity.

What may be obvious at this point is if the pulse is not reflected, no discontinuity exists in the cable. The mystery is why any portion of the wave reflects at all, and upon being reflected, why the electromagnetic wave may undergo a phase change.

Though the **TDR's** injected pulse is by no means sinusoidal, Fourier analysis shows it is composed of sinusoids. For the following discussion, it is easier to assume the traveling wave is purely sinusoidal and of some particular frequency. If the transmission line is terminated and assumed to be lossless, then the load,

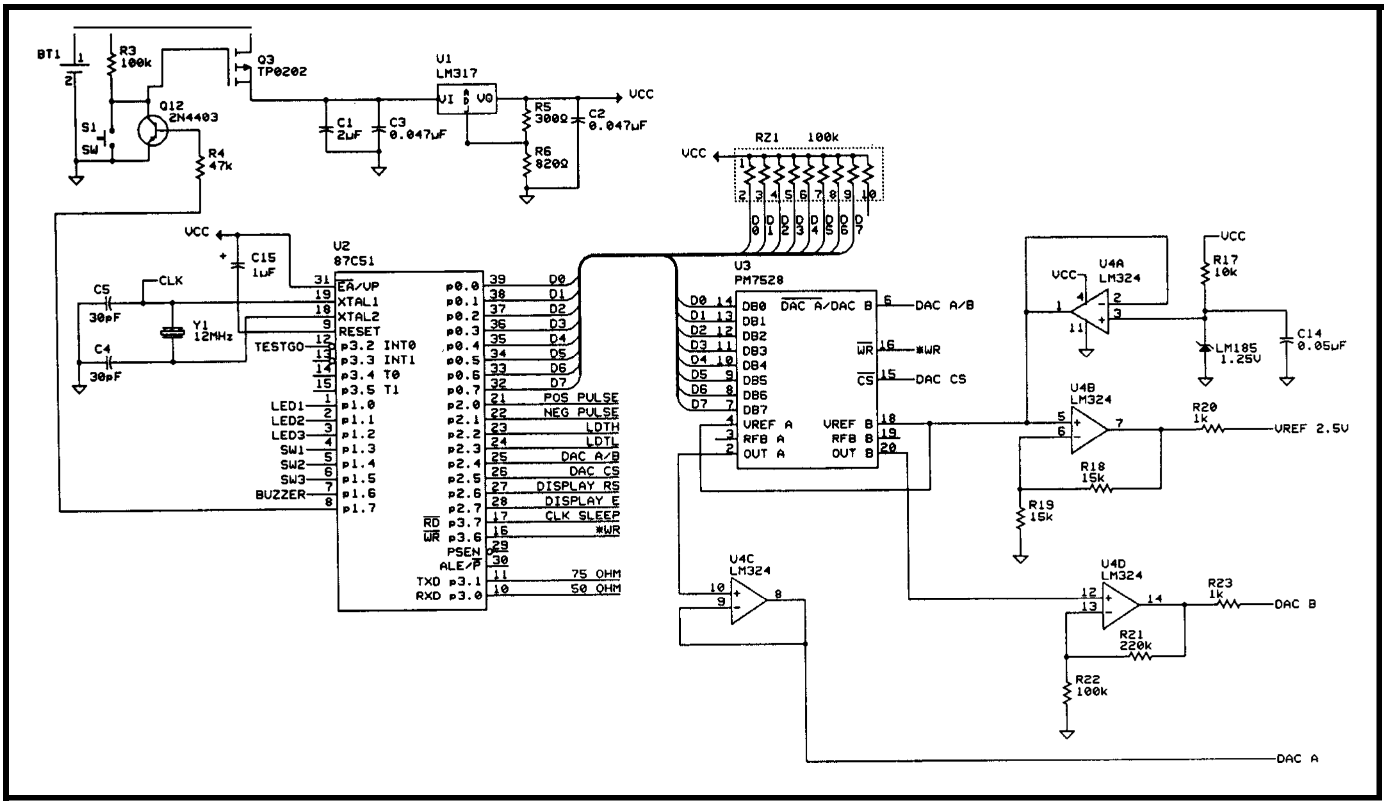


Figure 2-An ***87C51*** nms ***the TDR.*** A ***2-channel*** DAC ***is used*** b ***control the LCD display contrast*** and to ***create*** a ***variable offset for the measurement circuits.*** *The* ***power control circuitry automatically powers the*** *system up* ***at the press*** **of a** ***button*** *and* ***powers*** *if down* ***after*** a ***fixed period of inactivity.***

Z_L, must also equal V/I of the incident waveform. Thus, using the equation developed for V and I, the following relationships can be established:

$$V = Ae^{j\omega(t-z/v)} + A'e^{j\omega(t-z/v)}$$

$$Z_oI = Ae^{j\omega(t-z/v)} - A'e^{j\omega(t-z/v)}$$

Don't let the complex exponentials distract you. They are, in conjunction with the parameter A, part of the characterization of the general sinusoid and will be divided out. What remains is an equation relating the complex amplitude of the incident waveform to the load impedance, Z_L, and the characteristic impedance, Z_o.

$$\left(\frac{V}{Z_oI}\right)_{z_o-0} = \frac{Z_L}{Z_o} = \frac{Ae^{j\omega(t-z/v)} + A'e^{j\omega(t-z/v)}}{Ae^{j\omega(t-z/v)} - A'e^{j\omega(t-z/v)}}$$

$$= \frac{(A+A')}{(A-A')}$$

$$\frac{A'}{A} = \frac{(Z_L - Z_o)}{(Z_L + Z_o)}$$

This equation is the definitive relationship for TDR. With it, you can anticipate the reflection of any waveform with predictable results. In the limit, with Zo equal to zero, A' equals $-A$ and shows that the incident waveform will be entirely reflected with a 180" phase change. When the load impedance is infinite, A' equals A, and the entire waveform reflects, but this time without a phase change.

As long as the load impedance is purely resistive and does not equal the characteristic impedance exactly, the reflected wave will have at least some portion of its power reflected, with either a 0° or a 180° phase change. Not surprisingly, you will observe a phase change other than 0° or 180" when the load is reactive.

An understanding of how power dissipates in the transmission line explains why a portion of the wave reflects at all.

$$\text{Power} = IV = \frac{V^2}{Z_o} = v\left(\frac{1}{2}LI^2 + \frac{1}{2}CV^2\right)$$

Because the equations developed in the preceding paragraphs are based upon a lossless transmission line, power dissipation only occurs by transmitting power from one section of the cable to the following section. It does not occur by the conversion of electrical energy to heat energy as you might expect in resistive circuits.

In fact, as I have shown, although Z_o is real, P represents the magnitude of power passing any point on the transmission line during a given instant. As long as each subsequent section of the transmission line has the impedance Z_o, the waveform continues to travel down the transmission line undisturbed; each section passes energy to subsequent sections.

The specific V^2/Z_o changes upon reaching a section of cable or termination resistance that does not have the equivalent characteristic impedance. Some portion of the power must be either returned or reflected because a diminished or increased magnitude of power is transmitted to the new load. The 0° or 180" phase is a consequence of the increase or decrease in power transmitted at that point.

Real-life transmission lines do not behave strictly this way. At some point, the cable length becomes significant because the assumption that cables are lossless is not valid. In real cables, both the conductor and the

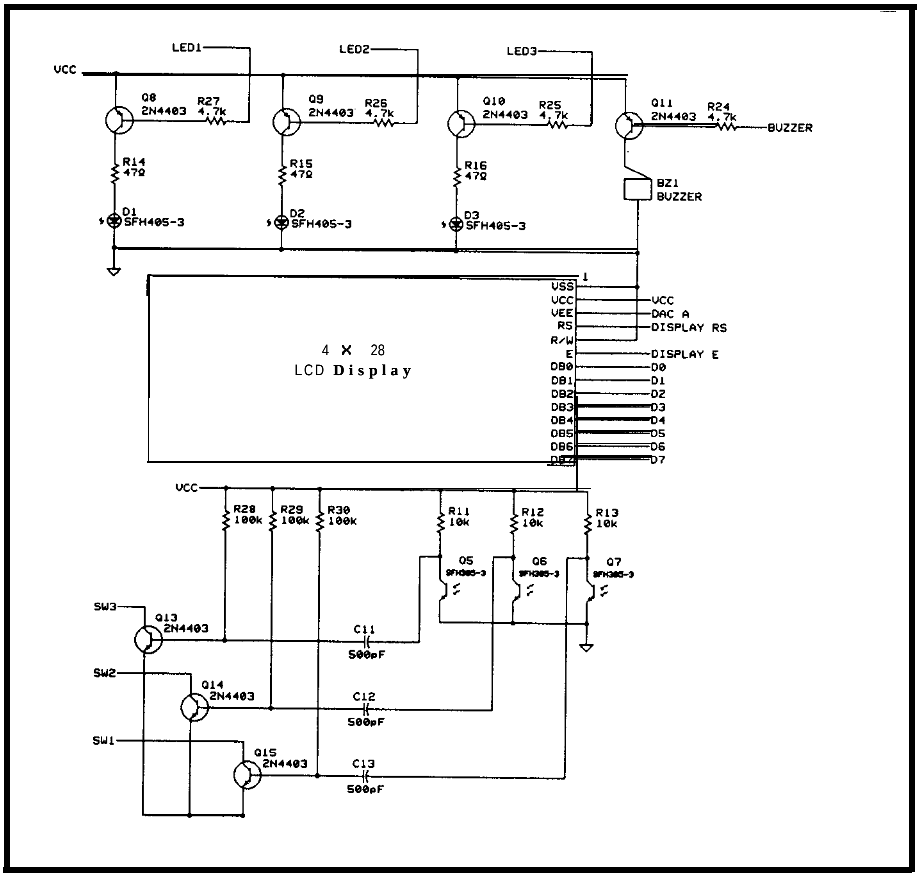


Figure 3—*User input to ihe TDR is through a touch screen interface. Three pairs of LEDs and phototransistors detect a finger pointing at one of three choices displayed on the LCD display.*

dielectrics have resistance. This resistance not only affects the transmitted wave amplitude, but also influences the phase relationships of the frequency components in the transmitted waveform. The TDR is forced to consider this fact when computating termination resistance. Ultimately, this real-life effect limits the length of cable the TDR can test.

HUMAN INTERFACE

We imagined how the TDR would be used and attempted to build the instrument around those purposes. The TDR had to have a simple interface and a small package. Simply stated, the number of basic actions required to operate any instrument or appliance should be as few as possible. Because nearly anyone can memorize three or four operations to perform some function, we restricted the number of keys on the TDR to three and let the software handle the rest.

With the above goals in mind, we decided to make a touch screen interface. That would allow us to present three "buttons" on the display and let the operator select one with the point of a finger. It also allows us to add features later by simply changing the firmware.

For our design, the best combination of keys was two for moving through menus and one for selecting an option. We ensured that any function or operation in the TDR would be compatible with the three-key interface, which unfortunately meant we would have to anticipate any parameter that eventual users may wish to enter and incorporate into the menu.

CABLE TYPES

Computer networks can and do use a variety of cable types. Most of them fall into two categories: coaxial and twisted pair. Each category has its own characteristic impedance. The problem is TDR has its own intrinsic characteristic impedance, which may or may not match the cable under test, so there may be reflections generated at the interface of the TDR and the cable under test.

Rejecting this reflection with software is possible, but the noise margin will be affected. Unfortunately, some portion of the waveform's power also will return before it reaches the cable fault, further reducing the instrument's usable range. Initially, we thought this problem meant a reduction in supported cable types because some form of internal termination was required to perform the measurement. In the end, we decided to cover the two most common types of coaxial cables with a very small relay, which toggles the TDR's internal termination resistance between 50 and 75 ohms.

HARDWARE DESIGN

The electronics for the system consist of two major parts: microcontroller and measurement. The microcontroller electronics include the LCD display, DAC, and IR touch-screen circuits. The measurement electronics consist of a high-speed comparator, a low-power latching relay, and a discrete cable interface circuit.

The microcontroller is an 87C5 1 operating at 12 MHz (see Figure 2). The microcontroller's oscillator also clocks the measurement PAL., U9. The clock to the PAL is enabled and disabled to save power when actual measurements are not in progress.

The three data inputs for the PAL, the LCD display, and dual DAC data inputs populate the microprocessor's port 0. The DAC is the only device actually addressed as a memory device (i.e., using the MOVX instruction).

A single 9-volt battery powers the TDR. Automatic power down is accomplished with the aid of PMOS FET Q3 and NPN transistor Q 12.

When the operator presses the "on" switch S1, the gate of Q3 is pulled low, applying power to the LM3 17 and subsequently the microcontroller. At power-up, all processor port pins go high including P1. 7, which keeps Q3 and Q12 turned on. To power down, the processor simply drops Pl. 7. The sum of the leakages of Q3 and Q 12, which are extremely low and comparable to the self-discharge rate of the 9-volt battery, equals the amount of standby current when the system is off.

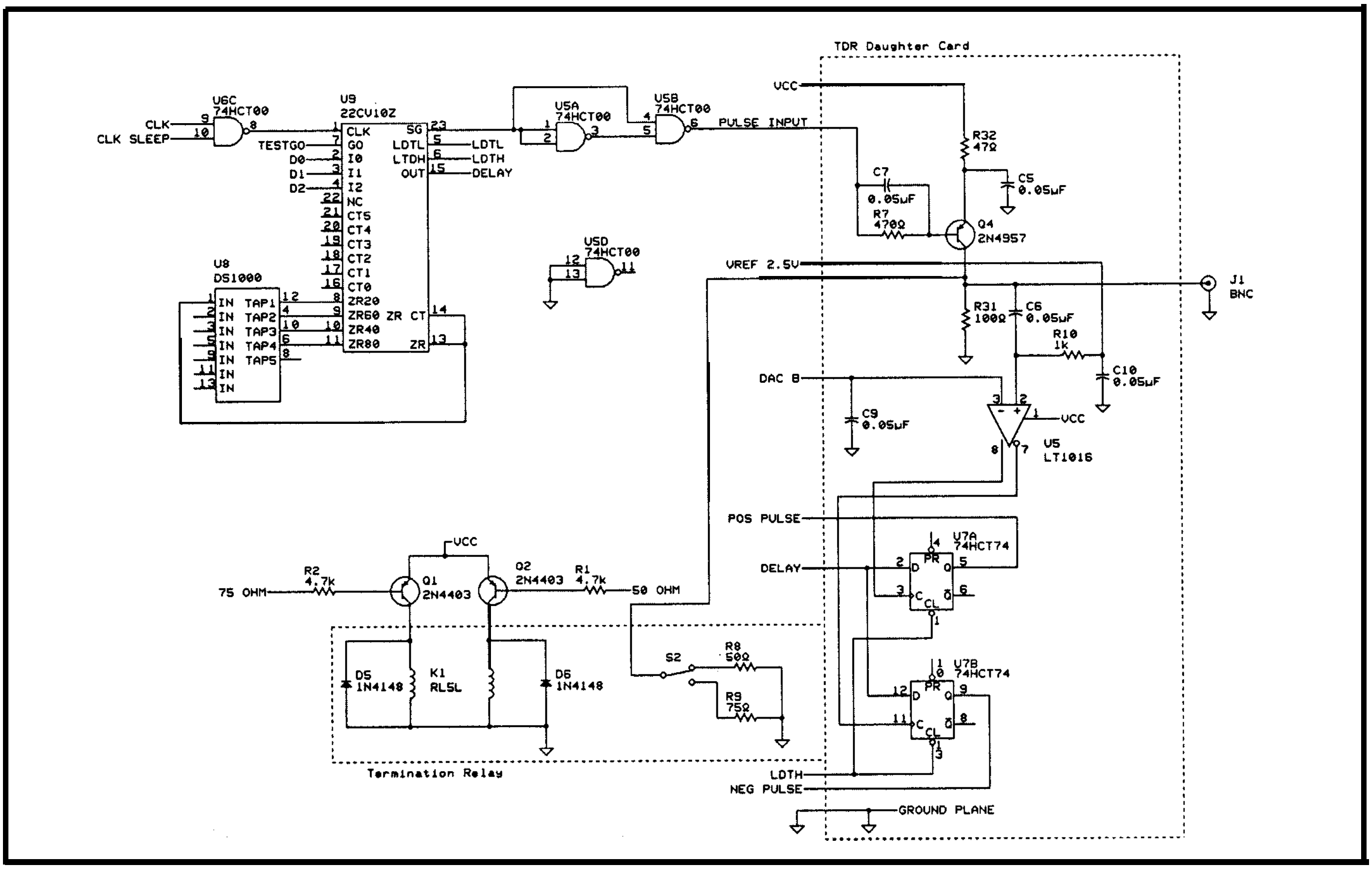
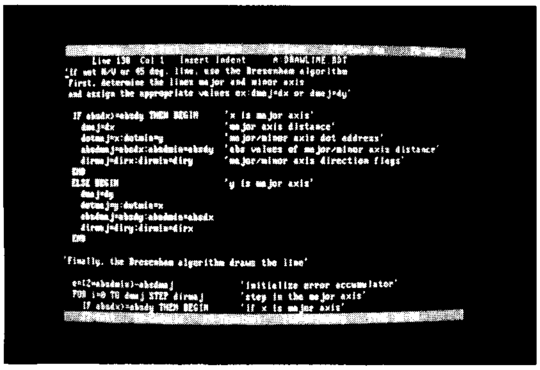


Figure 4—*All the noise-critical* *circuitry* ***is*** *wired* ***on a daughterboard*** with a ground ***plane.*** The ***22CV10Z*** *PAL* chip eliminates a ***handful*** of *circuitry.*



```
     Line 130  Col 1   Insert Indent     A:DRAWLINE.BDT
'If not H/V or 45 deg. line, use the Bresenham algorithm
 First, determine the lines major and minor axis
 and assign the appropriate values ex:dmaj=dx or dmaj=dy'

  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'
    dirmaj=dirx:dirmin=diry      'major/minor axis direction flags'
  END
  ELSE BEGIN                     'y is major axis'
    dmaj=dy
    dotmaj=y:dotmin=x
    absdmaj=absdy:absdmin=absdx
    dirmaj=diry:dirmin=dirx
  END

'Finally, the Bresenham algorithm draws the line'

  e=(2*absdmin)-absdmaj            'initialize error accumulator'
  FOR i=0 TO dmaj STEP dirmaj      'step in the major axis'
    IF absdx>=absdy THEN BEGIN     'if x is major axis'
```

See us *at the* Embedded Systems ***Conference—Booth #836***

THE LCD

The LCD display is a standard 4-line by 20-character module [Figure 3). The menu program uses the top two lines to display measurement data and menu information, and it uses the last two lines to display soft-key data for the touch-screen keyboard.

DAC A of U3 provides LCD contrast control *VEE* through op-amp follower U4c. The DAC voltage reference is provided by D4, a micropower 1.25volt bandgap reference. The reference is buffered by op-amp follower U4a and also provides a 2.5volt reference to the measurement circuit through amplifier U4b.

TOUCH-SCREEN LED CIRCUITRY

An infrared light across the face of the display, in a configuration where the operator's fingers interrupt it, implements the touch-screen display.

Dl, D2, and D3 provide the IR light to phototransistors Q3, Q4, and Q5, respectively. Transistors Q8 through Q10 drive the LEDs at 70 mA for brief periods of time. *P1.0,* P1.l, and *P1.2* provide the control signals through resistors R25, R26, and R27.

The phototransistors operate at low gains using small collector loads of 1 k to minimize ambient light swamping further. The transistors are AC coupled to transistor switches Q14 through Q15 by a differentiating network consisting of Cl3 and R28 through Cl 1 and R30, respectively. These transistors provide microcontroller input *P1.3*, P1.4, and *P1.5.*

When an LED is turned on, its respective phototransistor conducts about 2 mA with a fast falling edge. Differentiating capacitors pass this fast edge with a time constant of 50 µs. The pulse turns off the NPN transistor, producing a 50-µs high on the processor's input for each falling edge on its LED. The software samples the phototransistor outputs briefly after it turns on an LED. If it sees a high input, then the light from the LED must be reaching the phototransistor, so a "key" must not be pressed.

MEASUREMENT ELECTRONICS

The maximum resolution in termination resistance is limited by

Listing 1--The *TDR PAL is defined using a high-level description to let the compiler make better use of the part and to reduce the* number *of outputs.*

```
/********************************************************************/
/*                                                                  */
/* TIME DOMAIN REFLECTOMETER PAL                                    */
/*                                                                  */
/* It does the following:                                           */
/*   6 bit programmable down ctr. ld in two 1/2's, ldth,ldtl        */
/*   zr_ct goes high at zero                                        */
/*   synchronizes go output falling edge of go in and clock         */
/*   multiplexes dallas delay generator on inputs                   */
/*                                                                  */
/********************************************************************/

TDR (in clk,i2..0,ldtl,ldth,go,zr,zr20,zr40,zr60,zr80; /* inputs */
        out dlout;
        reg ct5..0 sg,zr_ct)                     /*  registered outs */

'* define relationships- begin */
 * -----------------            */


/* define groups                */
/* -----------------            */
group i[i2..0];
group ct[ct5..0];
group mux[i2..0];
group d[zr80,zr60,zr40,zr20,zr];

/* clocks                       */
/* -----------------            */
sg.ck = clk;
zr_ct.ck = clk;
ct[].ck = clk;

/* do output enables            */
/* ---------                    */
sg.oe = 1:
zr_ct.oe = 1
dlout.oe = 1
ct5.oe = 1:
ct4.oe = 1;
ct3.oe = 1;
ct2.oe = 1:
ctl.oe = 1:
ct0.oe = 1:

/* have to do this to get feedback */
/* do aclr and spre                */
/* -----------------               */
sg.aclr = 0;
zr_ct.aclr = 0:
sg.pre = 0;
zr_ct.pre = 0;
ct[].aclr = 0;
ct[].pre = 0;

/* define mux action, load low and high and count */
/* -----------------                               */
dlout = d[mux[]];             /* mux high level construct         */
if  (go==1){                  /* if go high wait for load or hang ● /
   if (ldtl==1 && ldth==1){ /* if both loads high just stay here */
      sg = 0;                 /* sync go = 0. no pulse            */
      zr_ct = 0;              /* zr_ct=0, input to delay line     */
      ct[] = ct[]:            /* don't decrement counter- hang    */
   }
   else {                     /* if one or more loads are low then */
    if  (ldtl==0){            /* if load low. load in to low ct    */
```

(continued)

the minimum signal-to-noise ratio, so critical portions of the measurement circuits are located on a small, very tightly hand-wired daughter board (see Figure 4) with a ground plane. These key circuits include the termination relay, the comparator, and the D flip-flop. The logic board contains the less-critical portions of the measurement circuit.

The heart of the measurement circuit is U9, a low-power, high-speed PAL device (ICT 22CV10Z). It reduces the chip count by as much as four MSI logic devices and makes fitting all the electronics into a small package possible. This device orchestrates all the high-speed operations and timing done by the system. Listing 1 presents the PAL's defining code.

Outputting a time delay to the PAL begins a measurement. The PAL includes a 6-bit presettable down counter, an B-to-1 digital multiplexer, and a synchronizer circuit. Only five of the eight multiplexer inputs are used, and they are connected to the delay line chip U8.

The input to the delay line is the zero output of the 6-bit counter (*ZR_CT*). The counter is clocked at 12 MHz (83.3-ns period) and followed by a selectable multiplexer delay of 0, 20, 40, 60, or 80 ns to produce the delay output. The PAL operates through three data inputs that have multiple uses, depending on the state of three other inputs: load high (*LDTH*), load low (*LDTL*), and Go (GO).

When *LDTH* goes low, the PAL preloads the three high bits of the counter. *LDTL* performs a similar function for the three low bits of the counter. The static value left on the three inputs selects the multiplexer routing. Note that the zero delay output (*ZR_CT*) comes out of the PAL and back into the PAL multiplexer zero delay input (*ZR*). Logically, the PAL could do this routing internally, but having it do so would skew the propagation delay for the zero delay output.

The microprocessor brings the PAL GO input low to begin a measurement. This signal is synchronized in the PAL with the next clock edge to output a synchronous GO output (SG).

Listing 1—continued

```
      ct5 = ct5:              /* leave high bits alone               */
      ct4 = ct4;
      ct3 = ct3
      ct2 = i2:
      ctl = il;
      ct0 = i0;
    }
    if (ldth==0  {            /* if load high. load in to high ct */
      ct5 = i2;               /* leave high bits alone               */
      ct4 = il;
      ct3 = i0;
      ct2 = ct2:
      ctl = ctl;
      ct0 = ct0;

   }
}
else {                        /* go is low- let's jam                */
 if (ct[]>0) {                /* if still count loop decrementing */
   sg = 1:                    /* put out sync go on first clock      */
   ct[]--;                    /* decrement counter                   */
     if (ct[]==1)             /* on last count bring zr_ct high      */
      zr_ct = 1;
     else
      zr_ct = 0:              /* else zr_ct is low- to delay line */

  else {
    if  (go==0){              /* if at zero then keep zr_ct 1        */
      zr_ct = 1;              /* leave count and sg atl              */
      ct[] = ct[];
      sg = I:

  }
}
/* put part definition */
/* ----------------     */

   putpart("P22CV10Z","tdr",

/*      1  2  3  4    5    6  7    8     9    10    11  12             */
      clk,i0,il,i2,ldtl,ldth,go,zr20,zr60,zr40,zr80,GND,

/*     13    14     15  16  17  18  19  20  21  22  23  24             */
      zr,zr_ct,dlout,ct0,ctl,ct2,ct3,ct4,ct5, _, sg,VCC)

/* end of definitions */
/* end of module      */
}
```

The rising edge of SG is applied to the glitch monostable circuit formed by NAND gates U6a and U6b. The output of this circuit goes low for about 8 ns on the rising edge of SG. This pulse drives the base of Q4.

When Q4's base is pulled low, it generates the very fast pulse at the BNC test cable output. C6 fixes the high-level output impedance of the TDR at a very low value. Once Q4 is turned off, the setting of latching relay K1 and the parallel combination of R8, R9, and R3 1 determine the impedance the line sees. This impedance is either 50 ohms or 75 ohms depending on the setting of K1.

Note that K1 is a permanent-magnet latching relay briefly pulsed by the microprocessor to change state through Q1 and Q2, although it draws no quiescent power.

The output connector is also capacitively coupled to U5, an LT1016 [a very high-speed comparator). This device has a propagation delay of 7 ns and specifically operates on a single supply. The comparator does have a limited common mode range of 1.25–3.5 volts when run on 5 volts.

To push the signal into this range, the positive input is biased to 2.5 volts by doubling the voltage reference, which brings it to the approximate center of the common mode range. *DAC B* and amplifier U4d provide the negative input to the comparator. The range of this input is O-3.95 volts. The DAC resolution is about 16 mV for measuring the height of the returned waveform.

The comparator has complementary outputs connected to the clock inputs of U7a and U7b. Two flip-flops eliminate the pulse-width uncertainty associated with measuring negative returning pulses. The microprocessor samples the flip-flops' outputs at the end of a trial measurement at P2.0 and P2.1.

The PAL's *LDTL* input clears the flip-flops at the beginning of a trial measurement. They are set after the return of a pulse if the D input provided by *DL_OUT* is present before the comparator senses the pulse.

The measurement circuit operates by injecting an extremely brief pulse [about 10 ns) onto the cable under test. This pulse travels down the cable and reflects from any discontinuities (changes in impedance) in the cable. These reflections return to the TDR and are captured by a very fast comparator, which clocks a D flip-flop.

A precisely delayed signal derived from the original output provides the D flip-flop's input. The PAL generates the delay, and a microprocessor can vary the delay line from 0 to 5.5 µs in 20-ns steps. This time resolution leads to a distance resolution of 6 feet for typical cable types and a maximum cable length of 1700 feet.

TERMINATION RESISTANCE

Pulse amplitude measurements are made by adjusting the threshold of the comparator using a DAC until return pulses are no longer seen. The reflection coefficient is calculated by comparing the transmitted pulse height with the returned pulse height. If you assume the cable is lossless, then calculate the termination resistance using the previously derived formula

$$Z_L = Z_o (1 - V_i/V_o) / (1 + V_i/V_o)$$

Assuming a cable is lossless does simplify matters, but it leads to large errors in the calculation of termination resistance for long cables. The actual attenuation coefficient per 100 feet depends on cable type and quality. Several standard nomographs are typically used by engineers for correcting reflection coefficient for line loss.

The TDR handles this problem by first determining the cable's length and the uncorrected reflection coefficient using the algorithm I've described. It then corrects the reflection coefficient for line loss based on the measured length before using the injected pulse amplitude in the previous equation.

The voltage attenuation constants for *RG-58* and RG-59 are stored in firmware. They are approximately 3% per 100 feet of length for both RG-58 and RG-59. Unfortunately, the attenuation is not linear. Though the TDR makes this simplification, it is able to compute termination resistances accurate to about ±1% for cables up to 100 feet. The accuracy of the termination resistance readout gets worse from here because the attenuation effect was empirically characterized out to 1700 feet. It is probably in the 10% range.

SOFTWARE DESIGN

The software for the TDR is very simple and straightforward with the exception of the menu system and the integer math package. We used the menu system to make the TDR easy to use and to permit some reasonable level of expansion and improvement. The integer math package affords some computational capability, but keeps code size within the constraints of the 4K maximum of 87C51.

The goal of the menu system is to provide a b-treelike structure through which menus and submenus can be called for and returned to in a straightforward fashion. We wanted to add, move, and remove menus and processes with ease. For the most part, we were successful.

Each menu item is defined within a menu structure. This structure may have an unlimited number of items and is terminated by a null. Each menu item consists of three words.

The first word is a pointer to the item's display menu string. The second word has the address to jump to if that menu item is selected. The last word is defined only when a submenu is to be loaded, in which case it holds the pointer to the submenu's data structure.

The menu system needed several routines to manipulate the data structure to operate. NXTMNU is a function created to load a submenu. It saves the calling menu's address in a *parent* menu stack, then it loads DPTR with a pointer to the new menu structure.

The function LASTMNU recovers the parent menu's table address and places that value into DPTR. UPDATE_MENU scrolls menu items through the display. PRINT_MENU updates the

Listing 2-The touch screen allows easy changes and additions to the menuing system.

```
; Main Menu Strings & Prompts
;---------------------------
MAINMNU: DB     '***** MAIN MENU ****'.OH
TESTCBL: DB     '1. Test Cable       '.OH
TESTTYP: DB     '2. Set Test Type    '.OH
CBLTYP:  DB     '3. Set Cable Type   '.OH
SHW_PAR: DB     '4. Show Setup       '.OH
ADJDSP:  DB     '5. Adjust Display   '.OH
;
;data structure for main menu
;
MMENU:   DW     MAINMNU      ; PTR to main menu string
;test cable
         DW     TESTCBL      ; PTR to function title
         DW     RUN          ; PTR to function routine
         DW     OH           : No data this time
;test type
         DW     TESTTYP      ; PTR to submenu title
         DW     NXTMNU       ; PTR to Nextmenu program
         DW     TTMENU       ; PTR to submenu data structure
;cable type
         DW     CBLTYP       : PTR to submenu title
         DW     NXTMNU       ; PTR to Nextmenu program
         DW     CMENU        ; PTR to submenu data structure
;show parameters
         DW     SHW_PAR      : PTR to submenu title
         DW     SETUP        ; PTR to Nextmenu program
         OW     OH           ; PTR to submenu data structure
;adjust display
         DW     ADJDSP       ; PTR to submenu title
         DW     NXTMNU       ; PTR to Nextmenu program
         DW     ALMENU       ; PTR to submenu data structure
:end of main menu- zero
         OW     OH           ; nul establishes end of structure

; end of data structures for main menu

;------------------------------------------------------------
; NXTMNU: A task that saves the current value in DPTR
      and reloads DPTR with a new menu structure.

      Registers used

      DPTR:      Loaded with new menu table address
      ACC:       Scratch register
      B(FO):     Scratch register
      OH:        Scratch register
                 -------------------------------------------
;
NXTMNU:  PUSH   ACC
         PUSH   OFOH         ; B register
         PUSH   OH             RO
         MOV    A,MENUPTR    ; This section of code gets the
         RL     A                pointer to the menu stack,
         ADD    A,#MNUSTCK   ;   saves the current OPTR in
         MOV    RO,A             the menu stack, and
         MOV    @RO.DPH      ;   the menu increments
         INC    RO           ,   stack pointer
         MOV    @RO.DPL      ;
         INC    MENUPTR      ;
         MOV    A,OPPTR      ; This section of code uses the
         ADD    A,#MDATA     ;   current menu table to obtain
         RL     A            ;   the submenu table address.
         MOV    B,A          ;   It then loads that address
         MOVC   A,@A+DPTR    ;   into DPTR.
         PUSH   ACC
```

(continued)

Listing 2—contiued

```
        MOV    A,B
        INC    A
        MOVC   A,@A+DPTR ;
        MOV    DPL,A
        POP    DPH

; Once a new table has been  loaded, schedule the screen
; to be repainted.

        SETB   DISPFLAG    ; Set "print option' flag
        SETB   TTLEFLAG    ; Set "print title' flag
        MOV    OPPTR,#0H   ; Reset pointer to top of menu
        INC    DPTR          Point to the first item
        INC    DPTR            in the table.
        POP    0H
        POP    0F0H
        POP    ACC
        RET
```

display with the currently selected menu. Listing 2 is an excerpt from the TDR's code.

A small collection of functions that operate on 32-bit variables performs all math on the TDR. The TDR needed only an addition, subtraction, multiplication, and division function.

THE COMPLEAT TDR

The instrument we've described works in its most basic form, but we think there is plenty of room for improvement before any reasonable production could be considered. It would be fun to extend the number of cable types supported and maybe even add optical cable support. The biggest leap in functionality would be to make the TDR, or something like it, a more integrated part of a network. It could be a device that resides semipassively on the network and is occasionally polled by a PC via an RS-232 connection. Use your imagination. ❑

Brian Kennez is a project engineer at Science Applications in San Diego, Calif. He has developed several fixed and hand-held microcontroller-based instruments used in vibration and radiation monitoring and analysis.

John Wettzoth is employed as the Chief Engineer of Science Applications' Military Products Division in San Diego, Calif. He also owns Tzavtech, a sole proprietorship involved in electronic instrumentation development and HP-48 calculator data acquisition.

IRS

404 Very Useful
405 Moderately Useful
406 Not Useful

Serial I/O on the IBM PC

FEATURE ARTICLE

Jim Schimpf

As anyone who's tried to write serial support code for an IBM PC knows, the BIOS routines just don't cut it when trying to keep up with multiple high-speed ports. Jim's package replaces the BIOS with capable routines.

Connecting various devices to an IBM PC or compatible via the serial I/O is very common, and done so with everything from a manufacturer's microprocessor development board to your latest widget. The serial interface is extremely convenient, requiring only a few wires and usually stretching far enough to make "remote control" mean something. The PC has the hardware to make this task easy, but the standard BIOS for IBM and clone PCs does not support anything but polling-type I/O, which means you are guaranteed to drop characters unless you use low data rates.

A large number of software I/O libraries are available that can make the PC serial I/O live up to its hardware. These libraries are all very good, but if you are like me and have more time than money, the simple full-duplex serial I/O package I describe here is all you need to make your project hum.

The code for SERCOM is written in Microsoft Quick C and can be easily added to your program. It can open multiple serial channels [your hardware limits the number] and each channel supports full duplex with ring buffers on both input and output. Various status checkers are present, so your program can monitor the arrival of each character or only be notified when a complete line is received. This way, even a program running on a slow 8088 can monitor two 9600-bps channels. Also included is a sample terminal emulator designed to talk to a popular microprocessor development board.

THE SERCOM SERIAL I/O PACKAGE

SERCOM has ring buffers on both input and output, which means the program using the serial input does not have to monitor the serial I/O line constantly. This arrangement is a major failing of the polling BIOS serial I/O. If your program is busy doing something else when a burst of characters arrives, you will get only a few characters and the rest will be dropped. The solution to this polling problem is to lower the data rate so the program will be able to check the line before another character can arrive.

Using SERCOM routines is like using the file operations of stdio.h. In this case, you include the header file SERCOM.H in all the modules of your program that you want to use the serial I/O routines. Listing 1 shows a skeleton for serial I/O.

Again like file operations, these routines let you have more than one open at a time. The supplied version of SERCOM limits you to two because standard PCs only support two ports. If you want to open more than two ports at a time, you will have to add their register and interrupt locations. You may open as many ports as you have hardware.

Table 1 shows the functions available when the port is opened. What each function does is obvious, but the comments in the SERCOM.C program source code provide descriptions of them.

Table 1 also shows a number of functions used a little less often. If you start your serial I/O program on the PC, then connect it to your remote system; plugging in the cables will probably generate a few garbage characters. In a case like this example, a call to clean_ser() ensures the input buffer is empty.

MTERM: A DEMONSTRATION PROGRAM

Included with the source code is a simple terminal emulator program called MTERM. I developed this program to talk to the Motorola EVM board and also included the capability to download S19 files to it.

Listing 1-A *skeleton C program shows how a serial port is defined, opened, and closed.*

```
#include <stdio.h>
#include <bios.h>
#include "sercom.h"

main()
{
    COM *ser;

/* Open the serial I/O port using COM1, 9600 bps, 8 data bits,
no parity, a 1000-byte input buffer, and a 100-byte output
buffer. */

    ser = ser_init(PORT_A,BAUD_9600,1000,100);
    if (ser == NULL) {
        exit(-1); /* Flag error, port not opened */
    }

/* Insert main program here */

/* Close the serial port. Failure to do this means the machine
must be rebooted before the next program run. */

    ser_close(ser);
}
```

Writing a terminal emulator using the functions in SERCOM is simple. At the heart is a loop that monitors the PC keyboard [using kstat()] and the serial I/O line [using serstat()]. If a character is seen on the keyboard, it is read and sent to the serial port. If one is seen on the serial port, it is read and sent to the PC screen. When a key is typed on the PC keyboard, the remote serial device must echo it before it appears on the screen.

MTERM can both capture received data and transfer files. The program easily sends all the received data to and from a file just like it came from the keyboard.

SERCOM INTERNALS

MostofthecodeinSERCOM.Cis very simple, but two areas deserve further explanation: initialization and ring buffers. The initialization [and shutdown) code manipulates a whole range of hardware registers in the serial I/O chip and interrupt controller of the PC. Ring buffers are a useful concept, and if you haven't used them in your code you should try them.

INITIALIZATION

At the start of the SERCOM code are a large number of #defines. Some of these are the constants for the serial I/O channels. They give the address of the controller, the interrupt it is using, and the EOI command. This last command tells the 8259 interrupt controller in the system that the interrupt is handled. Other #defines give the names of the various registers in the serial I/O chip and the interrupt controller.

Where did all these values come from? Easiest are the serial device registers. In original IBM PCs, the the serial chip used was an Intel 8250. While most modern clone PCs use highly integrated chips that include the serial ports instead of individual 8250s, they are accessed in the same way.

To make the use of the device registers clearer, I added the simple macro

```
#define SREG(x) ((unsigned)
  ((unsigned)x + c->com_base))
```

that generates the address of the device registers. This way, because c->com_base is defined in the program context, the address of any device register is simply its offset (the #defined value) plus the base. So you will see things like

```
val = inp(SREG(LCR));
```

This code reads the LCR register using SREG to generate the address and the inp function to read this address.

With this step out of the way, you can look further into initialization. The first step is to set the data rate, parity, and number of bits per word. To do so, I used the Quick C function _bios_serialcom(). Usingthis function was simpler than actually setting the bits in the registers. If your C does not have it, writing an equivalent function is quite easy using the data sheet.

The ring buffers are allocated in the next step. Using the C memory allocation functions allows you to have variable-size buffers, and to have them created when you initialize the serial I/O port. This feature makes the SERCOM package more general. How these ring buffers work and are used is detailed in the next section.

BOOLEAN putser(char ch, COM*c)	Output a character
int putsers(char*str, COM*c)	Output a string
int getser(COM*c)	Get a character
int getsers(COM*c, int len, char'str)	Get a string
BOOLEAN serhit(COM*c)	Check for a character present in the input buffer
BOOLEAN serline(COM*c)	Test for NEWLINE-terminated string in the input buffer
clean_ser(COM*c)	Clear the input buffer
ser_dtr(COM*c)	Control DTR output
ser_cts(COM*c)	Control CTS output

Table 1-The *SERCOM package contains a very basic set of serial port access functions.*

Finally, you have to do the most complicated and delicate operation in the initialization: the setup of the interrupt vector and enabling serial **interrupts.** Carefully complete this step's actions in the proper order because you are changing values inside the operating system. The actions done here are

1. Save the old interrupt vectors.
2. Insert the new ones.
3. Notify the interrupt controller (8259) that interrupts are on.
4. Turn on the serial I/O interrupts.

All the code to do steps #1 through #4 is in the routine ser_init0(). Separate sections in this routine handle the setup of COM1 and COM2. These sections set the unique constants in the COM data structure and set the different interrupt vectors. If you add another port, you will have to rewrite this routine, adding sections to support your hardware. Listing 2 shows the code to do all four steps. Note that you must turn the interrupts off before you do anything. This way, you won't be changing a vector while the system processor is trying to use it. The interrupts are turned off and the vectors are defined by Quick C; other C compilers will have similar but different conventions. Steps #3 and #4 are then completed, and the interrupts are turned back on.

In the above section, notice that you turned on the interrupts for serial input but not for serial output. This omission was made simply because there isn't any data to output yet; thus, the 8250 transmit buffer is empty. The 8250 is set to generate a transmit interrupt on an empty transmit buffer, so that is ***not*** what you want now. The code to transmit data [putser( )] turns on the interrupt whenever new data is to be sent and turns the interrupt off when there is no more.

Shutting down the serial I/O system is just the reverse of starting it. The most important functions are turning off the serial interrupt and removing and restoring the original interrupt vectors. If you don't complete these steps, then a random serial interrupt after your program finishes will use the wrong vectors and cause a system crash.

RING BUFFERS

Each serial port opened by SERCOM is assigned its own set of receive and transmit buffers (see Figure 1). Looking

Listing 2—*Hooking* he ***serial interrupts into*** he system and ***setting*** up he ***8250*** must be done ***properly to avoid*** a ***system crash.***

```
/* Set the interrupt vector for COM1 */
    _disableO;

/* Read old interrupt and set new one */
    coml = c;
    c->old =_dos_getvect(c->int_number);
    _dos_setvect(c->int_number, int_serl);
    break:

/* Enable IRQ4/COM1, IRQ3/COM2 on 8259 interrupt controller */
    val = inp(INTC_MASK);
    val &= c->irq_mask;
    outp(INTC_MASK, val);

/* 8250 hardware setup */
    val = inp(SREG(LSR));         /* Read and discard status */
    val = inp(SREG(RBR));         /* Read and discard data */

    val = inp(SREG(LCR));         /* Rst DLAB for IER access */
    val &= 0x7F;                  /* 01111111B */
    outp(SREG(LCR), val);
    outp(SREG(IER),1);            /* Enable Data Ready INT*/
    outp(SREG(MCR),0xB);          /* Enable OUT2, RTS, and DTR */

/* All done, restore interrupts to processor */
    _enableO;
```

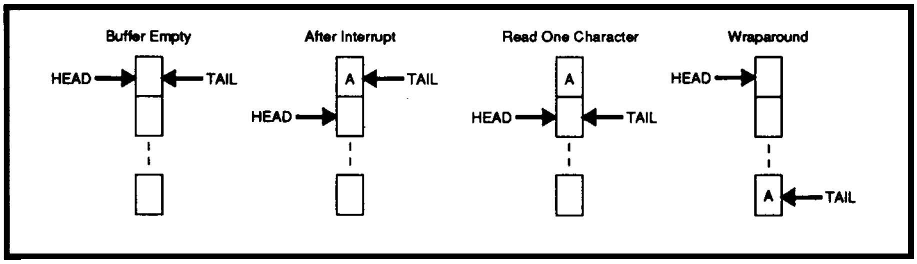


Figure 1-A *ring buffer* uses *a pair of pointers to keep back of the beginning and the end of the buffer.*

at the receive channel, you will see that the head pointer stores a character each time the hardware generates a received character interrupt. The character will be read from the hardware, inserted into the ring buffer at the head pointer location, and the pointer is advanced. The entire process happens in the background.

You can then get one character from the buffer by a SERCOM call. The character is taken from the tail pointer location, and the tail pointer is advanced. Each time a character is added to buffer, the head pointer is advanced. If the pointer hits the physical end of the buffer, it is reset to the start (which is where the *ring* in the *ring buffer* comes from). When the head and tail pointers point to the same location in the buffer, it is either full or empty. The program keeps a count of the number of characters in the buffer at all times, so distinguishing between full and empty is easy. The ring output buffer is run the same way except the noninterrupt part of the program puts in characters at the head pointer and the interrupt driven output removes them at the tail pointer.

TIME UP FRONT

Serial support routines may seem like a small matter in the scope of a much larger program. However, done incorrectly, they can bring a whole system down due to an unreliable serial port. Taking some time up front to ensure solid serial port operation will certainly pay off later. ❑

I *want to thank C. J. Dunford for the original code to DUMBTERM, a simple terminal emulator.*

Jim Schimpf works for an instrumentation firm designing hardware and software for highway measuring instruments.

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

Add Text Overlay to Any Video Display

SPECIAL SECTION

Bill Houghton

All the latest VCRs have on-screen programming in an effort to simplify the notorious task. Now you can have your HCS II or any other computer send you messages while you're watching your favorite TV show.

Suppose, for the moment, you've built and installed the **HCS II home control** system described primarily in issues 25 and 26 (February/March and April/May '92) of the Computer *Applications Journal. In* issue 27 (June/July '92), Ed Nisley described an add-on LCD output device as a way to obtain information about the status of the system and its various nodes. It's a nice addition to the network, but useful only when you're near to the display module. What do you do if you're across the room watching TV settled into your favorite armchair? You could get up and venture across the room. Or, if you build the interface described in this article, you could hit a button on your HCS II IR remote and see network information displayed on your TV set overlaid onto the program you are watching.

This article describes an On-Screen Display (OSD) terminal for HCS II (we'll call it "TV-Link" to match the other HCS II modules) that allows color text characters to be displayed on top of a background color video signal. The terminal is built around the Philips/Signetics 87C054 OSD microcontroller.

FEATURES OF 87C054

The 87C054 is an 80C51-based microcontroller designed to provide an advanced OSD for TV and video applications. It can produce characters in eight foreground and eight background colors. In addition, the background color can be removed, showing through the original video. It also has nine pulse-width modulator outputs for controlling analog functions. Similar to a standard 80C5 1, it has 28 digital I/O pins, two external interrupts, and two timer/counters. RAM and ROM spaces on the 87C054 are larger than the 80C5 1: 192 bytes of RAM and 16K bytes of EPROM. (The OSD has *additional* RAM and EPROM areas that are not part of the normal 80C5 1 memory map.)

One unique feature of the 87C054 is what Signetics describes as a "soft-

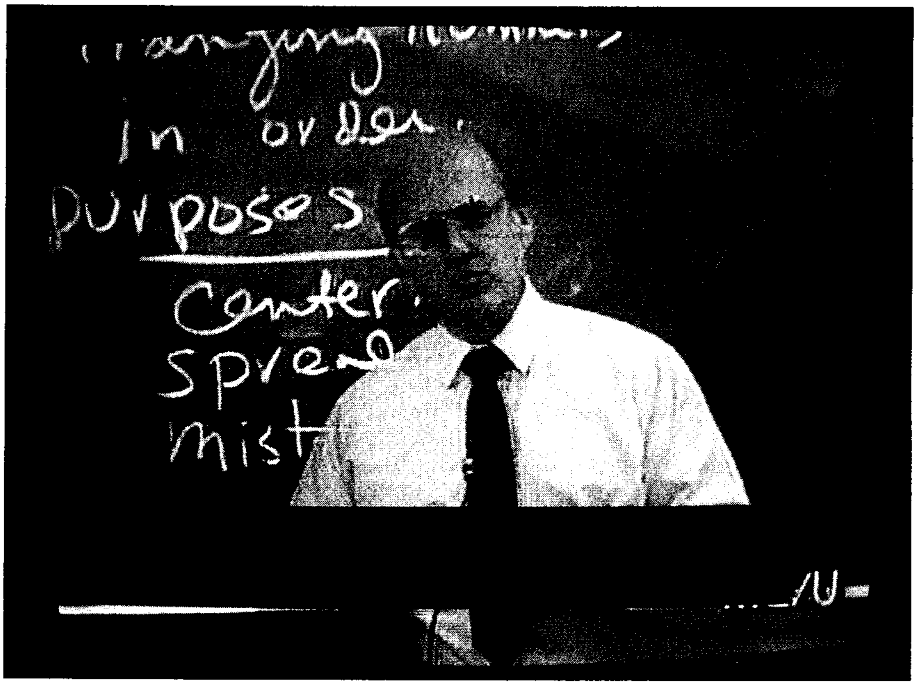


Photo I--The *W-Link can be used* ***by your computer*** *to overlay messages on any* ***video*** *signal.*

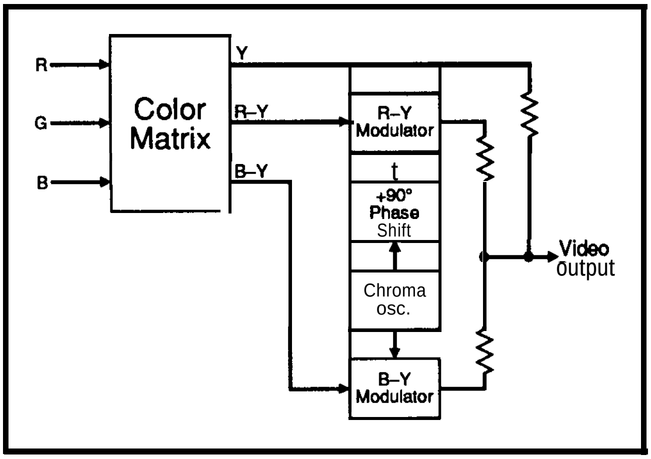


Figure I-Converting km *RGB* b *NTSC involves modulating and* adding *difference signals.*

ware ADC.” This ADC consists of an internal 4-bit DAC that feeds one input of a comparator. The other comparator input can be connected to one of four I/O pins. The output of the comparator is tied to a status bit in a register that is testable by software. A TV set often uses this logic for measuring the AGC voltage during tuning. A real-time clock and other low-precision analog measurements can also use it as a zero-crossing detector.

The OSD of the 87C054 consists of a 128-character RAM array (OSDRAM), a 64-character font EPROM, a video clock oscillator, and the OSD logic. The OSD logic accepts horizontal sync (*HSYNC*) and vertical sync (*VSYNC*) signals and provides three digital video outputs for character information. In the datasheet for this part, these outputs are called *VID0, VID1,* and VID2, but they can also (and perhaps better) be thought of as *RED, GREEN,* and *BLUE.* A multiplexer control output is also present to indicate when to display character data or original video information.

The video clock oscillator provides timing for the character dots. In most applications, this oscillator is simply an LC tank circuit connected to the *VCLK* pins. The frequency controls the character width. One nice feature is that the video clock is killed at the leading edge of *HSYNC* and restarted at the trailing edge of *HSYNC,* which causes the video clock to start in the same phase on every line, ensuring the dots align vertically from one scan line to the next.

The character font stores the binary pattern for the individual characters. Characters are 14 dots wide and 18 scan lines high.

The OSDRAM stores the characters to be displayed on the screen along with certain attribute data pertaining to those characters. Once a character has been written to the OSD, no further CPU intervention is required to “refresh” the screen.

Many OSD architectures have been developed over the years for use in the consumer television market. Almost all of them have required fixed character row formats, limiting the designer’s flexibility in designing video menus and screens.

The 87C054 was designed to avoid such constraints, and there are no architectural limits on the number of characters in a row of text or the number of rows of text to a screen. [There are physical limits imposed by the dot clock frequency and the scan rate, of course.)

The *HSTART* and *VSTART* parameters in the *OSORG* (on-screen origin) register define the intial position of the start of the OSD. Once the initial vertical and horizontal positions have been found, the 87C054 will “fetch” characters from the OSDRAM and place them sequentially on the screen. In order to have multiple rows of text, a special character has been defined and is referred to as *NEWLZNE. The NEWLINE* character is much like a carriage return/line feed sequence on a computer in that it terminates the current row of characters and starts a new row of text. One advantage of this architecture is that it eliminates the need to pad display memory with space characters. The fetching and painting of rows of text will continue until either a new vertical sync pulse is detected or until an *END* attribute is fetched along with a *NEWLZNE* character.

NOW FOR THE DETAILS...

Now that you understand the concepts of an OSD operation and the capabilities of the 87C054, focus your attention on the details required to overlay characters onto live video.

The 87C054 OSD has a multiplexer output for switching video sources. Simply switching between the input video signal and the OSD character data would be nice. Unfortunately, you can’t because the input video (from our home entertainment center) is in NTSC format and the character data is in RGB format. (Keep in mind that the goal is to input live video, add on-screen text, and present the result as a video signal at the output of our circuit.)

One solution is to decode the input video into separate red, green, blue, *HSYNC,* and *VSYNC* signals. Then you could perform the multiplexing between video information and character data in RGB format. The resultant signals could then be encoded back into baseband video. If

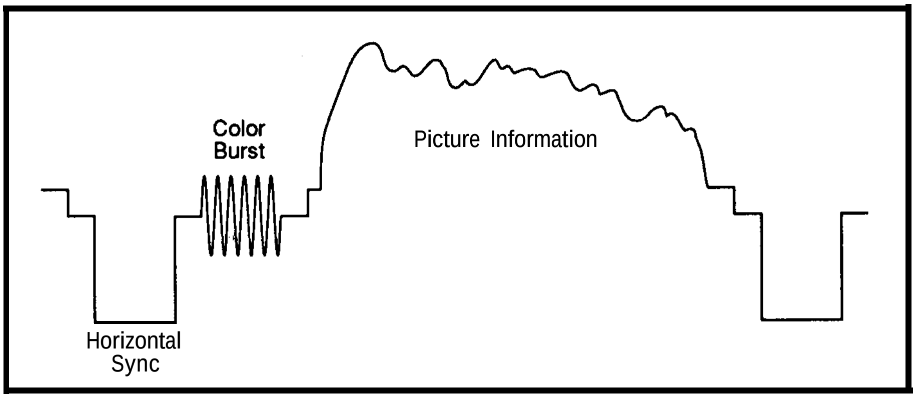


Figure 2-A *typical line of* NTSC *video consists of an initial horizontal* sync *pulse, followed* by *a short color burst signal, hen he actual picture information.*

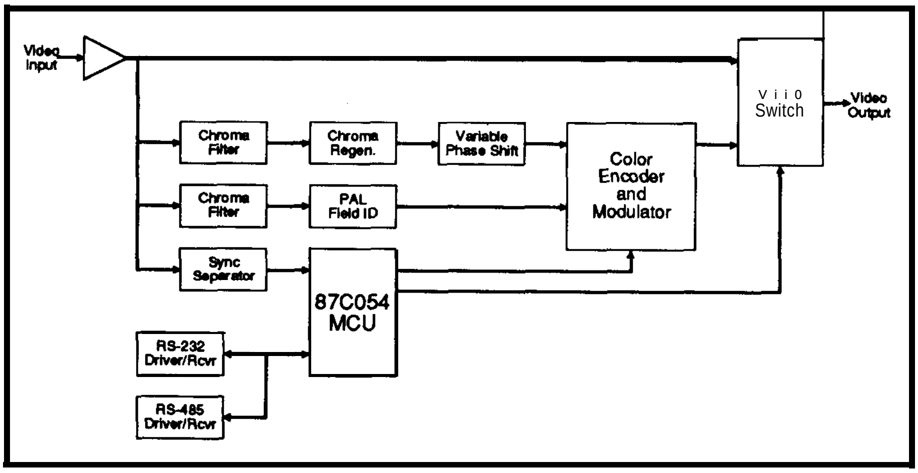


Figure 3—*At* the core ***of the TV-Link is the Signetics 87C054 microcontroller.*** The *unit* ***supports*** *boh* ***RS-232*** and RS-485 ***communications*** *with a host computer.*

there were other reasons for the conversion into RGB, such a conversion would be the way to go. However, the process of converting to RGB and then converting back to video introduces some distortion that could be visible on the screen.

Another solution is to find a way to encode the RGB data from the OSD microcontroller into video. Then you can simply switch between the two sources. Sound simple? The situation gets a little more complex when you consider the issues of making the characters appear with the proper color in NTSC. Reviewing how color is encoded in NTSC is in order.

COLOR TELEVISION

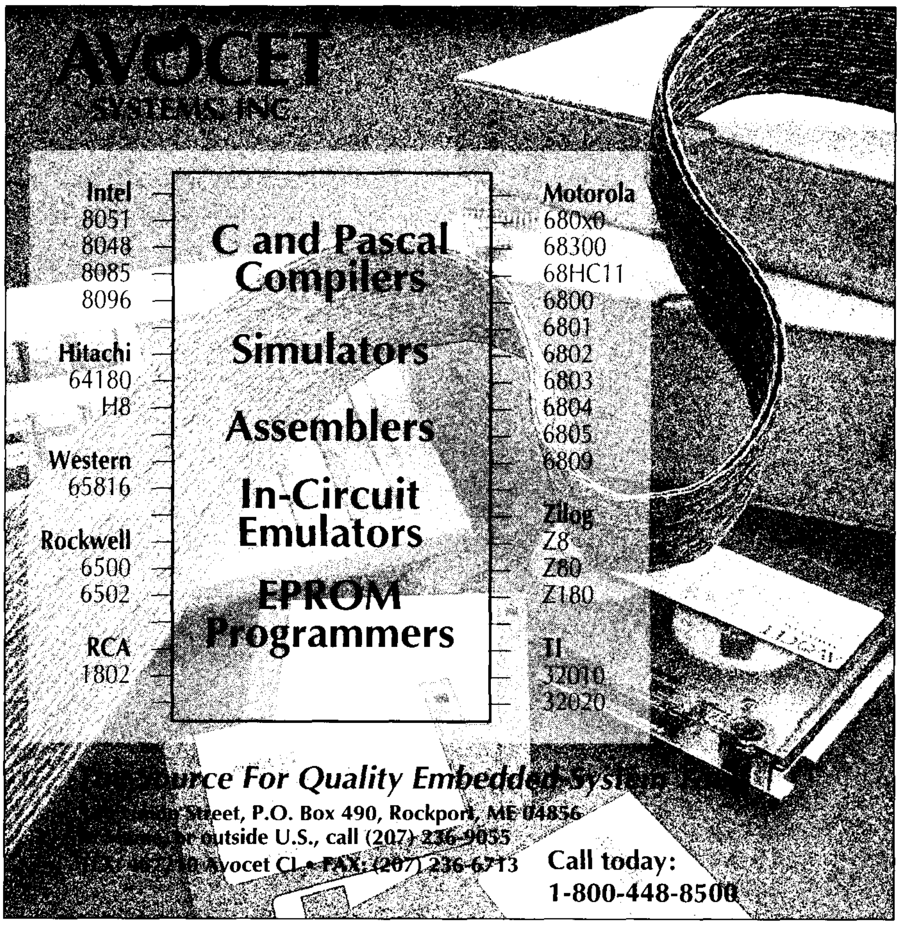
When you first look at video, you often wonder why in the world it was done the way it was. A long time ago, before Americans had ever heard the names of Japanese TV makers, RCA research labs were developing color televeision. One of the requirements imposed on designers by the FCC was that the broadcast signal needed to be compatible with existing black-and-white TV sets and had to be contained within the same bandwidth as a B/W signal. Such requirements meant that some component of the signal had to contain overall brightness information, which is the main reason why they could not simply transmit separate R, G, and B channels within the video bandwidth allowed. To make a very long story short, the engineers involved decided that the scene brightness (which they called "Y" or luminance) could be described by the relationship

$$Y = 0.59G + 0.3R + 0.1\ 1B$$

Someone observed that if they took two copies of the luminance signal and subtracted one copy from one of the colors (say, RED) and did the same with a different color (say, BLUR), the result would be two signals that contained all of the information needed to represent color. These resultant signals, R-Y and B-Y, are the color difference signals.

Now that you have two signals, how can they be transmitted on one RF carrier? The answer they came up with was to modulate one of the signals (B-Y] with an RF carrier. The other signal (R-Y) was to be modulated with the same RF carrier, but the carrier would be shifted by 90”. When the outputs of the two modulators are added together, the result is the vector sum of the two signals, containing both an amplitude and a direction (phase angle). See Figure 1.

We now have a single signal that contains all of the color information. The TV receiver needs just one more piece of information to demodulate this signal. It needs a reference for the carrier used for the modulation, that is, the receiver needs to know where 0° of the color carrier is (in video, this color carrier is referred to as the *chroma* subcarrier). In order to give this reference to the receiver, a small



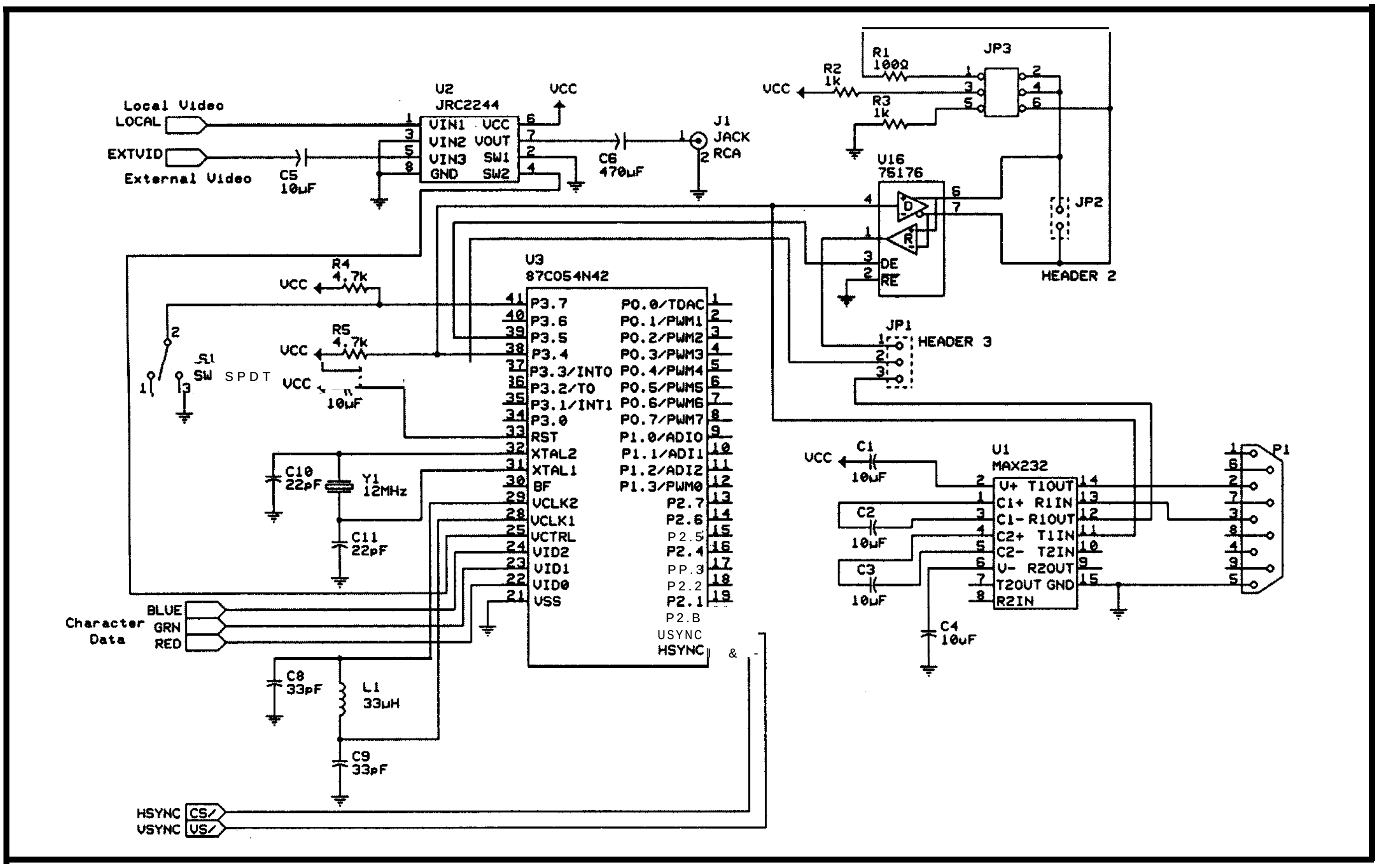
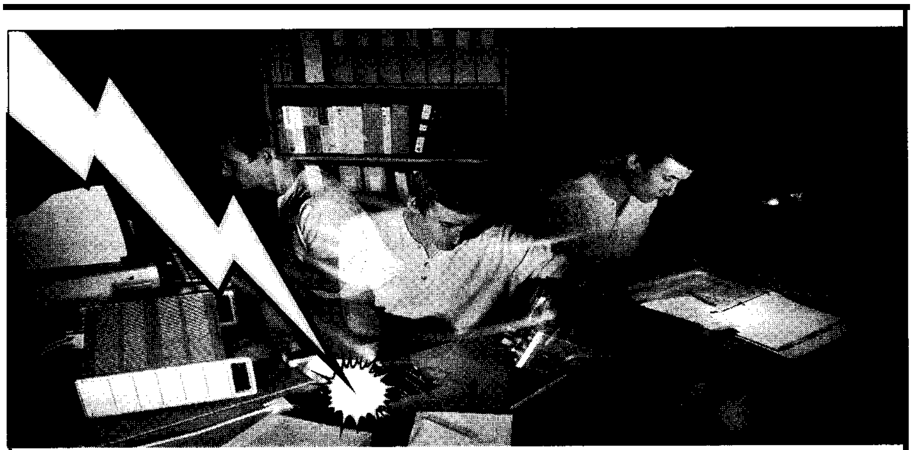


Figure 4a—*The JRC2244* switches **between** he incoming video **signal** and **the** on-screw characters under **control of the 87C054 MCU**.

number, or “burst,” of cycles of the color subcarrier (hence, the term color burst) are transmitted on the back porch of the horizontal sync pulse. In most NTSC systems, this chroma subcarrier has a frequency of approximately 3.58 MHz. A typical line of color NTSC video is shown in Figure 2.

In order to convert the character data from the OSD into NTSC, you will need to sum the data into R-Y and B-Y components. Then you will need to modulate these components with a chroma subcarrier at 0° for B-Y and +90° for R-Y.

One more item to consider. If you have an output video signal composed of a video source with characters overlaid onto it, the chroma **subcarrier** reference [i.e., color burst] present on the output video signal is the color burst provided in the original input video. In order for the receiver/monitor to interpret the color of the OSD characters correctly, the chroma subcarrier used to modulated the **OSD's** R-Y and B-Y components must have exactly the same frequency and



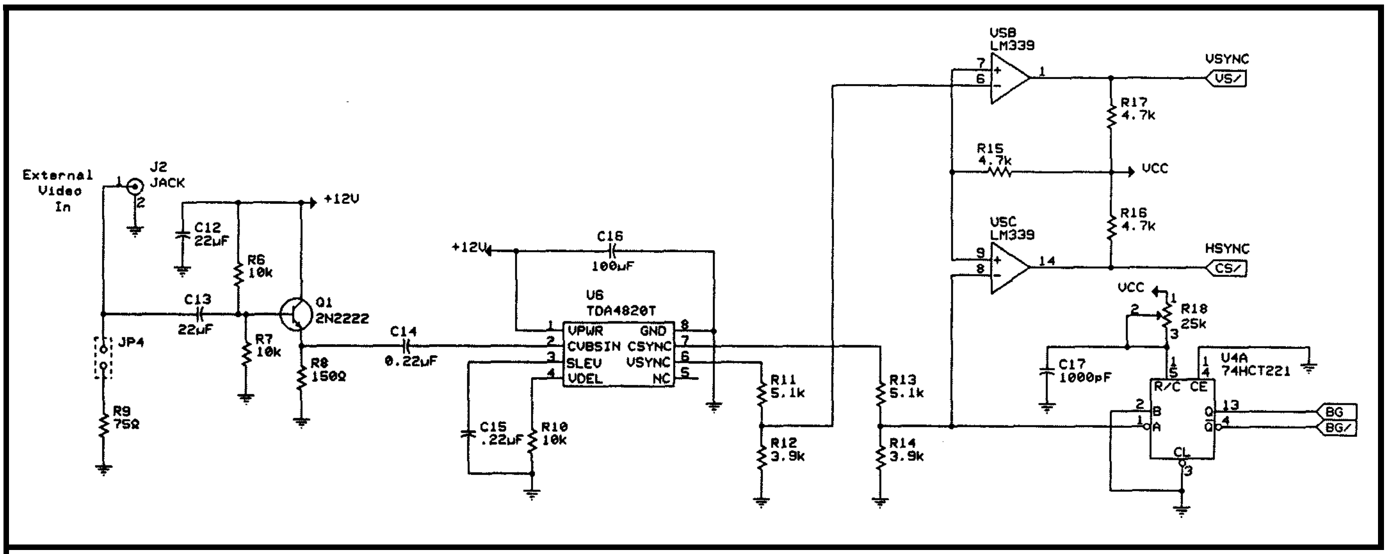


Figure 4b—The TDA4820T sync separator provides the processor with horizontal and vertical sync signals

phase as the color burst on the original video input signal.

Once you get the OSD information in the form just described, you can switch between this "OSD video" and the original input video to produce the final output.

THE TV-LINK HARDWARE SOLUTION

Figure 3 shows a block diagram of the TV-Link, while Figure 4 shows the schematic.

Referring to the schematic, the original input video connects to J2 and is AC coupled into buffer amplifier, Q1. This amplifier provides load isolation between the video signal source and the circuits on the TV-Link board. JP4 is a jumper allowing for a 75-ohm termination resistor to be connected to J2. The output of the Q1 buffer amplifier feeds the sync separator, the video switch, and the chroma subcarrier regenerator circuits.

SYNC SEPARATION

The sync separator consists of U6, a TDA4820T Philips sync separator. The video signal is coupled into the TDA4820T through capacitor C 14, where it is amplified with a gain of 15. The black level clamping voltage is stored in capacitor C14. From the stored black level voltage and the peak sync voltage, the 50% value of the peak sync voltage is generated and stored in capacitor C15. A slicing level control circuit ensures a constant 50% peak sync value regardless of the picture content amplitude provided the sync pulse amplitude is between 50 mV and 500 mV. A comparator in the composite sync slicing stage compares the amplified video signal with the DC voltage derived from the 50% peak sync voltage, producing the composite sync output. Vertical slicing circuits compare the composite sync signal with a DC level equal to 40% of the peak sync signal, producing the vertical sync output. The reduced vertical slicing level ensures more energy for the vertical pulse integration. The slope is double integrated to eliminate the effects of interference caused by noise or line reflections. The value of resistor R10 sets the vertical integration delay time.

The outputs of the sync separator are positive-going signals with peak amplitudes exceeding 10 V. Resistor pairs R11/R12 and R13/R14 serve as voltage dividers for the *VSYNC* and CSYNC outputs, respectively. An LM339 comparator, U5, serves as an inverter for the sync signals because the modulator circuits require active low sync signals.

There is a great tendency with video circuits to make the coupling capacitors very large to pass the low-frequency sync components (60/50 Hz, typically) into low-impedance nodes. The TDA4820T has a moderately high input impedance on pin 2. Because the black level is stored in C14, the value of Cl4 should be kept close to 0.22 µF.

THE 87C054 MCU WITH OSD

The 87C054 microcontroller, U3, accepts composite sync and vertical sync signals from the sync separator and provides RGB digital outputs for character data. The multiplexer control output, *VCTRL,* connects to the video switch, U2, a JRC2244.

Inductor L1 and capacitors C8 and C9 form a video clock oscillator that determines the width of a character font dot. The values of these components are not critical but are typically chosen such that a video dot width is equal to the spacing between scan lines. This oscillator is killed at the leading edge of the *HSYNC* signal and allowed to startup at the trailing edge. Such synchronization causes the oscillator to start at exactly the same point from one scan line to the next, causing character dots to appear in exactly the same spot on each line.

In addition to the OSD functions, the 87C054 also performs network interfacing and protocol tasks. This microcontroller has plenty of performance bandwidth because the OSD logic is self-refreshing and independent of the MCU core.

VIDEO SIGNAL SWITCHING

The JRC2244 video switch, U2, contains three video inputs, two of which are used in this application. One of these inputs, *VIN1,* is capacitively coupled to the OSD video signal. This signal is the 87C054's RGB data after encoding into baseband video.

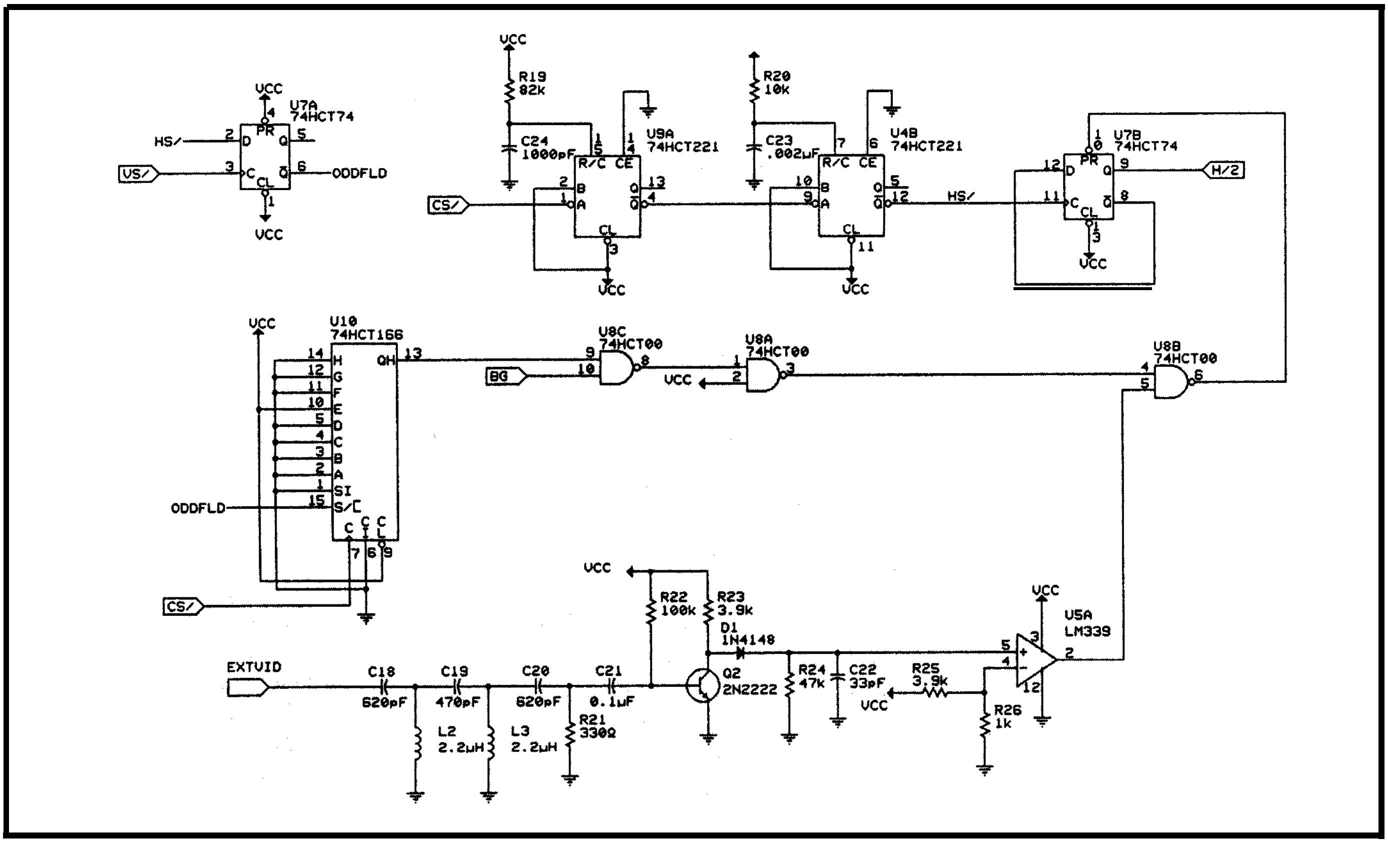
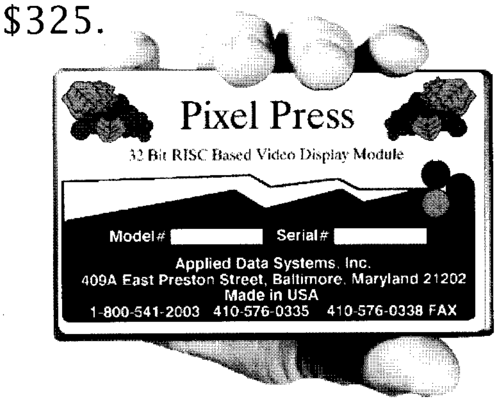
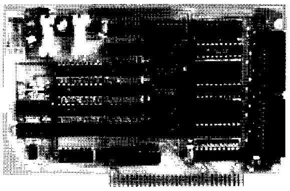
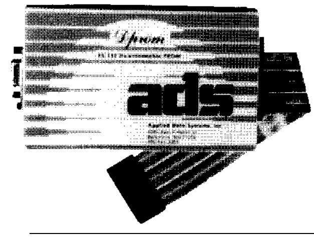
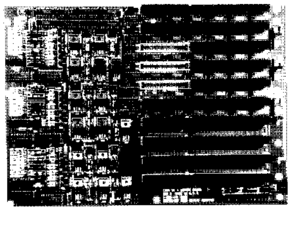


Figure 4c—*In order to properly overlay colors onto a PAL signal, you must &now whether you're on an odd or an even field, so extra circuitry must be included on the board to support PAL*


Pixel Press
32 Bit RISC Based Video Display Module
Model#
Serial#
Applied Data Systems, Inc.
409A East Preston Street, Baltimore, Maryland 21202
Made in USA
1-800-541-2003 410-576-0335 410-576-0338 FAX







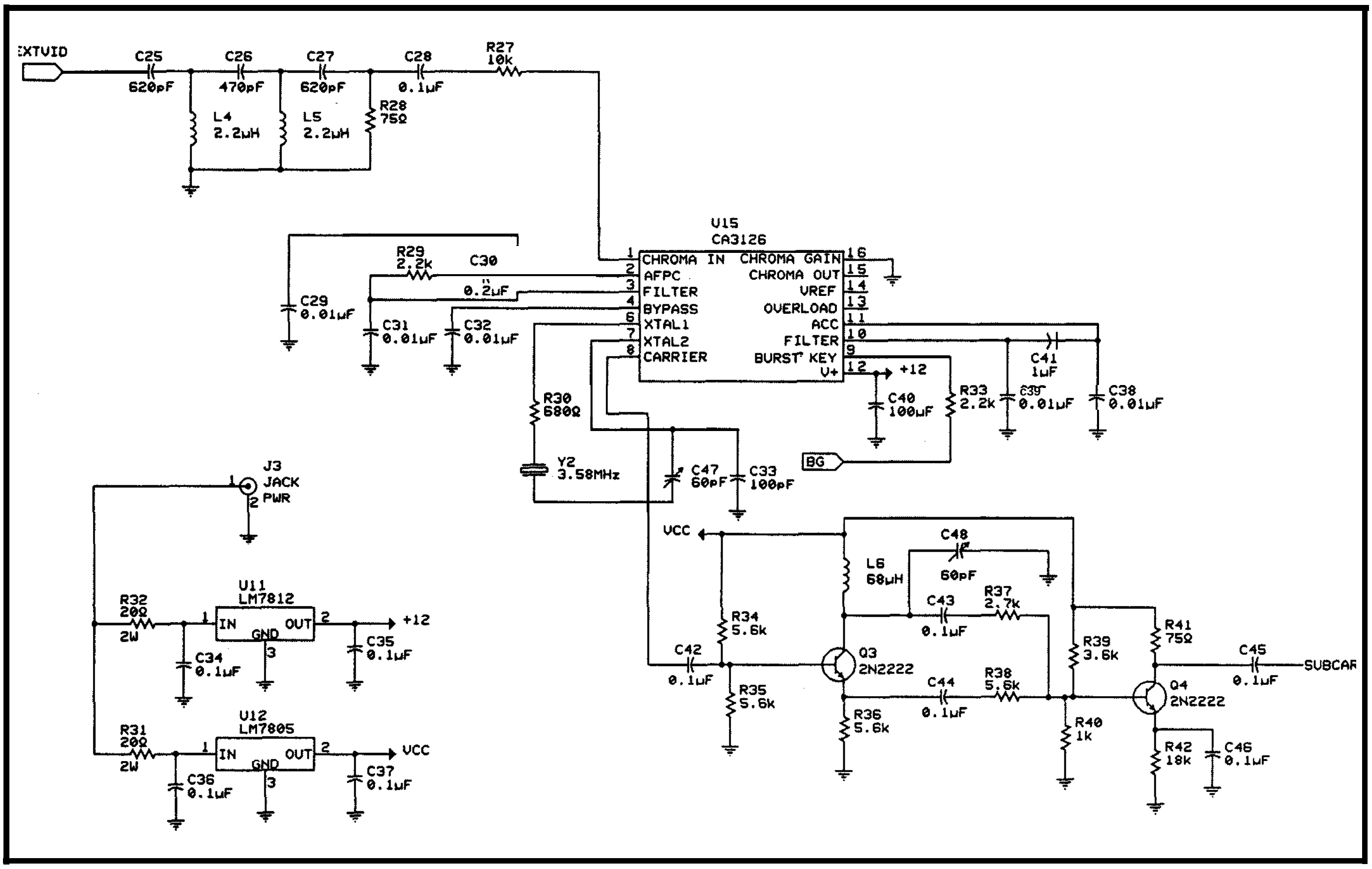


Figure 4d—*The CA31 26* TV *chroma* ***processor*** is designed ***specifically for regenerating*** chrwna ***subcarriers***

The other input, VIN3, is capacitively coupled to the original video input signal. The JRC2244 provides internal bias sources to provide DC restoration to its video inputs. The JRC2244 accepts a switching control signal from the **87C054** and switches its output between the original video input and the OSD video signal. The video switch also has an internal 75-ohm line driver in its output stage.

The JRC2244 has a moderate input impedance of about 1 **5k** ohms, allowing **10-μF** coupling capacitors to be used. The output coupling capacitor is large because this signal can be used to drive **75-ohm** loads.

RGB ENCODING

The LM1886, U13, and the LM1889, U14, encode the RGB data from the **87C054** into baseband video. The LM1886 has three **DACs,** one for each color. Each of these **DACs** has **3**-bit inputs, but because the **87C054** data is digital, the inputs to the LM1886 **DACs** are tied together yielding an output for each DAC that is either full-scale or zero. The outputs of the three **DACs** are internally summed to produce the luminance, R-Y, and **B**-Y amplitudes. The LM1889 accepts the regenerated chroma subcarrier, modulates the R-Y and B-Y signals, and produces baseband video on pin 13. Transistor **Q5** is used as a buffer amplifier with voltage dividers R49 and R50 producing proper levels for the video switch. Note that the LM1889 accepts an external subcarrier signal at the junction of R46 and C52, but this subcarrier undergoes a phase shift caused by the resistor and capacitor networks associated with pins 1 and 18 of the **LM1889.** This phase shift will need to be considered when the subcarrier is regenerated.

CHROMA SUBCARRIER REGENERATION

The circuits that reproduce a chroma **subcarrier** in the same frequency and phase as the color burst consist of a high-pass filter, a **sample**-and-hold phase-locked loop (PLL), and a phase shift network and amplifier.

The passive high-pass filter **consists** of inductors L4 and **L5,** resistor R28, and capacitors C25, C26, and C27. The filter starts passing signals at about 3.2 MHz, allowing the chroma subcarrier to pass through to the CA3126, U15.

The CA3126 is a TV Chroma Processor IC designed specifically for regenerating chroma subcarriers. This IC contains a VCO and a PLL with sample-and-hold circuits in the error correction loop. As a result, the **VCO**-generated carrier is compared with the chroma signal from the high-pass filter during the time that color **burst** is present, indicated by the burst gate pulse (which I will describe later].

The regenerated carrier output is present on pin 8 of the CA3 126. Even though this carrier is phase locked to the color burst, it is not at exactly the same phase as the color burst. The nature of a PLL is such that the output will be locked but will always have some constant fixed phase delay relative to the input. Also, recall that the input circuits of the LM1889 added

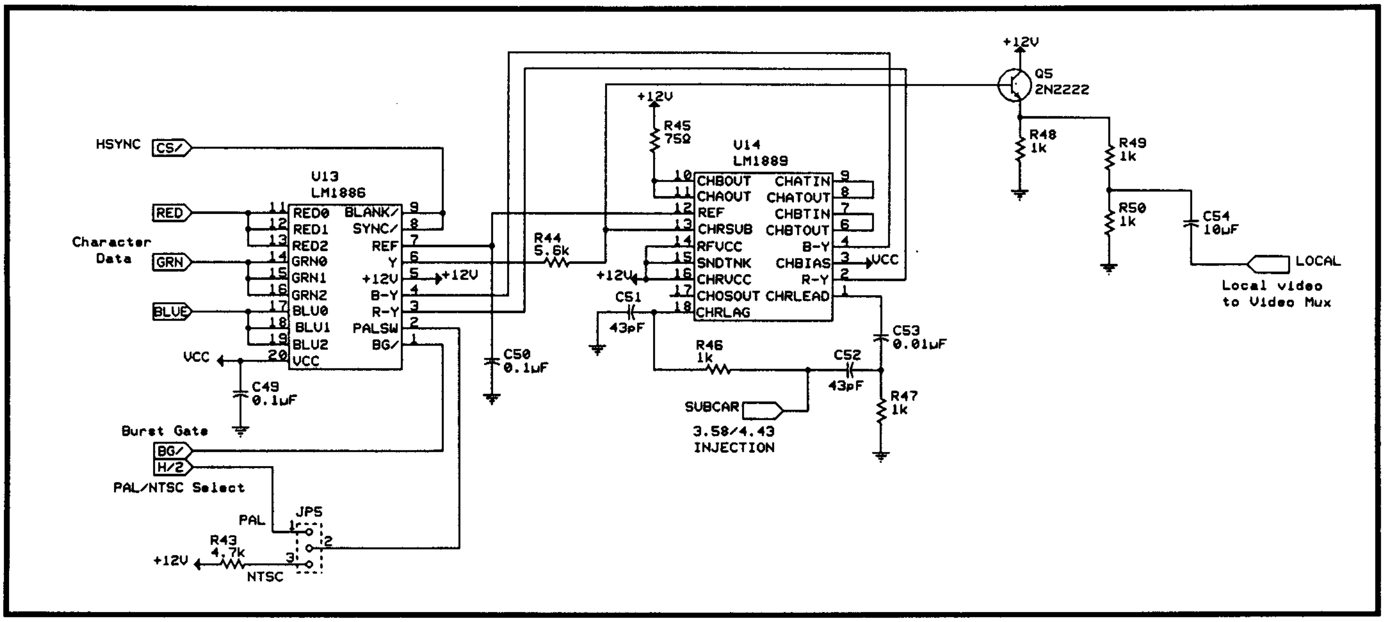


Figure 4e—*Encoding the RGB data into baseband video is accomplished with an LM1886, which contains three DACs, and an LM1889, which accepts the regenerated chroma subcarrier, modulates the R–Y and B–Y signals, and produces baseband video.*

an additional constant phase shift to the injected carrier.

The phase shift network and amplifier consisting of the Q3 and Q4 stages compensate for these fixed phase delays. This circuit provides an output whose phase is adjustable by means of variable capacitor C48, and has a tuning range of approximately 0° to 160° of phase shift. For a given input signal amplitude, the output signal amplitude is constant, regardless of the phase shift introduced. The output of this circuit is the signal injected into the LM1889 circuits.

CONNECTING TO THE HCS II

Now that you have a working terminal circuit for overlaying text onto live video, you still need to connect it to the HCS II network. In order to do this, you will need a serial interface compatible with the network, some software that handles network message formats, and software that interprets network messages and creates responses or actions (or both) to those HCS II network messages.

This particular design includes both an RS-232 and an RS-485 interface. U1, a MAX232, provides the RS-232–to–TTL conversion for both the transmitter and receiver. U16, a 75176, provides the RS-485–to–TTL conversion. JP1 connects the receiver pin of the 87C054 to either the RS-232 or RS-485 interfaces. The transmitter pin of the 87C054 connects to both the RS-232 and RS-485 interfaces. One pin of the 87C054, P3.5, controls the driver enable of the 75176, allowing for selective talking on the HCS II network. JP3 provides for termination of the network.

"But, wait a minute! The 87C054 doesn't have a UART," you say. True. There is no built-in UART on the

A = string	Set HCS II network address to string
Fx	Execute special function 0 Initialize screen 1 Display on 2 Display off 3 Display color bars 4 Wipe on 5 Wipe off
Hxy	Set Px.y high
Lxy	Clear Px.y low
Nn	Network response mode NO = normal network interface, no auto error or acknowledge responses N1 = auto error and acknowledge response
Px	Query port x 0 = Port 0 1 = Port 1 2 = Port 2 3 = Port 3
Px= nn	Write to port x where nn= two-digit hex value 0 = Port 0 1= Port 1 2 = Port 2 3 = Port 3
R X	Query register x; returns two-digit hex number 0 = OSDT (contents undefined) 1 =OSAT 2 = OSCON 3 = OSORG 4 = OSMOD 5 = Default char. attribute 6 = Default background space attr 7 = Default NEWLINE attribute
Rx = nn	Write to register x; for use from outside of a string of text; writes to these registers from within a string; should use the \Wxnn command 0 = OSDT 1 =OSAT 2 = OSCON 3 = OSORG 4 = OSMOD 5 = Default char. attribute 6 = Default background space attr 7 = Default NEWLINE attribute
S= string	String for OSD display; can include escape sequences for text formating, color, selection, etc.
\Wxnn	Write to register x; for use within a string; functionally equiv. to the Rx = nn command

Special **characterss** for use within a string of text

\E	End of Display at current position
\B	Background Space
\S	Split Background Space
\N	NEWLINE

Table 1—*The* **set** of supported commands *resembles* that *of most of the other HCS II network* **modules.**

87C054 and the part does not have a transmitter pin or receiver pin.

In this application, the serial data transmission and reception has been performed in software. The routine that handles serial transmission and reception was taken from the Signetics BBS ([800] 45 1-6644). It was originally designed for the 87C75 1 and had to be slightly modified to operate with one of the 87C054's timer/counters. The technique is often called "bit banging" and has the advantage of saving some hardware if you can afford the necessary time required of the software.

NETWORK PROTOCOL PROCESSING

As I indicated earlier, in addition to the serial interface software, you need code that handles network message formats. The code starts by waiting until either a "#" or an "!" is received, either of which indicates the start of a network message, then the entire message is stored in a buffer.

Once the carriage return has been stored, the beginning character of the message is checked to see whether the message includes a checksum. If the message does not contain a checksum, the packet is assumed to be valid and the contents of the packet are processed. If a checksum is included, then the VERIFY routine is called to perform a checksum calculation on the packet. If the checksum matches, the packet is processed; otherwise, it is ignored and I return to waiting for the next network message.

My original plans for handling network checksums included a checksum generator for sending network responses and a checksum checker for received messages. However, when I flowcharted the needs of both routines, I found that an awful lot of the logic was common to both. I went back and looked at the suggestions that Ed Nisley had provided for handling the checksums and understand now why he made those suggestions. My VERIFY routine's logic is based on Ed's previous work.

The VERIFY routine performs two functions. First, it takes the checksum digits in the packet, converts them to binary numbers, and stores them in temporary variables. Next, the checksum digits are replaced with ASCII zeros and the checksum of the string is calculated. If the checksum matches, the error flag, CHKERR, is cleared; otherwise, it is set. The checksum that was calculated is converted to ASCII and stuffed into the checksum digits position, replacing the ASCII zeros.

To prepare a string for transmission, all that is necessary is to stuff the message in the buffer with the checksum digits set to ASCII zeros and call the VERIFY routine. To check a message for correct checksum, simply call the VERIFY routine and check the CHKERR flag on return.

Once the checksum verification (if required) has been performed, you still need to process the packet to see if it belongs to this terminal, and if it does, then you need to determine what action the network controller is asking you to take.

The PROCESS routine first scans the packet, converting characters into upper case until the end of the packet has been reached. Next, the first character is examined to determine if the packet has checksums or not and a pointer is set to the NODEID position of the packet. The NODEID in the packet is compared with the N0DEID variable. If there is no match, the packet is ignored and you wait for the next network message. If it does belong do this terminal, you can process the body of the network message.

NETWORK COMMANDS AND SYNTAX

The real essence of a network message is to carry a command from the network controller to the terminal or carry a response from the terminal back to the network controller. Table 1 shows the syntax of the commands available for operating the TV-Link terminal. These commands allow the HCS II Supervisory Controller to manipulate ports on the 87C054, format text for display, implement special built-in display functions such as color bars, and to read and write OSD registers directly, giving full control of the OSD to the HCS II.

CONCLUSIONS

Developing this application was interesting and enjoyable. It also presented some challenges.

The 87C054 proved well suited to this application in large part because of the 80C51 core and that the OSD is independent of the CPU. Once characters have been written to the OSD, you can forget the OSD until you want to change the display, and the CPU is free to pursue other tasks.

The on-screen display and the microcontroller operations are primarily digital functions. The question of how to combine this technology with an analog video signal can be perplexing to most system designers whose professional experiences have been mostly digital circuits. One of the most perplexing issues during this project was how to recreate the **chroma subcarrier.** I knew that every color **TV** set had to perform this function, but finding out solutions took some searching before I discovered the CA3 126. I'm hopeful you can profit from my experiences on this project. ❑

My *thanks to Herb Kniess and George Ellis of Signetics for their help. Thanks also to Greg Goodhue from Signetics, who wrote the software-based UART code for the 87C751 that I modified for this project.*

Bill Houghton is an Applications Engineer at Signetics specializing in 80C51-based microcontrollers.

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

Requests for literature on Signetics/Philips microcontrollers including the "80C5 1 -Based 8-Bit Microcontroller Data Handbook" may be directed to Sharon Baker at (408) 991-3518.

Contact Bill Houghton at (408) 991-3560 with technical questions specific to the 87C054 and for information on the availability of a PC board and components for this project.

I R S

410 Very Useful
411 Moderately Useful
412 Not Useful

The Virtues of the Hue, Lightness, Saturation Color Model

If you thought RGB (red, green, blue) was the only (or best) way to represent color, think again. The Hue, Lightness, Saturation (HLS) model can actually be better than RGB at representing colors in some applications.

SPECIAL SECTION

James R. Furlong

have you ever **wondered why** graphics that use the default EGA/VGA color palette **look** as lousy as they do? Did you think your boss was kidding when he told you to mix equal levels of red and green to create yellow on your computer? My Crayolas never did that. Have you ever wanted to portray the shock heating of a piece of orbital debris impacting a satellite bumper shield at 6.5 kilometers per **zas** continuous tones of red, but gave up after trial-and-error guessing of RGB combinations made you surmise you were color blind? If you answered yes to any of the above, let me assure you that a) you are not alone and b) a well-established but little-publicized color model named HLS (for Hue, Lightness, and Saturation) is the cure for your palette woes.

Designed by people who thought about what color means to people before how color is described for a machine, the HLS model takes the guesswork out of making colors like shocking pink or pastel green. Simple analytic functions and a handful of lines of computer code create a palette of continuous tones of red, or periwinkle, or they can bridge dark blue with bright orange because the model's coordinates are dimensionless.

The following description is of the transformation equations between the **HLS** and RGB color models and the BIOS interrupts necessary to load IBM VGA hardware with RGB values, so you can think in terms of HLS but speak to your computer in RGB. I also introduce an extension to the I-ILS model that allows more color possibilities and unlocks the 262,144 colors of which the VGA is capable.

MODEL COORDINATES

Figure 1 illustrates the conceptual framework for the HLS model [1]. The figure is three-dimensional because the model has three coordinates. They are defined as follows:

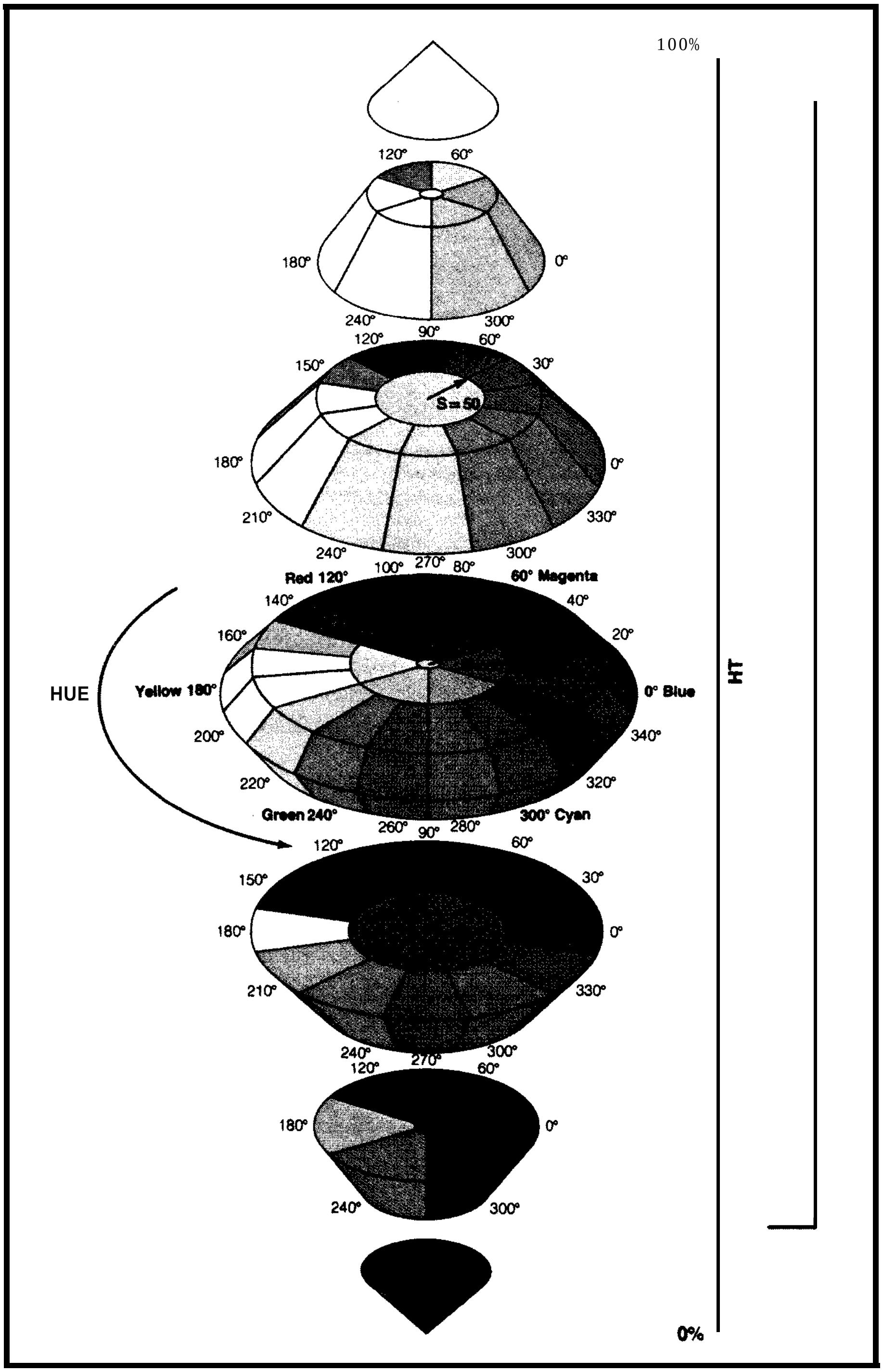
Hue-Hue is what many of us would commonly refer to as color. Red, green, and purple all are hues, but in the context of the HLS system they are colors absent of a definition for lightness (dark red or bright red), and saturation (pure red or pink).

Hue is perceived as the azimuth coordinate on the model. This designation isn't arbitrary, but rather it is a consequence of a special additive nature of the RGB system itself. In Figure 1, notice that if blue is arbitrarily taken as the starting point, you can smoothly scroll through the **cyans,** greens, yellows, reds, magentas, and finally wind up back at the beginning, namely blue, without any loss of continuity. You will have the sense that you passed through all the basic colors in the visible spectrum.

Saturation-Saturation defines the pureness of a hue for a given level of lightness. Conceptually, saturation is the radial coordinate in the model. The larger the radial coordinate, the purer the color. Colors of low saturation tend to be soft, or pastel, and fully saturated colors can be harsh, depending upon their lightness.

A lack of saturation defines the grayness of a hue. Note that grayness covers all levels of gray, from total black to pure white. Any hue at zero saturation will be pure gray. Whether the gray is black, white, or somewhere in between will depend on the lightness. A good example of an unsaturated color is pink, which is somewhere between pure red and white.

Lightness-Lightness is how bright or dark a color is, as the name implies. It is the vertical axis of the model. Due to the model's double cone structure, the maximum saturation level depends on lightness. This level varies from 1 at the equator to 0 at the poles. A lightness of 0 or 1 will have a maximum saturation of 0 because



Figun I-The HLS mode/ can be conceptualized in three dimensions as a pair of cones. Hue proceeds anwnd the cones, Lightness runs in the vertical direction, and Saturation extends radial& from the center and depends to some degree on the Lightness. (© Tektronix Inc.)

those values correspond to black or white, while a hue can be fully saturated at the equator.

The intuitiveness and elegance of the HLS model become apparent when it is compared to RGB. Take pastel green, for example. Under RGB you would first set the lightness of green with *GREEN* intensity, but to soften the green you would have to add equal levels of *BLUE* and *RED.* That doesn't really make sense, does it? Under HLS you would first set *HUE* to green, then make the green as bright or dark as you want with *LIGHTNESS,* and as pastel as you want with *SATURATION.*

Furthermore, pastel green happens to be one of the easier colors to make under RGB because equal levels of *BLUE* and *RED are* an exact complement of *GREEN. If you* shift the color from pastel green to say pastel cyan (cyan is equidistant from *BLUE* and *GREEN),* then the ratio between *BLUE* and *RED* would no longer be unity, and you would need an ambiguous combination of *BLUE* and *RED* to soften the cyan.

TRANSFORMATION EQUATIONS

The process of creating the HLS transformation equations begins by recognizing the relationship between RGB and the hue coordinate. Figure 2 lays out *HUE* in one dimension. Although *HUE* is the azimuth coordinate and should be referred to by angle, I have normalized its value to 1, maintaining consistency with the other coordinates. You can make the starting point any value you want because the hue coordinate is circular. In my system, *HUE* = *0* is defined as blue.

With a little imagination, you can see that if trapezoidal-like functions for *BLUE, GREEN,* and *RED* are used (which only differ from each other by their starting hue point) the entire hue coordinate can be traversed to mimic the color transitions shown in Figure 1. For example, starting from *HUE* = *0* and tracing through the functions in Figure 2, *BLUE* is the only function with a positive value. Advancing to *HUE* = 1/6, *BLUE* remains constant while *GREEN* begins to increase, eventually equaling *BLUE. The* equal levels of *BLUE* and *GREEN* create the familiar cyan hue.

If you continue to trace through the functions, for 1/6 < *HUE* < 1/3, *GREEN* remains constant while *BLUE* is brought to 0, resulting in the green hue at *HUE* = 1/3. A subsequent tracing through the functions will show that the growth and attenuation experienced by *GREEN* repeats for the other two colors, first for *RED* at *HUE* = 1/3 and then for *BLUE* at *HUE* = 2/3. At *HUE* = 1, you have come full circle and are back at blue.

Defining the trapezoidal-like functions for *RED, GREEN,* and *BLUE* requires nothing more than superposing linear functions with different amplitude offsets. For example, the function ***BLUE*** over the interval $0<$ HUE $<1/6$ is defined by

$$BLUE = f_1 + f_2$$

where,

$f_1 = -\frac{1}{Dx}x+2$ and $f_2 = \frac{1}{Dx}x-1$

where x represents ***HUE,*** and AX is a constant equaling 1/6. These two functions will maintain ***BLUE* at a** constant value of 1 indefinitely. ***BLUE*** needs to be attenuated for x >1/6, which is done by adding another linear function, such as

$$f = -\frac{x}{\Delta x}+1$$

for x >1/6. However, limiting f_2 to negative values with a **MIN function and letting *BLUE*** decrease according to f_1 is just as easy. Similarly, a MAX function can be used to prevent ***BLUE*** from becoming negative for x >1/3. The analytic form for the function ***BLUE*** over the entire range of **the hue coordinate as well as the functions for *GREEN*** and ***RED are* as** follows:

$$Blue = MAX\left(\frac{x_2-x}{\Delta x},0\right)+MIN\left(\frac{x-x_1}{\Delta x},0\right)+MAX\left(\frac{x-x_4}{\Delta x},0\right)+MIN\left(\frac{x_5-x}{\Delta x},0\right)$$

$$Green = \frac{x}{\Delta x}+MIN\left(\frac{x_1-x}{\Delta x},0\right)+MIN\left(\frac{x_3-x}{\Delta x},0\right)+MAX\left(\frac{x-x_4}{\Delta x},0\right)$$

$$Red = MAX\left(\frac{x-x_2}{\Delta x},0\right)+MIN\left(\frac{x_3-x}{\Delta x},0\right)+MIN\left(\frac{x_5-x}{\Delta x},0\right)$$

Listing 1 shows the C code for the functions ***BLUE, GREEN,*** and ***RED,*** **named** `mag_blue`, `mag_green`, and ma `g_red`, respectively. The values **returned** are always between 0 and 1.

Listing 2 contains the C code for the set pa 1, which transforms HLS values to RGB and then calls the video BIOS to load VGA registers with the RGB values. set pa 1 **receives as its** argument values for hue, lightness, saturation, and the register whose color you are going to define.

For VGA 640 x 480 resolution, only 16 colors may be displayed **simultaneously,** which means only register values O-15 are legal. set pa 1 is declared i n t, so it may return a value about the success of the function call. I did not show you the coding that checks if the arguments passed to `setpal` are **within range and if the current screen mode is 16-color VGA.**

The first operation set pa 1 performs is determining the complement of hue, referred to as ***euh*** in the coding. Knowing ***euh*** is essential for establishing the saturation.

Refer to Figure 3, which shows a horizontal slice through Figure 1 at the equator. This figure shows a vector that indicates a fully saturated hue **drawn, in this case, at the "orange" azimuth. Adding a vector *180"*** away from the hue vector reduces the radial magnitude of the hue because the saturation is the magnitude of the

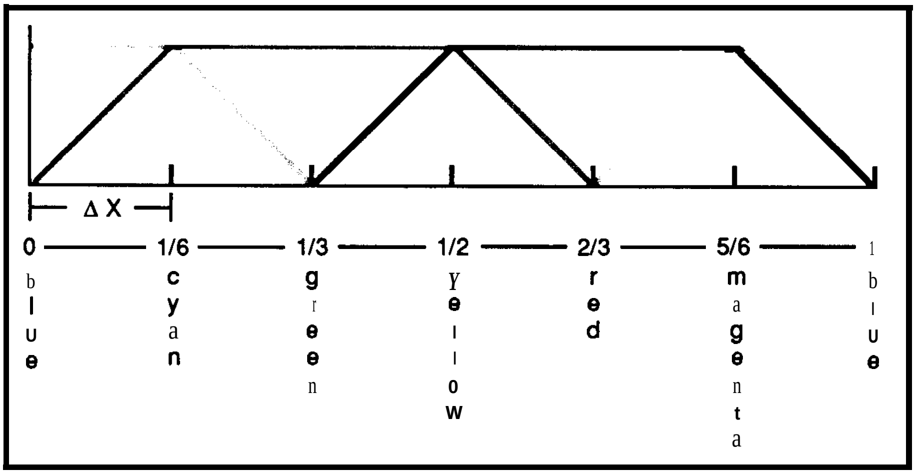


Figure 2—*By laying out hue in one dimension, it's possible to begin* bseehe *relationship between hue and RGB. Each of the RGB colors forms a trapazoid that partially overlaps he other colors.*

Listing I--*When de&mining the proper red, green, and blue components, a normalization process must be done to ensure he final values are between 0 and1.*

```
/*================================================
    return normalized blue component
================================================*/
float mag_blue(float x)
{
float result;

    result = (maxl(x2 - x. 0.) + minl(x - xl, 0.) \
           + maxl(x - x4. 0.) + minl(x5 - x. 0.)) \
           / delx;
   return(result);
}

/*================================================
    return normalized green component
================================================*/
float mag_green(float x)
{
float result:

    result = (x + minl(xl x. 0.) + minl(x3 - x, 0.) \
          + maxl(x x4. 0.)) / delx;
   return(result);
}

/*================================================
    return normalized red component
================================================*/
float mag_red(float x)
{
float result:
```

(continued)

radial coordinate. euh is just such a vector. It is determined simply by adding 1/2 to the hue if the hue is less than or equal to 1/2, or subtracting 1/2 from the hue if the hue is greater than 1/2.

Next, set pa 1 determines the maximum saturation level, sat 0. Due to the double cone structure of the model, sat 0 is a function of lightness. At a lightness of 0 or 1, sat0 must be 0, and at a lightness of 1/2, it must be 1. In a fashion similar to the making of the functions for *RED, GREEN,* and BLUE, you make use of two linear functions and a MIN function to create the desired function

```
sat0 =2*lite
      + minl((-4*lite)+2.0)
```

The hue value is systematically passed to mag_blue, mag_green, and mag_red to calculate blue, green, and red values for the hue, and the euh value is also passed to calculate blul, grenl, and redl values for the hue complement, *euh.* Functions max 1

Listing I-continued

```
   result = (maxl(x - x2. 0.) + minl(x3 - x. 0.) \
         + minl(x5 - x. 0.)) / delx:
   return(result);
}

/*===================================================
   return maximum of two floats
===================================================*/
float maxl(float argl. float arg2)
{
float result;

     result = argl:
     if (arg2 > argl)
       result = arg2:
     return(result);
}

/*===================================================
   return minimum of two floats
===================================================*/
float minl(float argl, float arg2)
{
float result;

     result = argl;
     if (arg2 < argl)
       result = arg2:
     return(result);
}
```

Listing 2—The se tpa 1 routine transforms HLS values to RGB and calls the video BIOS to bad VGA registers with the RGB values.

```
#include <dos.h>
#define VIDEO_IO 0x10

float. delx = 1./6., \
        xl = 1./6., \
        x2 = 1.13.. \
        x3 = .5,    \
        x4 = 2.13.. \
        x5 = 5.16.;
union REGS regs:

int setpal(float hue, float. sat. float lite, int reg)
/*================================================
 convert. HSL coordinates to RGB
 call s set_reg to load VGA hardware with palette data
================================================*/
{
void set_reg(int reg. int red, int green , int blue):
float mag_blue(float), mag_green(float), mag_red(float);
float maxl(float argl, float. arg2);
float minl(float argl, float. arg2);

float, ITENO = 63.;             /* Max VGA intensity */
float red, green, blue;
float redl. grenl. blul, euh. sat0;

   if (hue <= .5)               /* hue compliment (euh) */
     euh = hue + .5;
   else
     euh = hue .5;

   sat.0 = 2. * lite + minl((-4. * lite) + 2.. 0.);
   sat.0 = maxl(sat0, 0.);      /* max sat = f(lite) */
   sat0 = minl(sat0, 1.);

   blue = mag_blue(hue);
   blue = maxl(blue, 0.);
   blue = minl(blue, 1.);

   blul = mag_blue(euh);
   blul = maxl(blul, 0.);
   blul = minl(blul, 1.);

   blue = ITENO * lite * (blue + (1.   sat*sat0) * blul);

   green = mag_green(hue);
   green = maxl(green, 0.);
   green = minl(green, 1.);

   grenl = mag_green(euh);
   grenl = maxl(grenl, 0.);
   grenl = minl(grenl, 1.);

   green = ITENO * lite * (green + (1  sat*sat0) • grenl);

   red = mag_red(hue);
   red = maxl(red, 0.);
   red = minl(red, 1.);

   redl = mag_red(euh);
   redl = maxl(redl, 0.);
   redl = minl(redl, 1.);

   red = ITENO * lite * (red + (1. sat*sat0) * redl);

   set_reg(reg, (int)red, (int)green, (int)blue);
   }
}
```

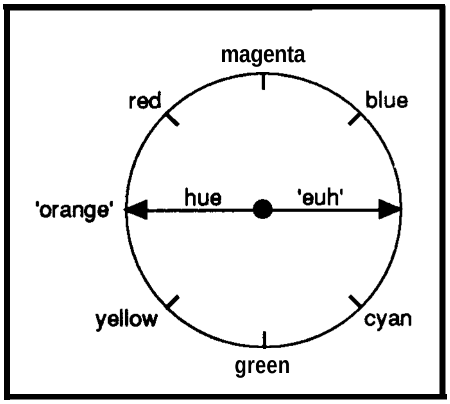


Figure 3—*Taking a horizontal slice through the center of Figure 1 shows the "euh," or tie complement of the hue, which is necessary b determine saturation.*

and min1 are used to keep returned values within range.

The RGB components of euh are modifiedby (1.-sat*sat0) to establish the correct radial magnitude of the RGB components of hue. If (1. - sat*sat0) is 1, hue and *euh* will cancel each other, the hue radius will be 0, and the resulting color will be a gray tone.

Finally, each component is multiplied by an intensity value, mapping the RGB values into video hardware space. The VGA is an 18-bit color device. It uses 6-bit values for *RED, GREEN,* and *BLUE.* Hence, the range for any component is O-63. The float variable ITENO is used to hold the value 63. The net intensity is simply ITEN0 times the lightness, or in setpal, the variable lite.

Defining the color for a VGA register is most easily done by making the video interrupt BIOS INT 10 call [2]. Listing 3 shows the C code for the subroutine set_reg, which uses the Microsoft C int86 function to make such a call. All values used by the BIOS calls must be integers, so each color is cast with the int type when set_reg is called and then recast as unsigned chars before being loaded into the regs union structure.

Inside set_reg, two BIOS calls are made. The first call sets the palette register with an attribute. In this case, I have made the attribute equal to the register value. The second call loads the register with the RGB values. To use the color, a program need only make the appropriate graphics library call to use the color defined by the attribute. In Microsoft C, the call would be _setcolor(attribute).

EXTENDED HLS

If you haven't already noticed, the HLS model I just described lacks the ability to define really bright or really dark colors. Before the colors can get bright or dark, they are unsaturated to the point of being either pure white or pure black. This step is because of the double cone structure of the model.

Listing 3—*The function set_reg is used to define a color for a VGA register.*

```
void set_reg(int reg. int red, int green. int blue)
/*===================================================
 BIOS call to set color in VGA 640x480/16 color mode
 1) selects register and attribute
 2) sets color
===================================================*/
{
        regs.h.ah = 0x10:           /* set palette register */
        regs.h.al = 0x00:
        regs.h.bl = (unsigned char) reg:
        regs.h.bh = (unsigned char) reg;

        int86(VIDEO_IO, &regs, &regs);

        regs.h.ah = 0x10:           /* set color register */
        regs.h.al = 0x10;
        regs.h.bl = (unsigned char) reg;
        regs.h.ch = (unsigned char) green:
        regs.h.cl = (unsigned char) blue;
        regs.h.dh = (unsigned char) red;

        int86(VIDEO_IO, &regs, &regs);
}
```

In the extended HLS model, each cone is converted into a cylinder. Now when LIGHTNESS goes from 0 to 1, the colors don't get washed out. Black and white occur only when saturation is 0. In effect, an increase in the number of colors possible given by the ratio of the volume of a cylinder to that of a cone has been realized. This ratio of 3 means an additional 174,762 colors for the VGA without any loss of concept. In fact, the only change to the code in Listing 2 needed is to define Sat0 as 1, or better yet, leave it out altogether.

SOME EXAMPLES

Having color defined in an easy-to-understand and dimensionless coordinate system means that not only can a single color be defined easily, usually on the first try, but that a complicated color spectrum from shocking pink to pastel green can be defined with simple analytic functions. Listing 4 shows several examples that illustrate the compactness of code needed to define an array of colors and the versatility of the HLS model.

Example 1 traverses all the colors of the rainbow in pastel shades. To achieve the pastel effect, the intensity and saturation variables are set to 1/2. The for loop divides the hue into equal intervals between 0 and 1.

Example 2 creates a gray scale. ***SATURATION*** is set to 0 outside the for loop. ***LIGHTNESS*** is graded between 0 (black) and 1 (bright white) within the for loop. Any value of hue can be used since ***SATURATION*** is 0.

Example 3 strives to define a palette that will elucidate the physics of a scientific calculation. In this case, the physics is the plastic strain in a simulation of an aluminum pellet striking an aluminum satellite bumper shield at 6.5 kilometers per second. ZeuS hydrodynamic computer code [3] performed the simulation.

Whipple [4] first proposed using satellite bumper shields for protecting spacecraft. Orbital velocities of micrometeorites or relics of yesterday's space missions vary from a few to tens of kilometers per second. At those velocities, even gram-sized particles can pack the wallop of a 30-

Listing 4-Several *examples serve to show the compactness of wde needed to define an array of colors and the versatility of the HLS model.*

```
#include <graph.h>

main0
{
#define MAXREG 15

int setpal(float hue. float sat, float lite, int reg);
void draw_graphicO:
void key_wait();
int reg. error = 0;
float hue. sat. lite:

   _setvideomode(_VRES16COLOR); /* VGA 16 color */

/*===============================================
      Example 1: blue to blue with pastels
  ===============================================*/

    lite = .5;
    sat = .5;
    for (reg = 1: reg <= MAXREG: reg++){
      hue  =  (float)(reg 1) / (float)(MAXREG - 1);
      if (error = setpal(hue, sat. lite. reg))
        goto error-handler;
      }
    draw_graphicO;
    key_wait();
    _clearscreen(_GCLEARSCREEN);

/*===============================================
      Example 2: grayscale
  ===============================================*/

    hue = 0.;
    sat = 0.:
    for (reg = 1: reg <= MAXREG: reg++){
      lite  =  (float)(reg - 1) / (float)(MAXREG 1);
      if (error = setpal(hue, sat, lite, reg))
        goto error-handler:
      }
    draw_graphic();
    key_wait();
    _clearscreen(_GCLEARSCREEN);

/*===============================================
  Example 3: blue to orange w/ increas sat and lite
  ===============================================*/

    hue = -.0428;
    sat = .6 -.0285;
    lite = .3- .0214;
    for (reg = 1: reg <= MAXREG; reg++){
      hue += .0428;
      sat += .0285;
      lite += .0214;
      if (error = setpal(hue, sat, lite, reg))
        goto error-handler;
      }
    draw_graphic();
    key_wait();
    _clearscreen(_GCLEARSCREEN);

error-handler:
   _setvideomode(DEFAULT);
   if (error) printf("setpal error # %d", error):
}
```

mm canon. An inexpensive, thin-walled sacrificial plate, or bumper, is used to intercept the particle. The shock wave stress caused by the collision is high enough to vaporize the particle so the debris carried downstream cannot harm the vehicle.

The palette definition begins with *HUE* set to blue (a subjectively cool color) and ends at orange (a subjectively hot color). *SATURATION* and *LIGHTNESS vary* linearly at different rates as *HUE goes* from blue to orange. This feature emphasizes the change in intensity. When you apply HLS to the problem, a tapestry of science and art allows a magnificent visualization of the event.

Other analytic functions produce different effects. Part of the fun with the HLS model is just experimenting with different functions. Creative functions lead eventually to a library of function definitions that might aptly be called color effects. With a little practice, you will find that going from periwinkle to thistle is both easy and fun. ❑

fames R. Furlong is a research scientist with a defense-related organization in Arlington, VA. He develops constitutive models for materials undergoing shock used in large-scale hydrodynamic computer codes. He is also head of software development for Eclectic Systems in Springfield, VA.

REFERENCES

1. *Tektronix 4107/4109 Programmer's Manual,* Appendix E, 1982.
2. *IBM Personal System/2 Display Adapter Technical Reference,* International Business Machines Corporation, 1987.
3. *ZeuS: Technical Description and User's Manual,* Computational Mechanics Consultants Inc., Baltimore, MD, 1990.
4. F. Whipple, *Meteoric Phenomena and Meteorite: The Physics and Medicine of the Upper Atmosphere,* University of New Mexico Press, Albuquerque, NM, 1952.

SOURCE

Readers interested in experimenting with the HLS model may contact Eclectic Systems for a free copy of the utility "MYPAL." MYPAL is an interactive program used to define palettes in HLS space. HLS coordinates are saved in ASCII files. Programs that read these files can use the source code given in this article to set their own palettes. MYPAL is available in DOS and Windows formats. A nominal fee to cover shipping and handling will be requested.

Eclectic Systems
8 106 St. David Court
Springfield, VA 22 153

I R S

413 Very Useful
414 Moderately Useful
415 Not Useful

Driving Multiple VGA Monitors

SPECIAL SECTION

Michael Swartzendruber

Monitor extenders costing hundreds of dollars are available that allow you to drive multiple monitors with a single display adapter. Michael uses just a touch of ingenuity to bring down the price considerably.

At times, driving more than one monitor from a single **PC video port is practical** or necessary. Software demonstrations or other kinds of group demonstrations are examples of one type of occasion. Side-by-side comparisons of different brands of monitors are another. Also, certain artistic displays work best when more than one monitor displays an image.

Multiple-monitor drivers that drive from two to sixteen monitors simultaneously are currently available. These devices start at about $300 and go up in price depending on the number of monitors the device can drive. However, the simple circuit I describe here is an alternative that addresses these problems at a much lower cost. I built the "core" of this circuit with three low-cost transistor chip arrays and a handful of resistors. Moreover, you can duplicate the core ad infinitum to drive as many monitors as demanded by the task at hand.

THE CIRCUIT

Figure 1 shows my **multiple**-monitor driver. This simple device implements any number of parallel emitter-follower amplifiers, which serve three essential purposes. The first is to provide a properly terminated load to the video card in the computer. Second, these amplifiers serve as isolation amplifiers between the PC video card and the multiple monitors being driven from the video source. The third purpose of these amplifiers is to provide drive current to the inputs of the monitors connected to the circuit.

Notice that the base of the driver circuits' transistors connect to ground through a **75-ohm** resistor. As I mentioned before, this resistance is provides the proper amount of load resistance to the PC video card. The base of each monitor's driver-amplifier connects to this point. The multiple bases connected to this resistor do not alter the **75-ohm** termination resistance appreciably, allowing multiple emitter-followers to be driven from the video card. In turn, this feature is what allows the PC video card to drive multiple monitors.

The amplifiers serve as isolation amplifiers between the video card and the monitors by the isolatable **charac-**

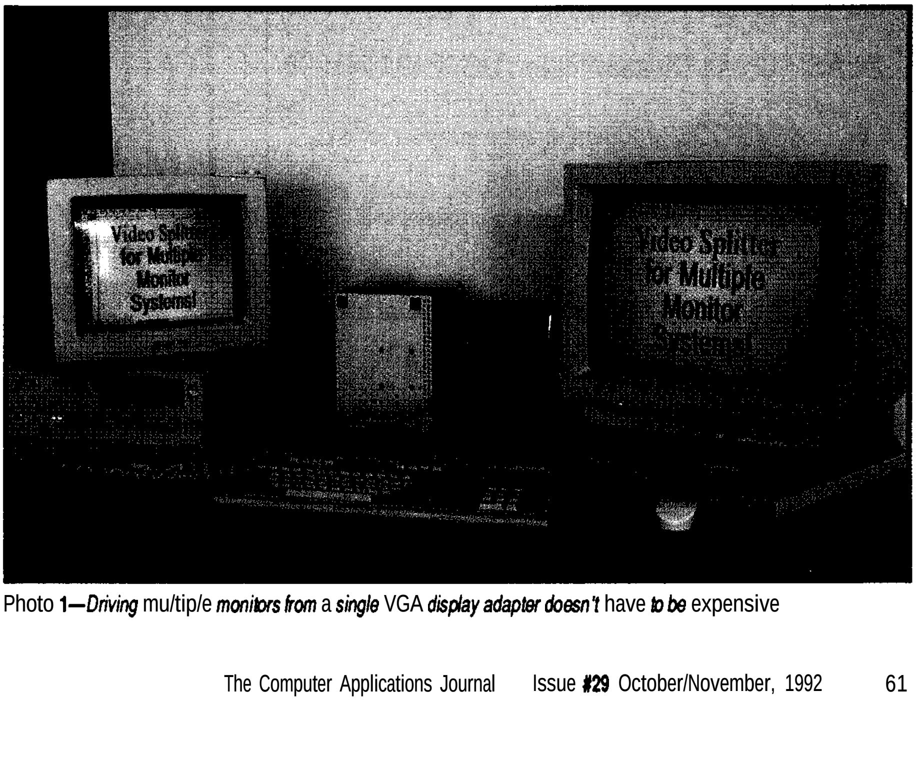

Video Splitter for Multiple Monitor Systems!
Video Splitter for Multiple Monitor Systems!

Photo 1—*Driving* mu/tip/e *monitors from* a *single* VGA *display adapter doesn't* have *to be* expensive

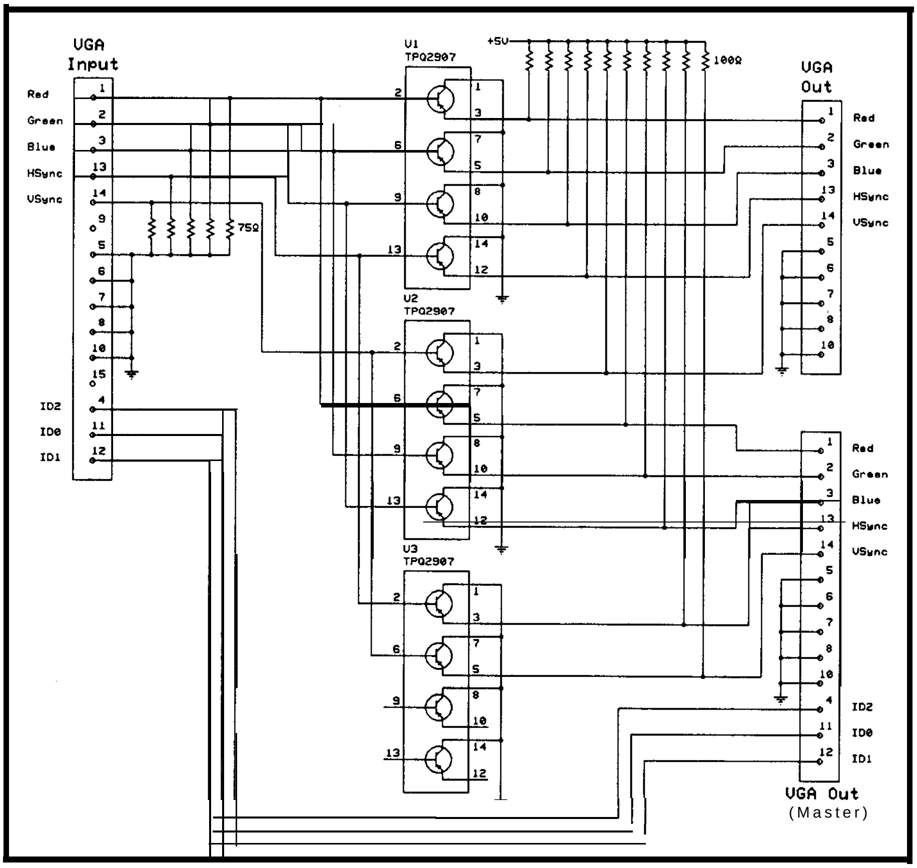
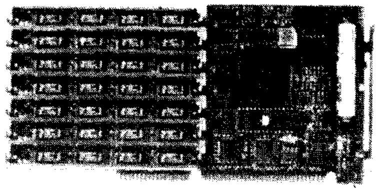


Figure I--The *TPQ2907 quad transistor* peck is the key *to the* multiple monitor *driver* circuit.

teristics of an emitter-follower amplifier. As mentioned above, the PC video card only "sees" 75 ohms of load resistance no matter how many bases connect in parallel at this point. The emitter of each transistor follows the voltage level of its base, which makes the emitter circuit the signal source to the monitor connected to it. At this point, the signals are directly derived from the video card.

The emitter-follower circuit provides load current that drives the monitor inputs. This drive current is provided by the collector emitter circuit and does not require any significant level of load current from the base circuit. Therefore, you can use this circuit to connect multiple monitors without affecting the load current the video card must supply.

The combination of all of these simple factors allows this circuit to operate. If you need more monitors, simply connect more transistors' base leads **to** the **75-ohm** resistors. Then just build the same "core" over again for each monitor.



I designed this particular version of the circuit to work with the standard VGA D-type 15-pin connector. You can easily change the circuit to have it work with any other color video monitor standard as long as the Red, Green, Blue, H-Sync, and V-Sync signals can be identified. These signals would be input into the core circuit in exactly the same manner. The outputs of the circuit could then be connected to the appropriate pins of the connector for that color monitor standard.

Note the signals are identified as *ID0, ID1,* and *ID2.* The VGA monitor provides these signals to the video board for input. They identify a monitor's type to the VGA board, and the video board uses this information during its power-up automatic mode-setting operation.

In order for this operation to finish correctly and without conflicts, only one monitor should provide these signals. Therefore, if you use multiple cores, only one port should pass these signals to the VGA board. The one port that has these signals connected through it will be named the *master monitor port* because the monitor connected to it will be used to set the power-up mode of the video card.

Connect the lowest-performing monitor to the master monitor port to ensure all monitors connected to the other ports are capable of operating. If you use the highest-performing monitor as the master, some monitors may be incapable of performing at the master's mode. Another way to operate this circuit with monitors that support different resolution levels or capabilities is to leave *ID0, ID1,* and *ID2* on all ports disconnected and issue manual video mode-setting commands to the video board.

CIRCUIT USE

This circuit is very easily installed and used. Connect the PC video port to the input port of the circuit assembly, then the monitors to its video output ports. The only restriction is connecting the lowest-performing monitor to the master monitor port, although even this recommendation is not absolute and depends on your application or your ability to issue video mode-setting commands.

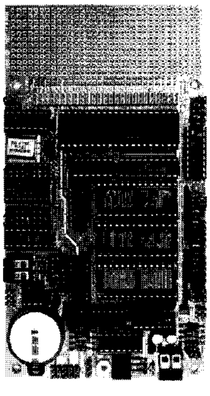
CONCLUSION

This easily built and usable circuit is a great help on those occasions where driving multiple monitors is necessary. A very low cost makes it the most sensible option when compared with similar devices, most of which are offered for sale at about 500 times the cost to build this device. This card has many more applications than the small number I've suggested. Just having a device like this one on hand reveals a whole new world of computing possibilities. ❑

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

IRS

416 Very Useful
417 Moderately Useful
418 Not Useful



DEPARTMENTS

FIRMWARE FURNACE

Ed Nisley

Extending Your Control: The HCS II MCIR-LINK

Since the introduction of the HCS II home control system, one of the most requested new features has been the ability to send IR codes to consumer electronic equipment. The new MCIR-Link makes those requests reality.

“If you wish hard enough, it will come true” is an aphorism often attributed to the Walt Disney School of Engineering. But at least in the wonderful, malleable world of firmware, wishes sometimes do come true!

Back in issue #25 of the Computer *Applications Journal,* Steve, Ken, and I introduced the Home Control System II (HCS II). The system is based on a Supervisory Controller, which handles all the grisly scheduling and oversees the whole system, and networkable “Link” modules, which expand the system I/O by adding digital I/O ports, LCD display, X-10 power-line control, and more. I’ve been steadily covering all those network modules in the last few issues.

Back in issue #26’s “Firmware Furnace,” entitled “Infrared Home Control Gateway,” Steve and I presented the IR-Link module as a command and control gateway into HCS II. It received and transmitted infrared signals in the 9-bit Motorola MC145030 format and included enough features to implement IR badges, people tracking, and so forth and so on. Refer to that issue for a complete IR-Link schematic and discussion of the hardware and firmware.

HD addr	dump MCIR data from RAM to HEX w/base addr
HR	receive MCIR data from HEX to RAM
SM n	send MCIR signal n (1–999 RAM, 1001–1999 EPROM)
Tn	train new MCIR code, store in RAM as n (l-999)
TD n	set duty cycle correction factor 0-255
TX	clear MCIR data from RAM

Figure l-The *MC/R-Link firmware adds these commands to the IR-Link's repertoire. Unlike the other Link modules, the new H commands transfer Intel hex data directly between the host computer and the MC/R-Link's RAM.*

However, sending commands to the HCS control program with an IR remote wasn't enough for you. What you really wanted was to send IR commands from the HCS to your TV/ VCR/CD/amp widgets. The IR-Link seemed to have all the essential hardware, so it ought to be a simple matter of software. Right?

The MCIR-Link module is our response to your requests. In addition to all the IR-Link functions, it can record and play back the IR signals sent by most hand-held remote control units. The initial training must be done manually (you need a finger to push the remote's buttons!), but transmission is entirely automatic and can be handled by the Supervisory Controller's XPRESS program. The MCIR-Link board's 32K-byte RAM has enough room for several hundred remote control signals, so you can control nearly anything.

Figure 1 presents the new MCIR-Link commands; the complete IR-Link set is given in "Infrared Home Control Gateway" in issue #26. You simply teach the MCIR-Link new IR signals using the T command and send them with the SM command. The H commands allow you to dump and restore the IR data, so you can save your work on disk and program it into EPROM.

As with all the other "-Links," you will need a PC with a terminal emulator program to drive the MCIR-Link in the interactive mode. Also like previous modules, this board and firmware can be used in applications that have nothing to do with HCS II. The Links use a straightforward ASCII command set over standard RS-232 or RS-485 wiring; therefore, you can drive this board manually with a terminal emulator, a simple PC program, or your own home control system.

That's about as complex as the user interface gets, so I won't say much more about it. Instead, I'll explore the MCIR-Link firmware and describe some of the gotchas that make IR control such a challenge. Along the way, you'll discover techniques for your own projects and, perhaps, find new respect for your TV set's lowly remote control receiver.

REMOTES UNCONTROLLED

The first step in any project should be figuring out what you need to do. What you need to do here is add to your home control system the capability to recognize the IR signals of the devices you wish to incorporate, which you do by training the MCIR-Link with the remotes. To begin, a quick review of the basics of IR controls is in order, although you should be reasonably familiar with them from articles about the original Master Controller, the IR-Link, and similar projects.

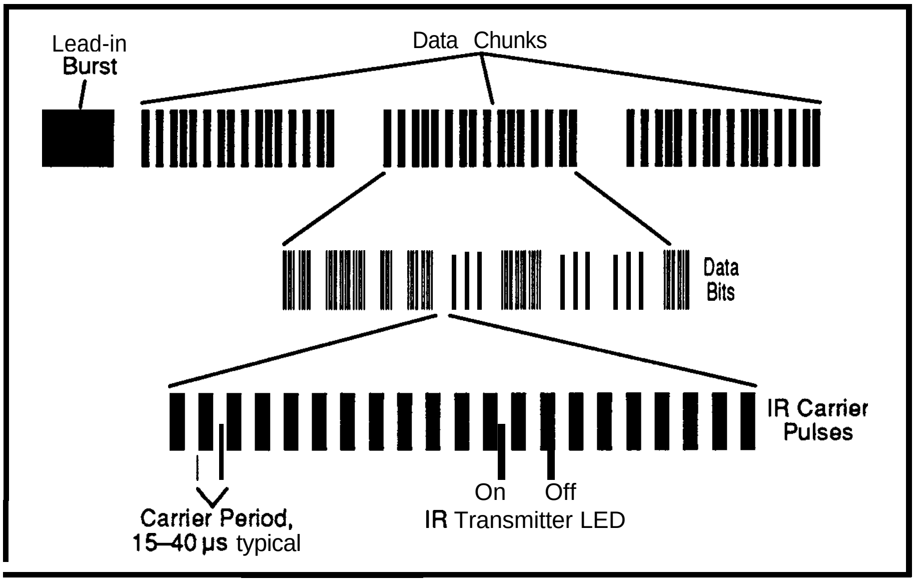


Figure 2—*Although remote controls use many different techniques to encode their information into the infrared signals, this sketch illustrates some of the common features.*

Most infrared remote control units use modulated IR signals, as shown in Figure 2. An IR signal is made up of large chunks (for lack of a better term) containing bits, which are bursts of IR pulses at a (usually] constant carrier frequency. The first chunk may have a long, lead-in burst to wake up the receiver. In some cases, this burst pulls the link out of a power-down deep sleep mode, but usually it is just a way to distinguish an incoming signal from the background clutter.

Each manufacturer uses a different scheme to stuff bits into the IR signal. In fact, two remotes from the same manufacturer may use different encodings, although this inconsistency is less of a problem now that complex A/V systems are the norm rather than the exception. In principle, this variability reduces the chance that an IR signal will trigger the wrong remote, but in practice, it's a source of confusion for those of us trying to make sense of everything.

Some [mostly older] remotes use unmodulated pulses, simply turning their IR LED on to send a pulse and off when it is done. These units tend to be more sensitive to interference from stray IR because there is no way to distinguish a glitch from a real bit. However, even these units do not have a universal bit encoding method.

Photo 1 presents a rogue's gallery of IR signals taken from a shopping-bag-full portion of Steve's remote control collection. Notice the different horizontal sweep speeds required to capture the signals, the number of bursts in each chunk, the timing variations, and the characteristic some

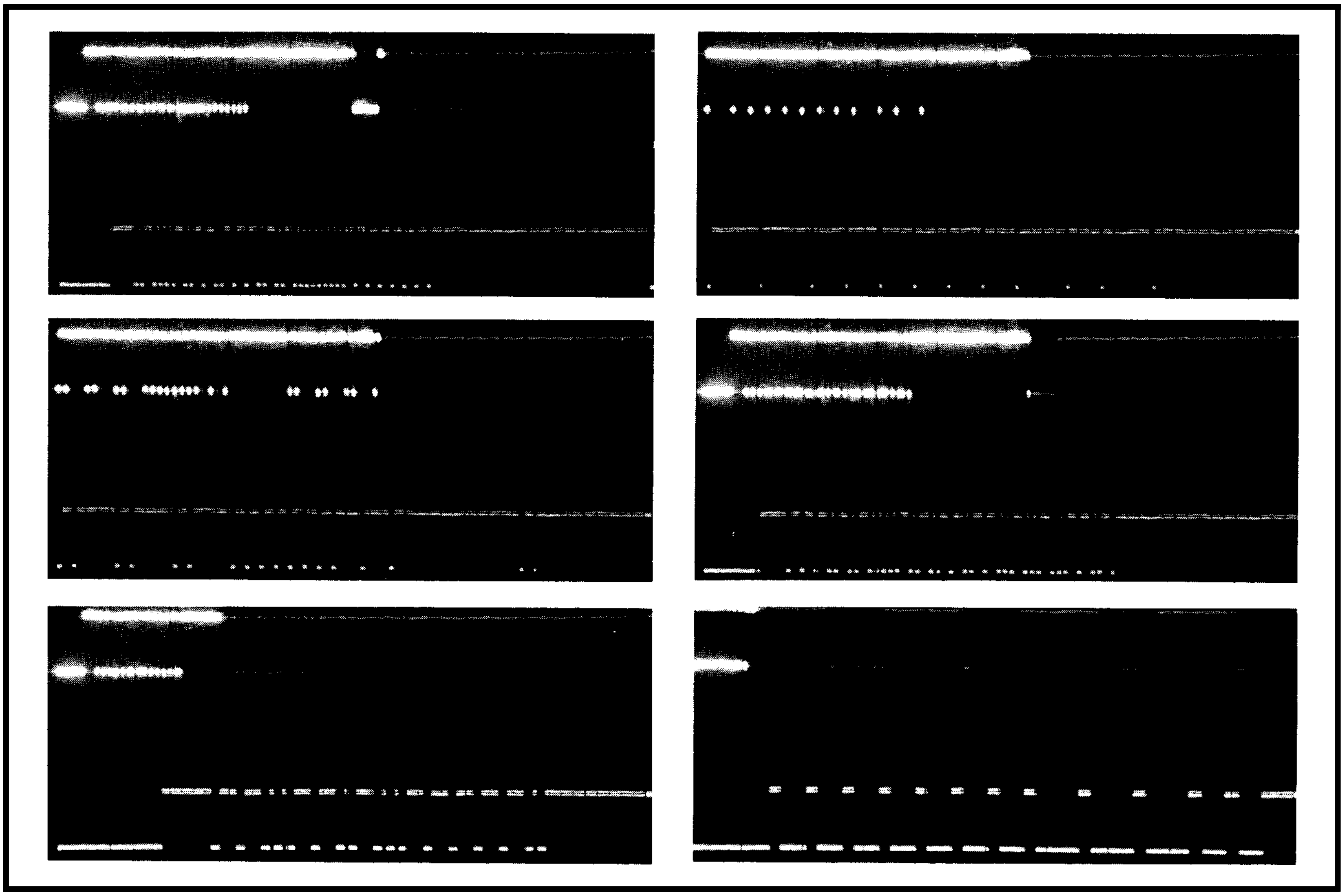


Photo I-Common *hand-held* ***IR remote control units*** *use a* ***vast*** *army olcoding* ***schemes*** *and timings.* ***The scope*** *was set* ***for 20*** *ms* ***per*** *division on he fop* ***trace*** *in* ***all*** *six* ***photos.*** *The* ***bottom trace*** *was* ***set for 10*** *ms* ***oer*** *division in* ***the top four photos,*** 5 ***ms*** per *division in* ***the bottom left photo,*** *and* 2 ***ms per*** *division in he* ***bottom right photo. Also note*** *he* ***differences*** *in* ***repeat rate*** *among he* ***remotes.***

remotes have of not repeating the same chunk over and over.

How to capture, store, and replay this wonderful variety of signals?

GROUND RULES

The IR-Link board "sees" infrared signals through a Sharp IS1U60 receiver that expects bursts made up of IR pulses repeating at a 38-kHz rate. While it can handle other carrier frequencies (roughly 25-60 kHz for very strong signals), its internal bandpass filter reduces the sensitivity, so the maximum distance drops off dramatically as the frequency varies from 38 kHz. The output is a low-going pulse starting 50-100 µs after the IR burst begins and ending 100-300 µs after the signal shuts off. The specification sheet is silent about actual timing limits, but it's reasonable to assume the turn-on and turn-off delays depend on the signal strength, carrier frequency, background light level, and, most likely, the phase of the moon.

Steve designed the IR-Link's output circuitry to produce the data format used by the MC145030 remote control encoder-decoder chip. The Manchester bit cell timings are controlled by firmware interrupt routines, and a 38-kHz oscillator modulates the outgoing signal. Bypassing the modulation is impossible, although a trimpot adjusts the modulation frequency around 38 kHz.

In effect, the hardware converts between real-world modulated IR bursts and firmware-timed binary pulses. As long as the remote control signal looks roughly like those shown in Photo 1, the firmware should be able to both record and reproduce it.

Some restrictions exist, though. Unmodulated IR signals, such as those used by many General Instrument (Jerrold) CAT'V decoder boxes, cannot be received by the IS1U60 or transmitted by the IR-Link hardware. Signals modulated with carriers far from 38 kHz may not pass through the IS1U60 without timing distortion and may also fall outside the IR-Link transmitter adjustment range.

Each IR burst must include several 38-kHz modulation cycles, which means that it must be longer than a few hundred microseconds. No upper limit exists, but based on the remotes I've examined, a few tens of milliseconds is reasonable. The IS1U60 data doesn't include precise timings, but it does respond to signals in that range.

Because a single cycle of 38-kHz modulation requires about 26 µs and the IS1U60 response time specifications are extremely vague, recording the burst times to the exact microsecond is pointless. As you will see later, this flexibility allows a data compression method capable of reducing the stored data by roughly a factor of two.

GOING ON RECORD

Recording a IR signal using the MCIR-Link is simple, at least in principle. You start a timer on the leading edge of the first pulse and record the time of each successive edge. The two pairs of times for each pulse (the ON and OFF durations) should be sufficient to reproduce the pulse. Unfortunately, the 8031 does not include a timer that can be controlled by an external signal in that manner, so it's done with firmware.

Because I was already using Timer 0 to send and receive the IR-Link's MC145030 pulses, adapting it to the MCIR-Link functions made sense. Timer Mode 1 selects a 16-bit up-counter clocked at 1.0851 µs (for an 11.0592-MHz crystal) and the TR0 bit enables and disables the clock signal to start and stop the timer.

Starting is no problem; just wait for an input pulse from the IS1U60. This step is somewhat complicated because you do not want to hang the system if you decide not to send a signal, so I used a short loop that bails out if a character shows up at the serial port. But as the saying goes, "The trouble with doing nothing is it's hard to know when you're done for the day." When recording IR signals, the trouble is there's no good way to tell when the signal is finished!

After examining all the signals available from the sack of remotes, I decided the only reasonable method was to allow a prolonged period of "silence" to mark the end of a signal. Unfortunately, several remotes insert delays up to 200 ms between some IR chunks. Given that Timer 0 is clocked at a fixed 1.0851-µs rate, I had to record 19 bits of timer information to get 569 ms of elapsed duration. That's three bytes per edge, and I had to keep track of the timer overflow while waiting!

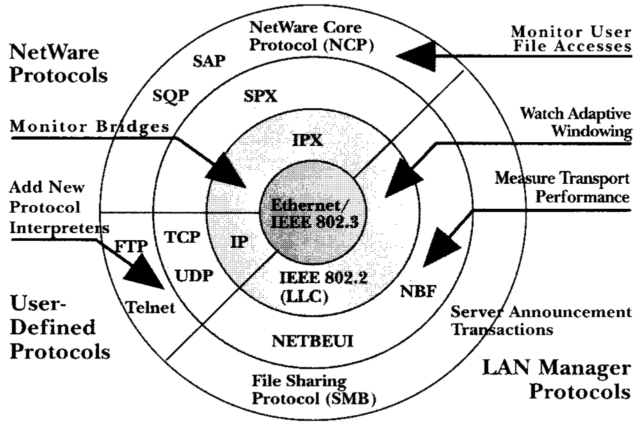
Obviously, there must be some upper limit on the number of pulses in each signal. I picked 1000 pulses, which seemed enough by far for any reasonable signal, and allocated a 3K-byte buffer in the MCIR-Link's static RAM. Equally obvious, you should press the remote control's key just enough to generate a complete signal; filling the MCIR-Link's memory with repeated bursts is not a good idea.

Listing 1 shows the heart of the "recording" routine. The RR_cLev variable counts the number of bytes in the serial port receiver ring buffer; if it remains zero, there are no incoming characters. When the IS1U60 detects an IR signal, the IRECEIVE input bit goes low and the loop exits into the timer capture section.

Although the 8031 was designed (and is still touted) as a "Boolean processor," there are a few gaping holes in the instruction set. The exclusive-OR bit operator is conspicuous by its absence, leaving no fast, clean way to detect a single bit change! As a result, two separate "wait for next edge" loops also keep track of Timer 0 overflows and check for the end of the signal.

After an edge occurs, the code shuts off Timer 0, stores both timer bytes as well as the current overflow count, restarts the timer, and checks for the end of the buffer. If all goes well, it returns to the top of the loop and waits for the next edge.

I've listed the duration of each instruction in the three loops to show the overhead involved. Ample time is available after each edge to record the data because the IR pulses are a few hundred microseconds long. However, the code will eventually have to account for the time that transpires while the timer is stopped to record each time stamp, as well as the average loop time overhead.



SIGNAL SQUASHING

The only hardware change required for the MCIR-Link function is to substitute a 32K-byte RAM for the 8K unit on the IR-Link board. The MCIR-Link code uses the lower 8K for variables, capture and replay buffers, and so forth, leaving 24K bytes for the remote control IR signals.

Simple division says that if typical signals have 100 pulses at 6 bytes per pulse, then that big RAM has room for about 40 signals. Obviously some data compression is in order!

While deploying an armada of signal processing algorithms against the captured signals would be nice, my plans were straitjacketed by three simple constraints. Whatever method I picked must work reasonably well for all remotes using modulated IR, not take forever to get working, and not displace any IR-Link functions.

I knew that the only practical way to produce accurate arbitrary pulses was to load Timer 0 with a value and wait for it to time-out, because the 803 1 has no dedicated pulse generation hardware. Thus, the first data reduction step was to convert the captured times from absolute to relative by subtracting each entry from the previous one (working backwards through the array, naturally!).

The IS1U60 timing is a little more complex than you might expect because the turn-on and turn-off delays for each pulse are not equal. In other words, although the sum of the ON and OFF times is correct, the pulse appears to be ON longer than it really is. My experiments showed that the correction is about 220 µs for strong signals. The code subtracts that amount from the ON time and adds it to the OFF time as part of the absolute-to-relative time conversion.

As I mentioned, an adjustment is also needed to account for the loop time overheads. However, I managed to match the capture and reproduction loop time overheads, so there was no need for an explicit fudge factor. In effect, the code loses just as much time acquiring the signals as it does playing them back. Think about it.

The next step is to divide all these times by eight, which ensures that the

Listing 1—*This pair of loops waits for the first IR signal, then begins capturing 3-byte time stamps for each edge.*

```
• --- wait for an IR signal or cancel keystroke

     SETB IRECEIVE          : ensure input mode...
     CLR  FOFLAG            ; indicate no timeout

?Rawl
     MOV  A.RR_cLev         ; 1 monitor serial character count
     LJNZ ?RawDone          ; 2 and bail out if any show up
     CPL  HEARTBEAT         ;1 indicate we are alive
     JB   IRECEIVE.?Rawl    ; 2 high with no IR signal
     CLR  FlagMCSaveIR      ;1 force memory low to match

*--- hold off interrupts and start timer at zero

     PUSH IE                ;2 save existing interrupt state
     CLR  EA                ; 1 shut them off entirely

     CLR  TRO               ; 1 reset the timer
     MOV  TLO,#0            ; 2
     MOV  THO,#0            ; 2
     CLR  TFO               ; 1
     SETB TRO               ; 1

*--- repeat for each time entry in array or until signal times out

     MOV  B,#0              ; 2 clear wrap counter accumulator

?RawLp
     JB   FlagMCSaveIR.?RawUO  ; 2 which bit did we have last time?

?RawDO
     CPL  HEARTBEAT         ;1 indicate we are alive
     JNB  TFO,?RawD1        ;2 check for timeout
     CLR  TFO               ; 1
     INC  B                 ; 1
     JB   B.3,?RawTO        ; 2 time out after 8 wraps
?RawD1
     JNB  IRECEIVE,?RawDO      ; 2 spin if still down
     SETB FlagMCSaveIR      ;1 remember new bit state
     SJMP ?RawEdge          ; 2

?RawTO
     ORL  ErrFlags,#$20     ; record for debugging
     SETB FOFLAG            ; and force loop exit
     SJMP ?RawEdge          ; after normal recording

?RawUO
     CPL  HEARTBEAT         ;1 indicate we are alive
     JNB  TFO,?RawU1        ;2 check for timeout
     CLR  TFO               ; 1
     INC  B                 ; 1
     JB   B.3,?RawTO        ; 2 time out after 8 wraps
?RawU1
     JB   IRECEIVE,?RawUO      ; 2 spin if still up
     CLR  FlagMCSaveIR      ;1 remember new bit state

*--- edge detected, store current timestamp

?RawEdge

     CLR  TRO               ;1 freeze timer
     MOV  A,B               ;1 pick up wrap counter
     MOV  C,TFO             ;1 incorporate possible wrap
     ADDC A,#0              ; 1
     MOV  B,A               ;1 save it for later
     MOV  R4,TLO            ; 2
```

(continued)

Listing **1—continued**

```
      MOV  R5,THO          ; 2
      CLR  TFO             ;1 clear timer wrap flag
      SETB TRO             ;1 and restart it

      MOV  A,R4            ;store the count in the buffer
      MOVX [DPTR],A
      INC  DPTR
      MOV  A,R5
      MOVX [DPTR],A
      INC  DPTR
      MOV  A,B
      MOVX [DPTR],A
      INC DPTR

*--- decide if we are done yet

      JB    FOFLAG,?RawEnd      ; force exit on timeout

      DecRR            R3,R2
      JNZRR            R3,R2,?RawLp
```

data will fit in two bytes rather than three. The division, which is implemented as a bit shift, preserves the remainder and adds it to the next time. Thus, each regenerated time may be in error by up to 7 µs, but there is no cumulative error.

Most of the remote control signals I measured started with a long lead-in pulse and a long delay, then had relatively short bursts in each successive chunk. Chunks might be separated by long pauses, but there were also few cases where long and short delays were mixed together. A simple run-length encoding compression scheme seemed to offer the best tradeoff of complexity and efficiency.

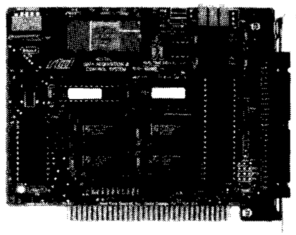
The division by eight means that time intervals less than 2.21 ms (255 x 8 x 1.085 µs) require only one byte. Values longer than that require two bytes, and no values bigger than that exist (by definition!). Most pulses are shorter than 2.2 ms, so much of the data can be squashed into runs of single-byte times.

I decided to work with pulses rather than individual times. If either time value required two bytes, I assigned four bytes to the pulse. If both could fit into single bytes, the pair took up only two bytes of stored data. In practice, this method works reasonably well, but of course there are some remotes that pair short pulses with long delays.

Figure 3 shows the final data for an actual captured IR signal. The record contains four elements: the record number, the overall pulse count, four bytes for the lead-in pulse, and runs of one- and two-byte pulse times preceded by a one-byte pulse count for each run. The counts may be zero if the data requires more than 255

```
01 00               Block ID Number: 1
1200                Number of pulses: 18
CO 03 F2 01         First pulse ON and OFF times: 03CO ON, 01 F2 OFF
10                  Number of pulses in one-byte format: 16 (32 bytes total)
38 B9 38 B9 37 40 37 40 36  the data...
 BA 38 B8 38 3F 37 B9 39 B8
 38 3F 37 40 37 B9 38 B9 38
 B8 38 B8 39 3F
01                  Number of pulses in two-byte format: 1 (four bytes total)
3800 EA ED          Pulse ON and OFF times: 0038 and EDEA (final delay is quite long)
00 00               End markers: two zero bytes
```

Figure 3—*The* firmware *stores IR* *data using* *run-length* *compression* b *encode* time *values* in he *least* number *of bytes.* This record was captured from *a remote* ***control that uses*** *a bng lead-in* ***pulse followed*** *ty* ***several shorter pulses.***



successive pulses of either type or if the data starts out with more than one long pulse. A pair of zero counts marks the end of each block.

Your mileage will vary, but typical data records weigh in at 70 to 150 bytes, so you may be able to fit 200 IR signals into that RAM, which is probably enough to get you into serious trouble. Think of this collection as the ultimate universal controller with no k&pad!

The original IR Master Controller included a character *string* with each IR signal, but that unit was intended for stand-alone operation and sported a keypad and LCD panel. The MCIR-Link is designed for computer-driven applications, so I felt that any strings should be stored in the system controller. Each IR signal is identified by a record number between 1 and 999 that takes up only two bytes in the data record.

Although numbers may be cryptic, they suffice for a simple row-column key mapping layout. For example, call upper-left key 11, the key to its right 12, the third key in the fourth row is 43, and so forth. This system breaks down for some recent A/V controllers with shuttle control knobs, but it works like a champ for remotes with vast rectilinear arrays of tiny keys. If you intend to store a zillion keys, write a program on your PC to keep track of 'em, please!

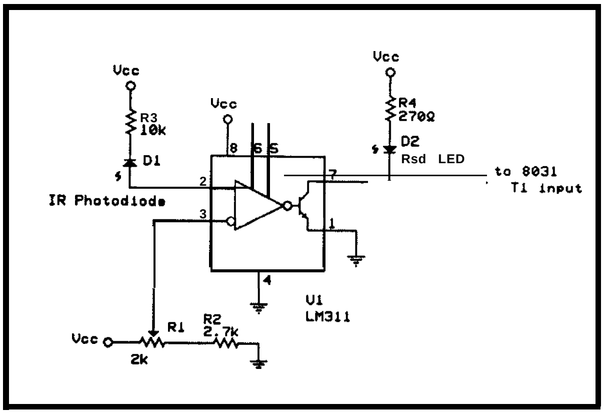


Figure 4—*The IRSAMPLE.C program uses this circuitry to capture raw IR signals from a remote control without the* processing *imposed by* he *normal IS1U60 receiver.* Adjust *R1* so *the* LED is *off under ambient light* and WI *for an IR signal.*

PLAYING DATA

By comparison, playing back the stored data is straightforward. The data records are stored nose-to-tail in RAM, preceded by a two-byte record count and followed by another pair of zeros, so the firmware simply scans through the records to find a matching record number.

Although I could have included an overall record size in each record, I found that scanning through each record and decompressing all the data requires only about 150 μs per pulse. Again, your mileage will vary, but scanning to the last of 100 records, each with 30 pulses, might take 400 ms. While this may sound excessive, remember that typical IR signals weigh in around 100400 ms themselves!

The data expands back into the 3000-byte buffer used to capture it in the first place, with each value occupying the low two bytes of each three-byte entry. The values are then multiplied by eight to recover clock cycles, which puts the timer overflow count in the high-order byte. Because the 8031 timers count upward, the time values must be two's complemented: 0123 hex becomes FEDD. Finally, to allow a simple DJNZ loop, the number of timer wraps is incremented.

Once the data is ready, the code simply fetches each time value, loads the two low bytes into the timer, starts it, and counts down the number of timer wraps found in the high byte. As I mentioned before, the recording and playback loops have similar overhead, so the time lost in the former is made up in the latter.

That's all there is to it!

WELL, ALMOST

The good news is that the IS 1U60 responds to nearly any modulated IR signal, regardless of carrier frequency, as long as you hold the remote within a few inches of the receiver's lens. Remember that the 38-kHz bandpass filter decreases sensitivity for off-frequency signals, but if you have enough firepower, it doesn't matter how bad the response is.

The bad news is that the IS1U60 also responds when hit with singleton

Listing 2-This *section of code from IRSAMPLE.C captures he raw IR input signal from We circuit shown* in *Figure 4 and stores if* in *External RAM at the address formed by P2 and RO.*

```
        MOV  A.#HIGH(IRBuffer); point to buffer
        INC  A                  ; align to next 256 byte boundary
        MOV  P2.A
        MOV  RO.#$FF            ; start from the top

?IRWait
        JB   IR_INPUT.?IRWait   ; wait for input bit

?IRTop
        MOV  C.IR_INPUT         ; 1 fetch input
        RLC  A                  ;1 add to byte
        MOV  C.IR_INPUT         ; repeat for full byte
        RLC  A                  ; at two cycles per sample
        MOV  C.IR_INPUT
        RLC  A
        MOV  C.IR_INPUT
        RLC  A
        MOV  C.IR_INPUT
        RLC  A
        MOV  C.IR_INPUT
        RLC  A
        MOV  C.IR_INPUT
        RLC  A
        MOV  C.IR_INPUT
        RLC  A
        MOVX [RO].A             ; 2 save in external RAM
        DJNZ RO.?IRTop          ;2 step to next sample
```

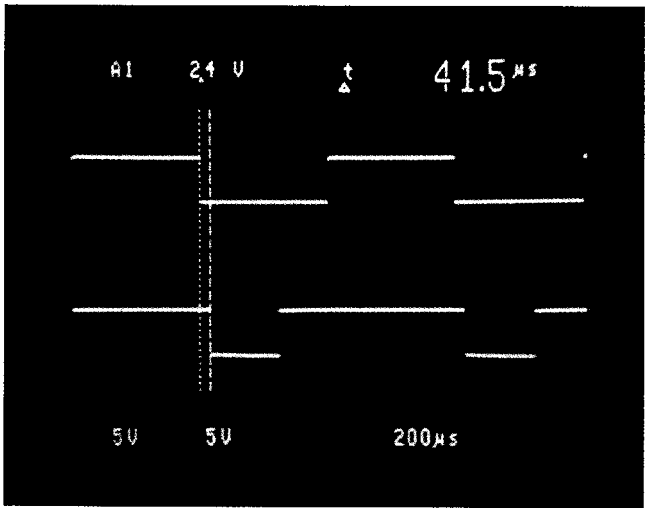


Photo 2—*The IS1U60 can be fooled ty single pulses from unmodulated remotes, so differentiating between good data and trash can be difficult at times.*

pulses from unmodulated remotes. Photo 2 shows the response to a low-frequency square wave: a 300-µs pulse each time the IR LED goes ON! I have also seen clusters of two or three pulses for a single unmodulated input pulse. Distinguishing trash from treasure is difficult when they look alike.

I used an oscilloscope and logic analyzer to classify the remotes for this project, but I realize many of you don't have access to that kind of test equipment. To help you decide if MCIR-Link will work with your remotes, I wrote a utility program to capture raw IR data and dump it to the serial port in logic-analyzer format. IRSIGNAL.C runs on a standard IR-Link board [with an 8K RAM installed), but it requires an IR photodiode to "see" the raw IR signal. Figure 4 shows the circuitry required for this addition. T1 is normally used for the carrier frequency calibration, so make sure you remove IR-Link jumper JP6 as part of the modification to prevent confusion.

Listing 2 is the core of IRSAMPLE's code to capture 8 bits of data and store each byte into external RAM. Two cycles are needed to capture each bit, two to write a complete byte into RAM, and two more to tick the counter or address register. I think this loop is the fastest one to capture external data that you can pull off; suggestions and improvements are welcome! Note that the bytes are stored "backwards" to allow RO to

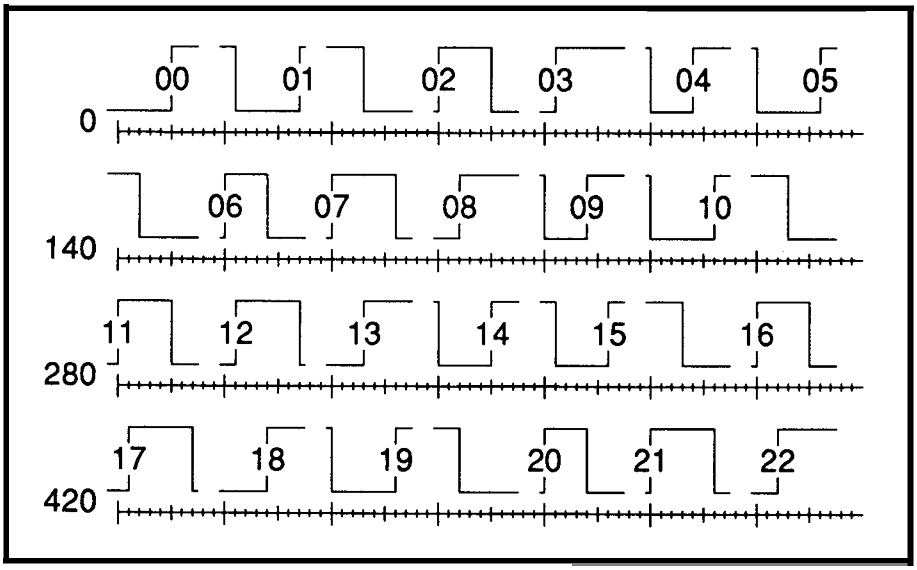


Figure 5—*IRSAMPLE.C captures and dumps the first 5.5 ms of raw IR in a logic analyzer format. The gap in the simulated waveform indicate where the firmware stores the data in external RAM. Each vertical tic represents one sample of 2.17 µs.*

serve as both address and counter with a DJ N Z instruction.

Figure 5 shows the first few lines of a 38-kHz signal dump. Each group of ten vertical tics represents one byte of samples, with the two blank sample periods indicating the dead time required to store the eight data bits in RAM. The first IR burst is long enough that all 255 bytes are filled with carrier, but that is precisely the level of detail I RSAM P L E is intended to present.

You can calculate the modulation frequency by finding the elapsed time for 10 or 20 cycles, dividing to get the period of one cycle, then taking the reciprocal to get the frequency. I RSAM P L E enumerates the rising edges to simplify this process: the elapsed time for pulses 10 through 22 is 544 - 252 = 292 cycles = 3 17 µs. Dividing by 22 - 10 = 12 gives 26.4 µs per cycle or 37.9 kHz.

Many remotes need a few cycles to "get up to speed," so you might want to skip the pulses in the first line or two when you calculate the frequency. You should also avoid using edges immediately after the timing gaps, such as edge 2 in Figure 5, because the firmware was busy storing the previous byte and could not sample the input bit. That transition may have actually occurred any time in the previous two sample times, so precise timing isn't possible.

Unmodulated IR signals will appear to be straight lines, low while the LED is ON and high for the OFF times. Remember that the MCIR-Link hardware and firmware can't handle those signals.

ON THE AIR

The MCIR-Link EPROM hex file (MCI RB . H EX) is available on the Circuit Cellar BBS for your noncommercial use. The source code is not available, but it may be licensed from Circuit Cellar Inc. (not INK). Contact them for details.

You may also download the complete I RSAMPLE.C source code and I RSAMPLE. HEX for your IR-Link board to check your remotes for MCIR-Link compatibility. You will need the current version of Dunfield's Micro-C compiler to recompile the code.

Pure Unobtainium has most of the odd parts you need to build the IR-Link and MCIR-Link projects, including the IR photodiode and LM3 11 for the I RSAMPLE. C program. ■

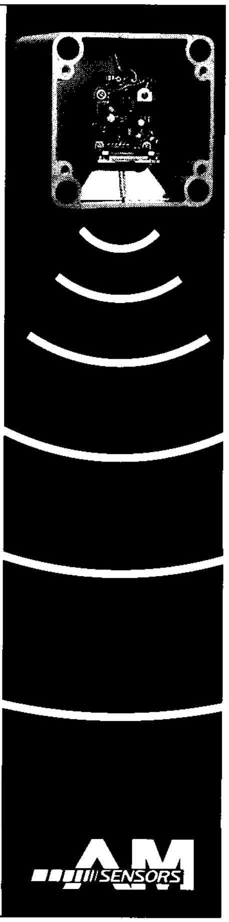
Ed Nisley is a Registered Professional Engineer and a member of the Computer Applications Journal's engineering staff. He specializes in finding innovative solutions to demanding and unusual technical problems.

SOURCE

Pure Unobtainium
89 Burbank Road
Tolland, CT 06084-2416
Fax/Voice (203) 870-9304

I R S

419 Very Useful
420 Moderately Useful
421 Not Useful



X-10 Interfacing with PLIX

FROM THE BENCH

Jeff Bachiochi

The TW523 might be considered by some to be the perfect X-10 computer interface. The complicated timing can be daunting to many, though. Throw a PLIX chip at the circuit and your troubles may be solved.

Oet me see the hands of all the engineers out there who still enjoy getting down on the floor and building forts and castles out of blocks or Legos with the "kids." For some of us, doing such an activity with young children is a successful enough smoke screen to save us from the label "eccentric." Call me what you like, but building a fantasy empire out of wood is just as rewarding as designing a product.

I enjoy using the rules of gravity and balance with good old wooden blocks, and building with Legos has its own reward even though working with them is a bit different. Every task has its own set of rules. An understanding of these rules and a bit of imagination is all you need to do well. The same concept used in castle construction applies to product design. I guess that's why I enjoy being an engineer. If you choose the correct mix of components, your design will withstand the test of time (or gravity). If not, well that's how we got the term "smoke test."

One of the most widely discussed topics here in the ***Computer Applications Journal*** (and on the Circuit Cellar BBS) is home control. X-10 control is a favorite subject within this topic. Let me show you how the new PLIX (Power Line Control for X-10) chip can ease you into X- 10 control without even breaking a sweat, and why you should consider using a chip to handle X- 10 communication in your next design. I think you'll find this blend a perfect one.

WHERE WE'VE BEEN

Many of us have been using X-10 modules for appliance control for over ten years. X-10 Powerhouse remains unchallenged as the leader in power-line carrier transmission control systems. CEBus's PLBus may render X-10's code transmission format obsolete, but its large, installed user base will prevent that from happening any time soon. (For more information see the assorted CEBus articles by Ken Davidson in issues #10, #15, and #21 of ***Circuit Cellar INK.)***

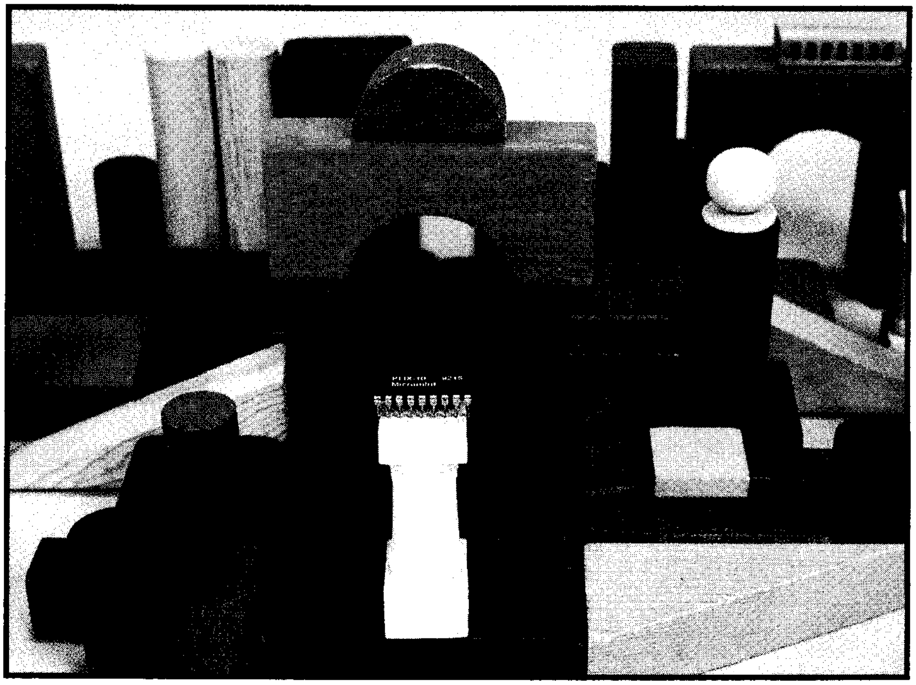


Photo 1—*PLIX simplifies the task of X-10 interfacing by handling all the complicated liming.*

Housecode	Data	Function	Data	Function	Data
A	**6**	1	6	All Units Off	16
B	7	2	7	All Lights On	24
C	4	3	4	On	20
D	5	4	5	off	26
E	6	5	6	Dim	16
F	9	6	9	Bright	26
G	10	7	10	All Lights Ott •	22
H	11	6	11	Extended Code •	30
I	14	9	14	Hail Request •	17
J	15	10	15	Hail Acknowledge •	25
K	12	11	12	Preset Dim Low •	21
L	13	12	13	Preset Dim High •	29
M	0	13	0	Extended Data •	19
N	1	14	**1**	Status=On • *	27
0	2	15	2	**Status=Off** • *	23
P	3	16	3	Status Request • *	31

- • denotes not supported by any current X-10 module
- • * denotes supported only by the RR501 RF gateway module

Tabb I-X-10 ***transmissions* ahvavs ,contain a *housecode* and a *function* wde. *The function* code *either selects*** a ***particular* module or invokes *some action.***

The X-10 system can handle up to 256 unique module addresses, but the manual controllers marketed by the company can deal with only eight or sixteen of those at a time. If you have more than sixteen modules in your house, putting them under computer control is the next logical step in making them easier to use.

Although some companies developed X- 10 products in the past that used a microcontroller, there was no clean way to connect the controller and the power line. This void limited product development to those companies who could afford UL and FCC testing of such interfaces. Several years ago, X-10 finally eliminated this bottleneck with the introduction of the PL513 computer interface module.

The PL5 13 provides the designer with an optoisolated interface to the power line. A signal coming from the module tells the computer when each power-line zero crossing occurs. Additionally, a signal to the module controls when the 120-kHz carrier is applied to the power line. The user is the one who must send properly formatted data to the PL513 in synchronization with the zero crossings. [See "Power-Line-Based Computer Control" by Ken Davidson, ***Circuit Cellar INK 3,*** May/June 1988.)

One other void had to be filled before X-10 became more practical. In addition to sending commands, the capability to listen for them was important as well. X- 10's TW523 corrected this omission. This module error checks any data received and sends it to the computer, again during zero crossings. Like transmissions, the user is responsible for decoding the received binary data. (See "The X- 10 TW523 Two-Way Power Line Interface" by Ken Davidson, ***Circuit Cellar INK* #5,** September/October 1988.)

PRE-PLIX GYMNASTICS

Confusion as to what makes up an X- 10 transmission still exists, even with these two interfaces. The X-10 format consists of a serial data stream containing a start code, a housecode, and a function code. Table 1 shows valid housecodes and function codes.

Each bit of data in the bit stream consists of six **1-µs** time periods. Each time period is 2.778 **µs** apart, or six per **60-Hz** line cycle (Figure la). (If the first period is synchronized to the zero crossing of the **60-Hz** line, then the remaining five time periods will be in step with each zero crossing of a **three**-phase power line and, in principle, be detected on systems with three-phase power.] A data bit with a value of "1" is sent as On-On-On-Off-Off-Off (Figure **1** b), whereas a bit value of **"0"** is sent as Off-Off-Off-On-On-On [Figure lc). "On" is the presence of a**1-µs** pulse of the **120-kHz** carrier and "Off" is its absence. Thus, one bit time equals one **60-Hz** cycle.

Notice that each data bit is sent as three copies of the data bit plus three copies of the opposite logic state to allow simple error checking. Noise occurs equally on each half of the power line's cycle, whereas **120-kHz** bursts must be present on only half the cycle to be considered legal. This aspect is true for all house and function code data bits . **The** start code uses a different format. It is always the same two-cycle sequence: **On-On-On**-On-On-On for the first cycle and **On**-On-On-Off-Off-Off for the second (Figure **1d**).

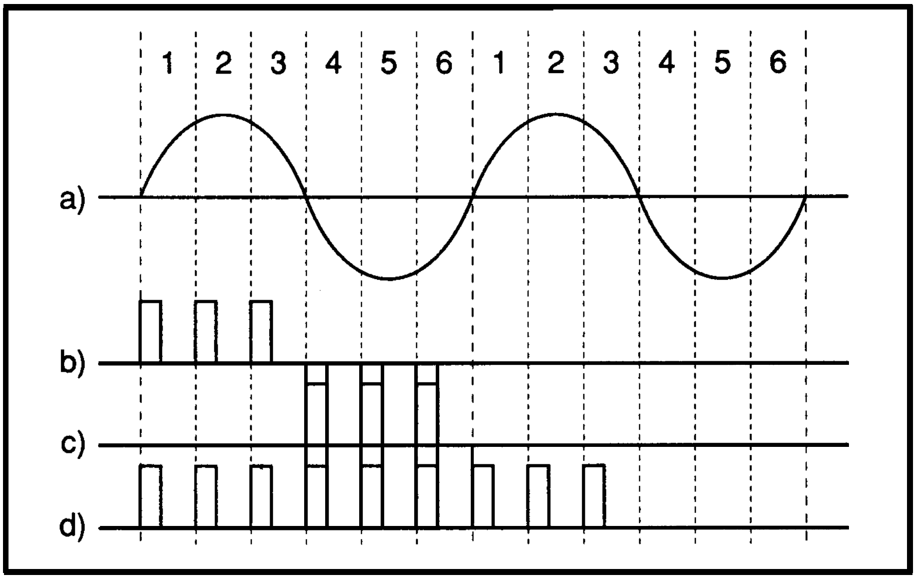


Figure l-a) ***X-10 transmissions* are *synchronized*** b ***the*** AC power ***line zero crossings. b) A 1 data bit is represented*** *by three* 1-ms ***bursts*** of ***120 kHz*** *signal,* ***followed*** *by silence during he next half* ***cycle.*** *c) A 0 data* ***bit*** *is just the opposite, with the bursts* ***occurring*** *during he* ***second half*** *of* ***the cycle.*** *d)* ***Every transmission begins*** *with a unique* ***start*** *code, which lasts* ***two full*** *AC* ***cycles.***

Listing 1—*Sending and receiving X-10 commands can now be done even using interpreted BASIC.*

```
10 PORT1=09FH: REM P1.0-4 AS I/O. P1.5-6 AS OUT, P1.7 as IN
20 V=O: GOSUB 1000: GOSUB 1000: GOSUB 1000: V=31:GOSUB 1000: REM
   SYNC
30 PRINT 'Hit 'W' to write to PLIX. 'R' to read from PLIX'
40 G=GET
50 IF (G<>57H.AND.G<>77H.AND.G<>52H.AND.G<>72H) THEN 40: REM RrWw
60 IF (G<>57H.AND.G<>77H) THEN 180: REM Rr

70 REM Write Routine
80 INPUT 'Write which house code (0-15)?",V
90 IF (V<0.OR.V>15) THEN 80
100 GOSUB 1000
110 INPUT 'Write which function code (0-31)?",V
120 IF (V<0.OR.V>31) THEN 110
130 GOSUB 1000
140 INPUT 'Repeat code how many times (1-30)?",V
150 IF (V<1.OR.V>30) THEN 140
160 GOSUB 1000
170 GOTO 30

180 REM Read Routine
190 GOSUB 2000
200 IF V>15 THEN PRINT "New'. ELSE PRINT 'Old'.
210 PRINT . House Code =",V.and.15
220 GOSUB 2000
230 PRINT 'Function Code =",V.and.31
240 GOTO 30

1000 REM Write a Value
1010 IF (PORT1.AND.80H)=80H THEN 1010: REM Wait for RDY to drop
1020 PORT1=((PORT1.OR.60H).AND.OEOH).OR.V: REM Set DIR, CS. DATA
1030 IF (PORT1.AND.80H)<>80H THEN 1030: REM Wair for RDY to rise
1040 PORT1=PORT1.AND.9FH: REM Clear DIR and CS
1050 RETURN

2000 REM Read a Value
2010 IF (PORT1.AND.80H)=80H THEN 2010: REM Wait for RDY to drop
2020 PORT1=PORT1.OR.OBFH: REM Set DIR, Clear CS. Data as inputs
2030 IF (PORT1.AND.80H)<>80H THEN 2030: REM Wait for RDY to rise
2040 V=PORT1.AND.1FH: REM Get the lower five bits
2050 PORT1=PORT1.AND.9FH: REM Drop CS
2060 RETURN
```

A normal transmission will take 11 60-Hz cycles: two for the start code, four for the housecode, and five for the function code. In addition, every command should be sent twice. To turn on module A3, send housecode A, function code 3 (for unit 3), then housecode A, function code On.

Taking into account the repeated transmissions and the required three cycles of silence between complete transmissions, you end up with 2 x [(2 x 11) + 3], or 50 cycles, which is close to a full second. I hope you don't have anything else for your processor to do while it's busy watching for zero crossings and turning the 120-kHz carrier on and off. To make matters worse, I haven't even taken receiving into account, where you must check for received data at every zero crossing (using the TW523). I have better things to do with my processing time. There must be a better way.

A BETTER WAY: PLIX

PLM takes the burden of X-10 serial transmission and reception off the processor, performing these functions as background tasks on its own. Simply send the housecode, function code, and the number of times to repeat the command (normally two), and PLIX outputs the proper gating sequences to the PL5 13 or the TW523. If you're using the TW523, you can ask PLIX for the last X-10 transmission it heard. The reply also indicates if the data is new [was received after the last query). If your system is battery backed, you may want to use PLIX's "AC power fail" output pm as a system input that indicates power-line status.

A simple bit-programmable, bidirectional, 8-bit port is all that is necessary to carry on a conversation with PLIX. Two output bits control chip select and data direction {RD/WR), one input bit reflects PLIX's status {RDY/BUSY}, and five bidirectional lines transfer data.

Only five external components are necessary: two resistors (pull-ups for the PL5 13 or TW523}, two capacitors, and a crystal. The 18-pin DIP package requires less than 2 mA at 5 volts to operate. Slightly more is needed when supplying the gating pulse to the optocoupler of the PL5 13 or TW523 during X- 10 transmissions.

USING PLIX

Figure 2 shows the PLIX chip connected to Port 1 of an 8031/8052 processor. Listing 1 shows a sample program that excercises the chip. The code is straightforward and easy to understand because the port is bit addressable and accessible directly through BASIC.

If you're not using an 8052, Figure 3 shows a pair of flowcharts that describe the proper algorithms to use to write to and read from PLIX.

Getting in sync with the PLIX chip after reset and terminating a function prior to finishing are both good practices. Simply send the PLIX three or more 0 data bytes followed by a 3 1. [The 0 values are illegal repeat commands and PLIX will hold in the repeat state. If a 31 is received, which is also an illegal repeat value, the previous commands are flushed and chip resets.)

DESIGNING WITH PLIX

PLIX does not have the speed required to interface directly as an I/O bus peripheral, so it must interface through port bits. As you have seen, interfacing to the 8052's port 1 is easy because each bit is individually programmable for input or output. However, all control does not have to

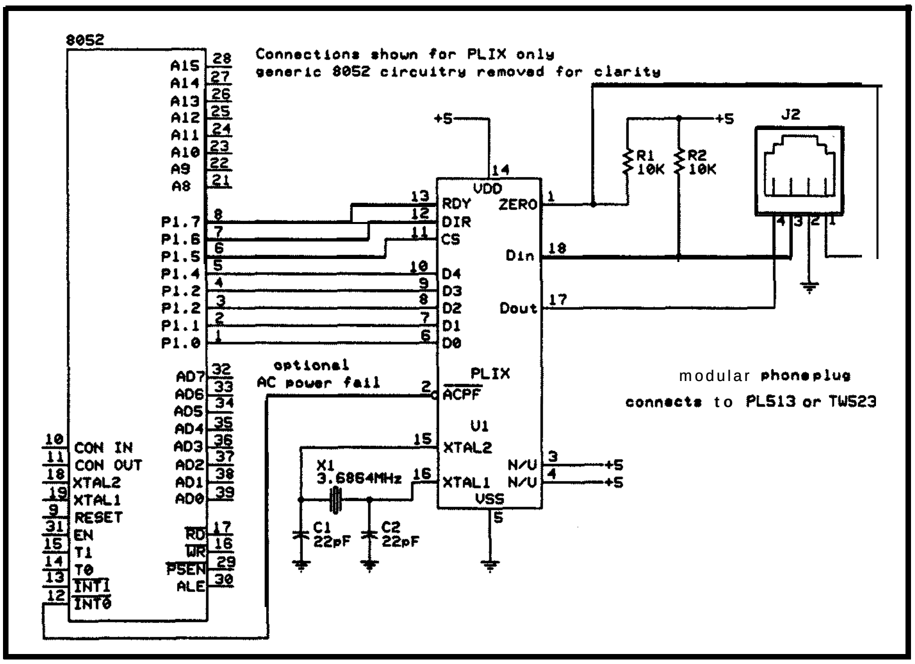

8052
Connections shown for PLIX only
generic 8052 circuitry removed for clarity
A15 A14 A13 A12 A11 A10 A9 A8
P1.7 P1.6 P1.5 P1.4 P1.2 P1.2 P1.1 P1.0
AD7 AD6 AD5 AD4 AD3 AD2 AD1 AD0
CON IN
CON OUT
XTAL2
XTAL1
RESET
EN
T1
T0
INT1
INT0
RD
WR
PSEN
ALE
optional
AC power fail
+5
VDD
RDY
DIR
CS
ZERO
Din
Dout
D4 D3 D2 D1 D0
PLIX
ACPF
U1
XTAL2
XTAL1
N/U
N/U
VSS
X1
3.6864MHz
C1
22pF
C2
22pF
R1
10K
R2
10K
J2
modular phoneplug
connects to PL513 or TW523

Figure 2-Adding PLIX lo a generic 8052 circuit is just a matter of connecting it to the processor's spare I/O port

come from the same port. A minimum of two output bits, one input bit, and five bidirectional bits are needed from any source. If your processor does not have any spare bits, then use a peripheral interface chip, like the 6821, mapped into the available I/O space. Each of its two 8-bit ports is bit programmable.

However you choose to interface with it, the PLIX chip will allow your processor to keep up with all its

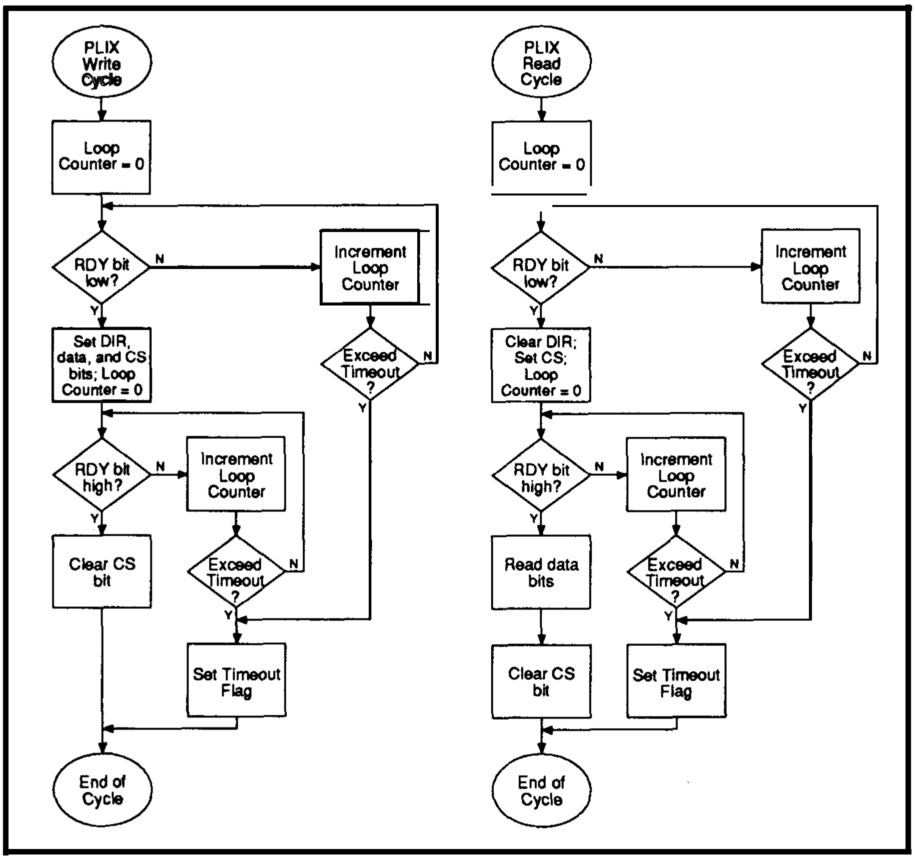


Figure 3-The PLIX read and write cycles use very similar algorithms and rely on handshaking between the processor and the chip.

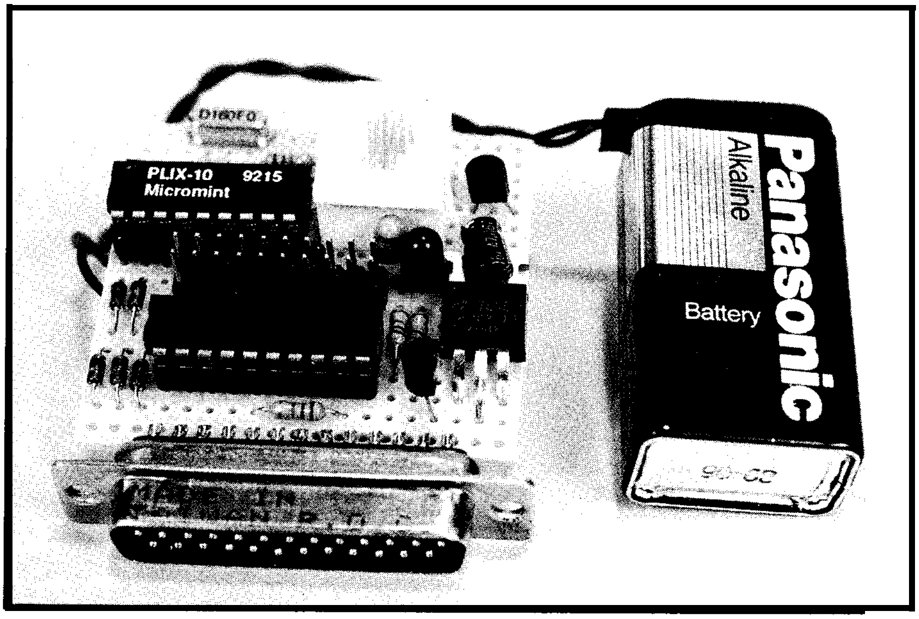


Photo 2—*Due to the PLIX chip's low power consumption, the demo board can be run off a 9-V battery.*

important functions without having to twiddle its thumbs [or bits) during a time-intensive X-10 serial transmission or reception. If I didn't know better, I'd have guessed it was X- 10 who came up with this missing link. Some companies just can't wait for the future; they have to invent it.

PLIX DEMO BOARD

I've put together a little board that will allow you to experiment with this chip [Photo 2). Even though PLIX only needs five external components and can be handwired easily, immediate gratification can be a worthy attribute. A modular phone jack permits direct connection to either the PL513 or the TW523. A short piece of ribbon cable off of the square-pin header simplifies interconnections between PLIX and your port bits. Finally, power is supplied by a 9-volt battery.

If you don't have a controller project in the works and can't directly connect to PLM, but still want to experiment, you are in luck. The PLIX demo board will also interface to your PC's parallel printer port. Because the PC printer port is not a true bidirectional port, the data must be read in through the status port as nybbles, which adds a bit more complexity to the circuit. The low power of the PLIX chip allows battery operation.

The routines written for the PC's parallel port are written in BASIC, but are adaptable to any language, including straight assembly language. The quick evaluation of PLJX is this demo board's purpose. Once you see the benefits of using PLIX, you will likely want to include it in your future X-10 designs. ◼

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.

SOURCE

Micromint Inc.
4 Park St.
Vernon, CT 06066
(203) 871-6170

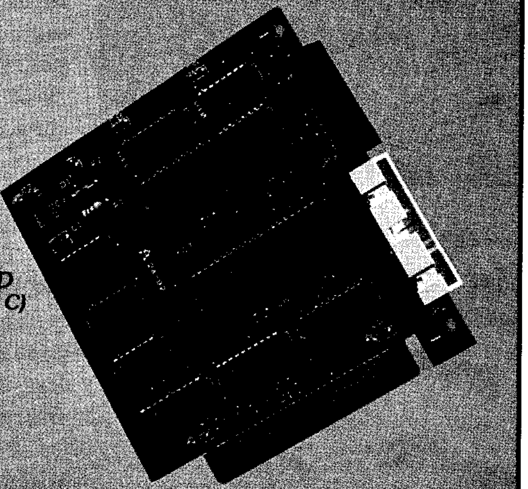
1. PLIX Chip and Data Sheet.... $20

2. PLIX-EKit Evaluation Board. Includes PLIX chip, data sheet, PCB with all components, application note, and PC-compatible software on disk. $29

Add $20 for TW523 X-10 transceiver module (sold with PLIX-EKit only). Shipping extra.

I R S

422 Very Useful
423 Moderately Useful
424 Not Useful



I'm 18.432–and I Like It

SILICON UPDATE

Tom Cantrell

Some people just can't get enough horsepower out of their processors, and Tom is certainly one of them. Zilog's newest speed demon runs the faithful Z180 (HD64180) up to 18.432 MHz and beyond.

Sometime during the '70s, Alice Cooper paid homage to **those with raging** hormones by making his classic statement, "I'm 18 and I like it." Today, many of us yearn for a return to the days of our youth when we were strong and good looking.

Meanwhile Alice, who still wears tights and makeup 20 years later, barely made it out of adolescence before jumping directly into mid-life crisis. Come on Alice, update our generation's anthem-how about "I'm middle-aged and it stinks."

Baby-boomers and the like aren't the only ones having mid-life crises. ICs are having them, too. Chips that once were the darlings of industry have ended up feeble "commodities."

Now a chip, unlike the typical burnt-out Silicon Valley executive, can't turn to Grecian formula, a Nehru jacket, and singles bars for a mid-life "kicker." However, as I will show, old chips and systems can also be rejuvenated.

DANCING AS FAST AS I CAN

The Z80, and its follow-on the Z80 180 (a slightly modified HD64180), certainly qualify as some of the most popular chips of all time. Nevertheless, these parts are starting to show their age.

Now, Zilog has come to the rescue with the new Z8S180. Though fully plug compatible with the existing '180, the 'S180 has a number of new features.

The S stands for the ***static*** design of the chip. The primary benefit of static [as opposed to ***dynamic)*** designs is you can arbitrarily run the clock slowly and, indeed, stop it altogether.

This ability to stop the clock is exploited with the new power-down modes ***IDLE*** and ***STANDBY,*** which supplement the ***SLEEP, IO STOP,*** and ***SYSTEM STOP*** modes of the original '180 (Figure 1). The ***STANDBY*** mode (10 µA) cuts power consumption dramatically when compared to the previous lowest power mode, ***SYSTEM STOP*** (17.5 mA), making the 'S 180 far more suitable for battery-powered applications.

A neat addition, but nobody is going to rave about a new chip that runs slower than ever. However, the new static CMOS design offers another benefit: the 'S180 runs at up to 20 MHz, twice as fast as the regular '180!

Longtime readers know I'm a big believer in clock rate when boosting CPU performance. Let others dabble with the complexities of superscalar, superpipeline, and various other super-duper tricks. I'd rather just crank the clock. So, I grabbed a '180 board and headed for Zilog. Mission (Possible, I hope): double the performance or bust!

NO FREE LUNCH

In my view, the performance of general-purpose computers boils down to bus bandwidth. Yes, architecture, compilers, and system design have an impact, but in today's competitive environment-"religious wars" to the contrary-gaining a significant advantage in these areas is difficult. However, the performance gains of a faster clock come at a price: the need for a corresponding faster memory. Without it, any improvement will be lost to the dreaded wait state.

In fact, boosting the CPU speed without facing up to memory bottlenecks can lead to a somewhat pathological decrease in performance. The reduction happens when the "granularity" of a wait state exceeds the percentage speed-up in the clock.

This scenario is most easily illustrated with RISCs that call for 1-clock bus cycles. In this case, the granularity of adding the first wait state is 100% (i.e., a 1- to a 2-clock bus cycle). Now, what happens if the clock is boosted without faster memory?

Power-Down Modes	CPU Core	On-Chip I/O	osc.	CLKOUT	Recovery Source	Recovery Time (Minimum)
SLEEP	stop	Running	Running	Running	RESET, Interrupts	1.5 Clock
I/O STOP	Running	stop	Running	Running	Programming	
SYSTEM STOP	stop	stop	Running	Running	RESET, Interrupts	1.5 Clock
IDLE†	stop	stop	Running	stop	RESET, Interrupts, BUSREQ	8+1.5 clock
STANDBY†	stop	stop	stop	stop	RESET, Interrupts, BUSREQ	2+1.5 Clock 2+1.5 Clock

Notes: † IDLE and STANDBY modes are **only** offered in **Z8S180**. Note that the minimum recovery time can be achieved if INTERRUPT is used as the Recovery Source.

Figure 1—*Due* b ifs ***static*** *design, the* ***Z8S180's clock oscillator*** *can be* ***stopped completely (STANDBY) for*** *extreme&* ***low power consumption.***

Say the RISC runs at 10 MHz and thus has a 100-ns bus cycle. The current memory has 70-ns access time resulting in zero-wait state operation. Now, Joe Naive-User hears that a hot 16-MHz CPU is available and plugs one in, ignoring messy details such as upgrading memory.

While the placebo effect may cause Joe to think his PC seems a little snappier, the reality is the 16-MHz CPU, with its 66.6-ns bus cycle, is going to need a wait state to work with the existing 70-ns memory. The sad reality is that the hot CPU is running 33% slower (133.3 ns vs. 100 ns) than before the "upgrade."

Fortunately, the RISCer's need for speed has driven IC manufacturers to deliver ever faster memories. In this era of 50- to 100-MHz CPU chips, fast memories are de rigueur.

Unsure of which memories I could scrounge at Zilog, I took the precaution of tweaking the '180 board's BASIC EPROM to take advantage of the '180's on-chip wait state generator (a great feature) to inject the maximum number of three wait states. This stretches memory access to six clocks (three is the minimum), making the bus cycle 300 ns at 20 MHz. Past experience told me that the access time required of the memory is about half the bus cycle, or 150 ns, which isn't a problem even for EPROM and certainly not for the SRAM.

Before heading off, I also wrote a simple test program to allow me to exercise the board with different wait state settings.

HUMAN ERROR

Over at Zilog, '180 board and test program in hand, I sat through a nice presentation describing the 'S180, though I was chomping at the bit to head for the lab.

The helpful folks at Zilog pointed out some of the other key enhancements. One particularly unique feature is the Z8S180's ability to reduce the drive programmably to all of its outputs or just to selected subsets. The effect is to slow the edge rates, thereby reducing radiated ElectroMagnetic Interference (EMI) significantly [Figure 2). Besides keeping the FCC at bay, this feature is especially helpful in wireless communications applications.

Surprisingly, the 'S180 consumes little more power than the '180 even at full-speed operation. Normally, CMOS power consumption is almost linear with the clock rate. If the clock rate is doubled, the power consumption should increase significantly. However, the 'S 180 consumes only 40 mA at 20 MHz compared to 36 mA at 10 MHz for the '180. One simple explanation for this difference is the shrink from a 1 S-micron to a 1.2-micron process. A less obvious reason is that the static design, just as for memories, eliminates the need to refresh internal circuits like the register file.

A programmable divide-by-l or divide-by-2 clock option is a really nice touch Zilog added. The original '180 requires a 2x clock input, so my board had an 18.432-MHz crystal running the CPU at 9.2 16 MHz (the weird clock rates are called for by the '180 on-chip baud rate generator]. When the 'S180 is reset, the programmable clock divide is set to 2; thus, the 'S180 also comes up running at 9.216 MHz. A single `OUT` instruction can then toggle the divide bit to lx and speed things up.

This result means existing '180 designs can be upgraded without changing the crystal! That's good, because I wasn't looking forward to doing the switch; my soldering skills are definitely not improving with age.

As the presentation ended, I thought that this upgrade, a simple change of three chips (the '180, BASIC EPROM, and SRAM), would be about as easy as one can get.

Once in the lab, we dug up an 85-ns EPROM and a 35-ns SRAM. I'd been told that the 20-MHz 'S180 required memories of about 60 ns. Even relaxing that to reflect 18.432-MHz operation, I realized I'd need at least one wait state to deal with the EPROM. No problem. I'd programmed the BASIC EPROM to boot up with the maximum number of wait states and the 'S180 boots at 9.216 MHz anyway, so at least it should come up.

Sure enough, I popped the parts in, hit power, and-success, the board worked! I exercised my test program and everything was solid. The test confirmed the 'S180 was functionally equivalent to the older '180, something that shouldn't be taken for granted when a chip is redesigned.

But now the time was right for the real test, a double-dose of megahertz.

OK, how do I toggle the clock-divide bit!

Oh, simple, just output an 80H to register 3FH.

Inserting an `OUT $3F, $80` at the front of my program, I was ready for blast-off. I entered "`RUN,`" left my finger poised over the return key, took a deep breath, and thought, "Hang on to your hats. Here we go."

Of course, you know what happened when I hit that key. The board just locked up, remaining mute and unresponsive while I pounded on the keyboard. Finally, I gave up hope and reached for the board's reset switch. Hmm...still came up at 9 MHz.

When facing such a situation, an inexperienced user will often jump to the conclusion "the chip is busted." After all, the 'S180 was marked "ENGR SAMPLE" and these things happen when you're on the "bleeding edge."

However, having done my post-graduate work in the school of hard knocks, I know that 99% of the time "the chip is busted" excuse doesn't pan out. An old-timer will sit down and say "What's the best way to figure out what I did wrong?"

After a little head-scratching (heaven forbid I should get out schematics and scope this early on), I began to wonder about some of the '180's on-chip control bits. Hadn't the CPU gone through some revisions to better adapt the bus timing to Zilog peripheral chips? I seem to remember an "I/O compatibility" bit somewhere. I thought I'd better take a quick look at the register definitions. I also thought about the rest of the bits I was setting in register 3FH. Maybe they did something.

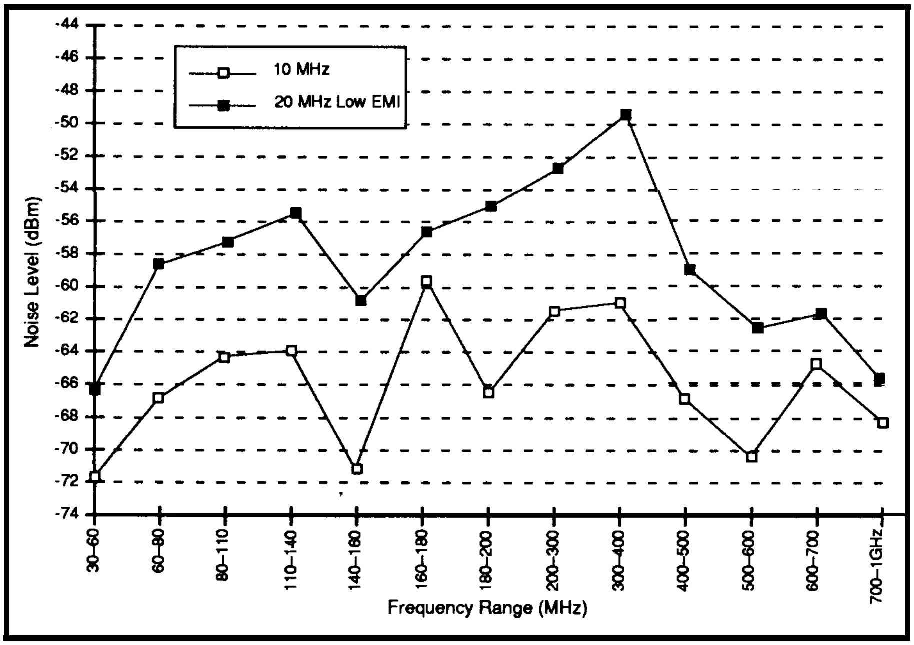


Figure 2-One of the Z8S180's new features is he ability to reduce drive current on some key control lines to reduce radiated EMI. With he feature turned on, a 20-MHz Z8S180 is quieter than the original 10-MHz Z80180 that doesn't have he feature.

Hey, ***register 3FH doesn't have any "clock-divide" bit.***

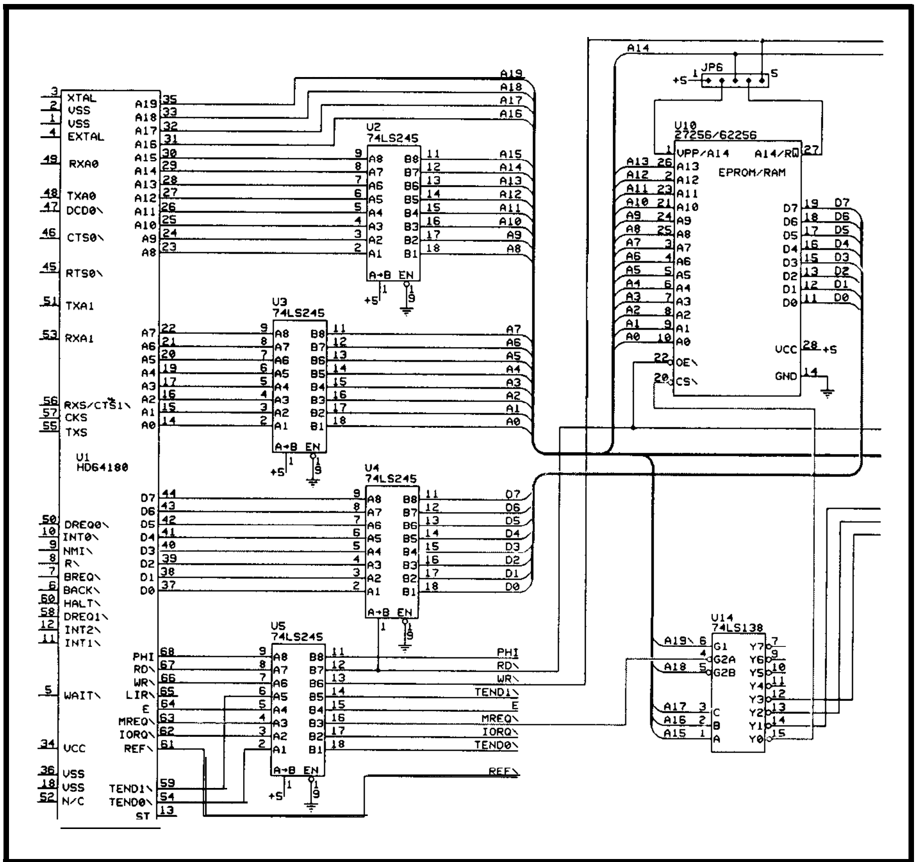


Figure 3-A *sample HD64180 (Z80180) circuit shows the critical timing paths in the memory section.*

Oops-oh yeah, it's in register 1FH.

The bits I was setting in register 3FH happened to have the interesting property of relocating the '180 on-chip I/O addresses. Thus, after the OUT instruction, the software was trying to talk to I/O ports (including the console port) that weren't there anymore!

Changing the, OUT instruction from 3FH to 1FH, I skipped the dramatics and hit'it.

I experienced'an apprehensive moment when garbage appeared on the screen, but it was only a reflex because I'd already anticipated the doubling of all the chip's baud rates, timer constants, and so forth. I calmly switched my terminal software from 9600 to 19200 baud and-voilh!

Exercising my test program, I discovered the board worked pretty much as expected [i.e., it would work with a single memory wait state). I tried zero wait states knowing that chips are often faster than specified, especially at room temperature, but the EPROM couldn't hack it.

I left Zilog happy that despite human error (the cause of most "computer problems"), I had succeeded. However, even as I was driving away, that one remaining wait state started to gnaw at me. . . .

CHIPS IN THE FAST LANE

Zipping down the freeway, I pondered the problem. Yes, state-of-the-art memories exist, but where can I get one? Those puppies aren't exactly a stock item at the local "Ye Old Chip Shoppe."

On autopilot, I registered the red lights stacking up ahead and dove for the off-ramp, planning to take one of the secret short cuts that harried Silicon Valley commuters seek out like rats in a maze.

Though I am skeptical of psychic phenomenon, I wonder if it was just coincidence that my roundabout path took me right by WaferScale Integration!

Hey, didn't I just get something in the mail from these folks about fast EPROMs!

I must have made quite a sight, first careening into their parking lot, then dashing into the lobby waving my '180 board and demanding to speak to the marketing manager. However, the receptionist remained calm, called the appropriate authorities [not the police), and it wasn't long before I was back on the road, this time clutching some real gems, 55-ns EPROMs!

Back at my lab, I programmed a 55-ns chip and plugged it in. I successfully got into high-speed mode and started exercising my test program first with three wait states, then two, then one. Then for the big test: zero wait states!

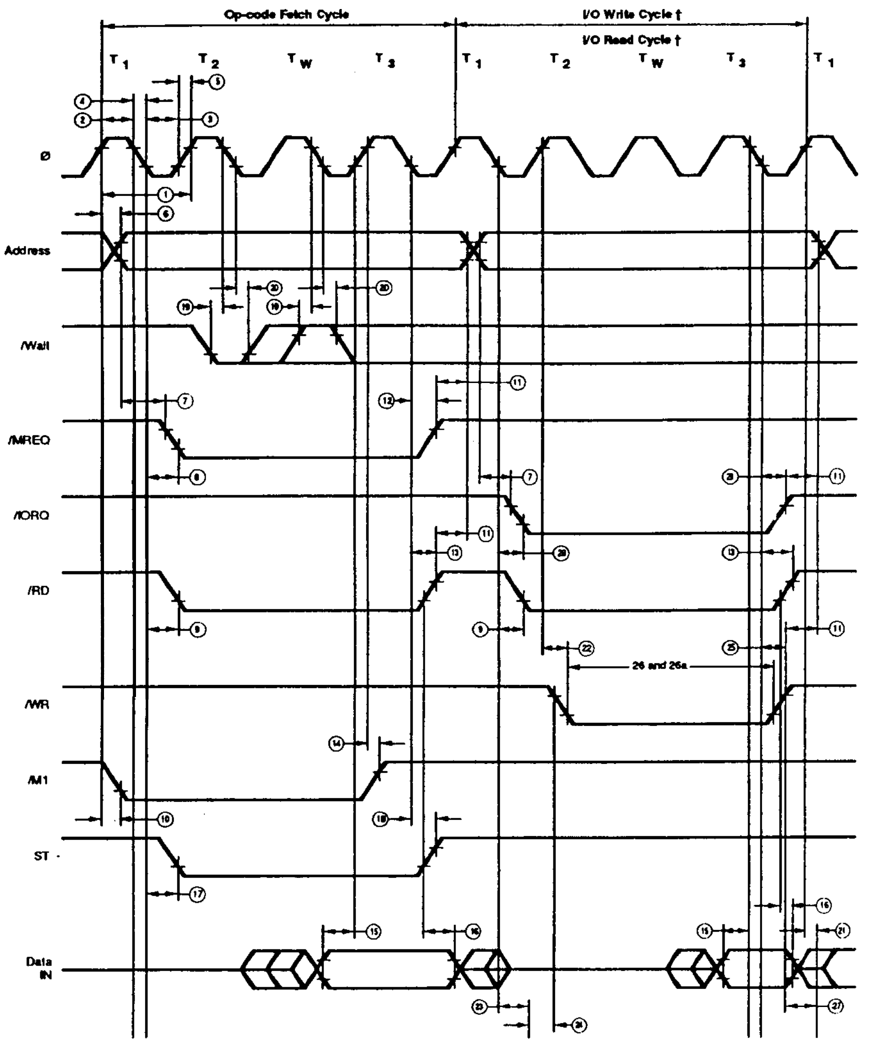
Crashola-darn!

WaferScale has 45-ns EPROMs (and they'll even be offering 35-ns soon). Therefore, I knew the simplest fix would be to continue plugging in ever faster memories until it worked. But now my curiosity was piqued (and I knew I wouldn't sleep until I snuffed that wait state). Zilog said 60 ns should do the trick, and I wondered why it wasn't working. Knowing I didn't have a lot of time for a full-blown engineering exercise, I nevertheless reached for the schematic and data sheets.

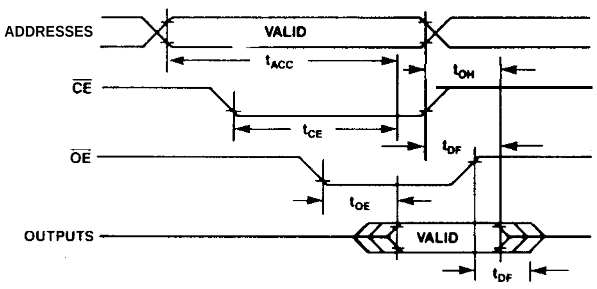
The relevant portion of the board design is shown in Figure 3, while the timing diagrams are shown in Figure 4. Identifying the likely critical path didn't take me long. Highlighted in Figure 3, this path is described in words as follows:

The CPU outputs addresses and ● MREQ (Memory REQuest} at the start of the cycle. The addresses are guaranteed to be valid before ● MREQ, and furthermore ● MREQ passes through a 'LS245. Thus, *MREQ is the last signal to reach the 'LS138. After it arrives, the 'LS138 will drive the EPROM ● CS (Chip Select). After ● CS assertion, the EPROM will output data within the specified 55 ns, but the circle isn't completed until that data passes through another 'LS245 before finally reaching the CPU.

Time for a little calculation. First of all, at 18.432 MHz, the clock cycle is roughly 55 ns, so a zero-wait bus



Symbol	Parameter	-35		-45		-55		-70		Units
		Min.	Max.	Min.	Max.	Min.	Max.	Min.	Max.	
tACC	Address to output delay		35		45		55		70	ns
tCE	CE to output delay		35		45		55		70	
tOE	OE to outpt delay		20		20		25		30	
tDF	Output disable to output float		20		20		25		30	
tOH	Address to output hold	0		0		0		0		



			-16 MHz		-20 MHz		
			Min.	Max.	Min.	Max.	
8	tMED1	Clock fall to /MREQ fall delay		25		25	ns
15	tDRS	Data read setup time	15		10		ns

Figure 4-Using the timing diagrams along with typical discrete-gate delays (not shown), the minimum access lime necessary for memory components can be calculated.

cycle is 165 ns (three clocks). Yes, 165 ns sounds like a lot more than 55 ns, but proceed with the calculations and you'll see how the access time gets nickeled and dimed to death.

First of all, there is a fair amount of slop at either end of the cycle. The ● MREQ output doesn't occur until after the second half of the first clock (spec #8), while the data must be given to the CPU prior to the first half of the last clock [third for zero wait states) (spec #15). So immediately you can subtract an entire clock, composed of a half clock at each end of the cycle, from the available access time, reducing it to 110 ns.

Now, you have to subtract the worst case ● MREQ output delay (spec #8) and input data setup time (spec #15). According to the 'S180 data sheet, these are 25 ns and 10 ns, respectively. Subtracting the combined 35 ns from 110 ns leaves us with 75 ns access time.

Because the board with the faster memory isn't working with a 55-ns memory, where did the 20 ns go? If you haven't looked at a TTL data book in a long time, you may be like me and need reminding that the TTL in the critical path-two stages of 'LS245 and the 'LS 138-will typically consume an astounding 30' ns! Thus, you're left only about 45 ns access time, which explains why a 55 ns EPROM didn't cut it.

I learned a couple lessons from this experience. The memory access time required is always much less than the bus cycle time [in this example, it's 1/3, or 55-ns access time, for 165-ns cycle time). Also, the discrepancy becomes much worse as the cycle time shrinks, unless you deal with fixed propagation delays (i.e., pokey TTL).

What's clear is that the 'LS138 is the main bottleneck, typically taking 21 ns to drive the ● CS output after ● MREQ input. I checked into an 'ALS138 (ALS is a more modem and faster technology than LS) and discovered I could buy the 10 ns I needed for about 50 cents.

Finally, I'm 18.432, with zero wait states, and I like it! ❑

Tom Cantrell 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.

CONTACT

Zilog Inc.
210 E. Hacienda Ave.
Campbell, CA 95008-6600
(408) 370-8000

The 100-piece price for the 20-MHz Z8S180 (PLCC) is $17.86.

WaferScale Integration Inc.
47280 Kato Rd.
Fremont, CA 94538-7333
(5 10) 656-5400

The 100-piece price for the 55-ns WS57C256F (32K x 8) EPROM is $11.70.

IRS

425 Very Useful
426 Moderately Useful
427 Not Useful

The Middle Ground

Negotiating a Keyboard Interface

Much of the computer engineer's life is spent making tradeoffs between doing things in hardware and doing them in software. John looks at several options available when designing a keyboard interface.

PRACTICAL ALGORITHMS

John Dybowski

Oirmware, the middle ground between hardware and software, has the disposition at times to take on the attributes of both. That this type of code has the capacity to perform tasks normally assigned to hardware may explain why its design often resembles a mechanism more so than the workings usually associated with programming. Firmware tricks can impersonate many hardware functions, but as usual you must take into account the strengths and limitations of the methods you choose.

SOFT HARDWARE

Generally speaking, either using ICs dedicated to serving the desired purpose or emulating these functions in firmware can achieve the peripheral functions required by an embedded system. The choice is usually based on the need to strike a balance among overall system necessities, complexity, and cost. As an example, you can use a UART to fulfill asynchronous serial communications or, alternatively, you can take a firmware-only approach.

Furthermore, depending on system requirements, the firmware-based approach can perform as a dedicated loop that basically shuts down all other operations and bangs the bits using embedded timing loops. You can refine this approach by referencing the timing to a timer interrupt where other system functions can continue to be serviced as well. As usual, any choice you make is relative, and the demands of the system under consideration dictate what's appropriate and effective.

A system that simply uploads collected data to the host computer when in a dedicated dump mode has no penalty regarding the suspension of other system functions. This lack of restriction exists to perform the communications chores entirely in firmware using simple bit-banging.

When performing communications on an intermittent basis with a system that has to remain live, running the communications in firmware while off-loading the timing burden to a timer interrupt may be appropriate because some processing power is available for other functions. Of course, if the system in question must be on-line in a networked environment with heavy communications traffic while performing its routine functions, then keeping it on-line without the use of a hardware UART makes no sense.

When considering functions such as serial communications, the choice of using a hardware or firmware approach is usually clear cut, based on the prevailing needs. Some decisions may not be quite as apparent, and taking the wrong approach can cause significant problems down the road. Although much can be done using a firmware-only approach, there comes a point of diminishing return where a little hardware can save you much heartache.

One function that I've never felt compelled to use a purely hardware approach on is keyboard scanning. Although special cases exist where a strictly hardware approach makes a great deal of sense, I've had good results using firmware-based scanning, particularly because the process load is extremely minuscule.

Before I show you several different arrangements for scanning a keyboard, I'll familiarize you with the common requirements of a keyboard driver.

THE FUNCTIONS OF A KEYBOARD DRIVER

The basic functions of a keyboard driver consist of doing a contact scan [checking the state of all the key contacts), debouncing a key closure, implementing the keyboard style, and performing the key code translation.

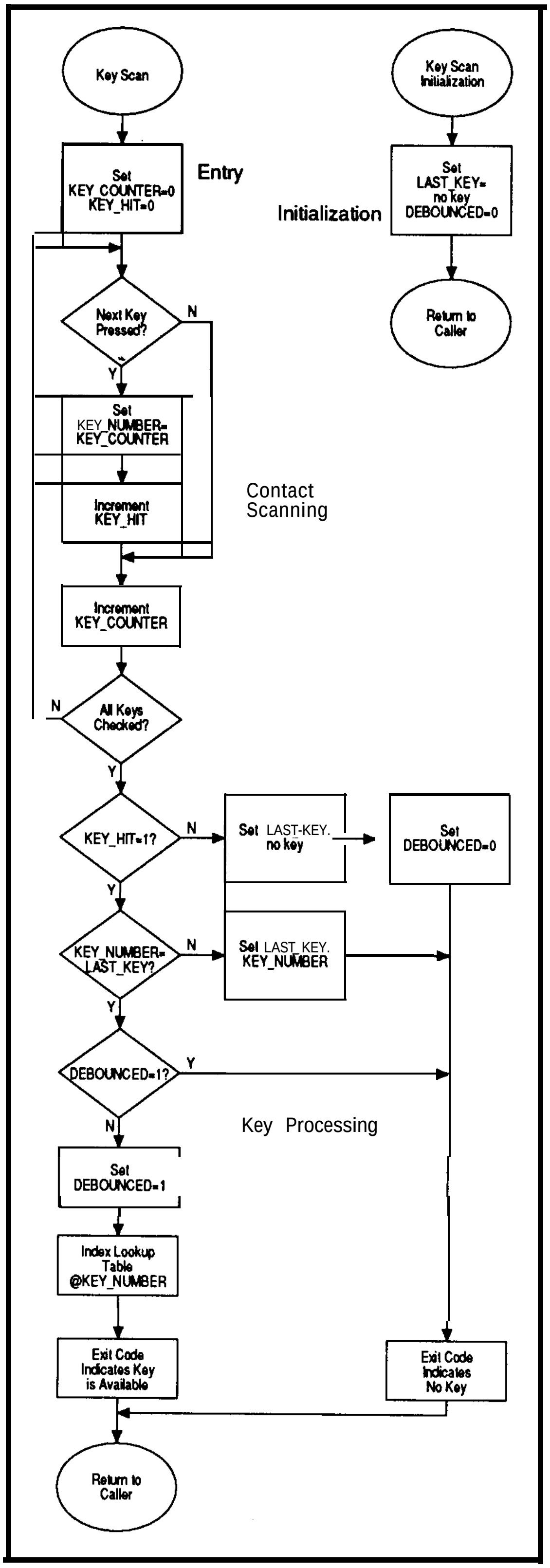


Figure 1—*The generic keyboard driver routine can be broken into three sections: entry, contact scanning, and processing.*

As far as the contact scan goes, it may consist of doing a row-column matrix scan using a variety of methods or simply reading the key switches if they connect in a parallel fashion.

Debouncing is the rejection of the switch chatter that usually accompanies a key hit. It includes the proper deglitching of the occurrence to prevent indicating a false closure in the event of transient noise pickup at the interface.

The keyboard style describes the type of action that the keyboard presents to the user. A few examples of this function are two-key rollover, n-key rollover, and **multikey** lockout. Some of these styles are the result of keyboard operation streamlining to accommodate the needs of touch typists. My intention is to show how to develop a simple driver suitable for embedded applications that are most often served by simple switch-type or membrane keyboards. For this application **multikey** lockout is most appropriate. **Multikey** lockout refers to the rejection of multiple closures at the keypad and the recognition of a single key hit as the only valid condition for processing.

For solid response and proper **debounce** operation, have the entire keyboard scanned once about every 20 ms either piecemeal or all at one time. My preference is just to go ahead and scan the entire keyboard, which is not a problem because I don't do the actual scan from an interrupt service routine. You could do the scan from an interrupt, but if you weren't careful you might end up hogging the system for perhaps 100 µs or more at a shot. If interrupt level processing is desirable, then the piecemeal approach makes more sense, but you will have to maintain state information on where you are in the scanning process, adding to the overhead load.

THE GENERIC KEYBOARD DRIVER

The idea behind the way I implement key scanning consists of invoking a callable routine at **20-ms** intervals, which is usually referenced to an interrupt-driven timebase. A minimal amount of state information must be retained so the process can progress in a seemingly unbroken fashion because the operation is discontinuous. Fundamentally, the routine checks all the keys, then determines whether a valid key is available using local and global variables. On completion, an exit code indicates if a key code is being returned to the caller.

Although uncomplicated enough in principal to begin with, you can further simplify the keyboard scan algorithm by breaking it up into three sections. As I'll show, viewing the process thus allows adapting the general concept for use with various hardware schemes.

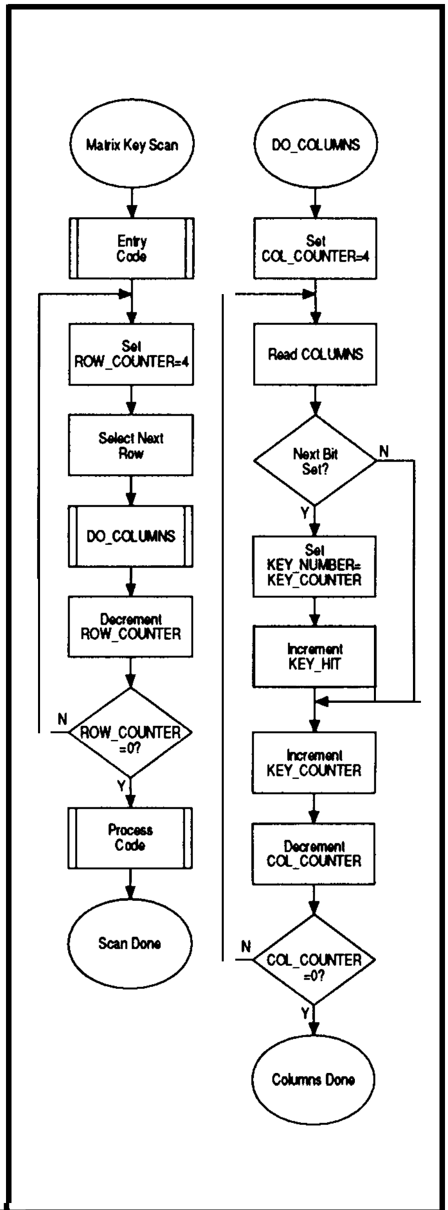
I describe the three components of the key-scanning procedure as the entry, the contact-scanning, and the processing sections. Additionally, you must provide a separate initialization routine to set up the global variables to their default state on power up before key scanning can begin.

Local working storage is allocated for the elements called KEY-COUNTER, KEY-NUMBER, and KEY_HIT.The entry code simply consists of clearing KEY_ COUNTER and KEY-HIT to zero; KEY_ NUMBER may be left indeterminate.

The contact-scanning procedure consists of a repetitive loop that checks the state of each key. You will best understand the function of the loop from a quick definition of the local variables: KEY-COUNTER, a counter that increments each time a key is checked; KEY _HIT, a counter that is incremented each time a key closure is detected; and KEY-NUMBER, a register to which the KEY-COUNTER is copied when a key closure is detected. The loop terminates when KEY_ COUNTER reaches its terminal value, indicating all keys have been checked.

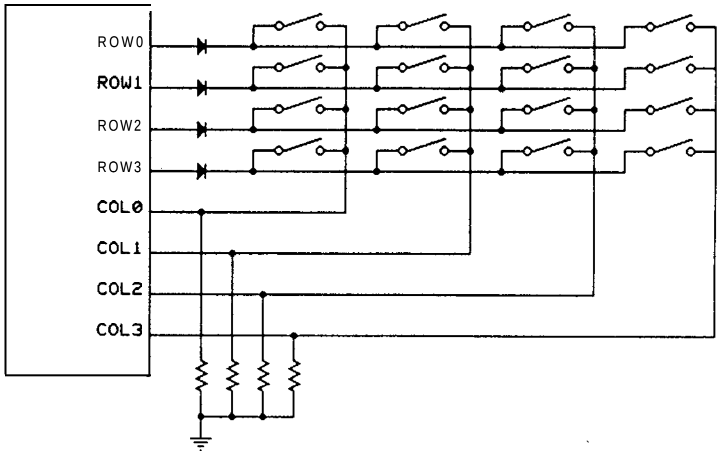
At the conclusion of the **contact**-scanning loop, control is passed to the process code where a decision is made as to whether a valid key was been detected, and if so, if the key has already been debounced. This operation consists of first checking KEY_ H IT. Only single key hits are allowed, so the routine terminates if KEY_HIT is anything other than one.

If KEY _H IT is one, then the global variable LAST-KEY is consulted. If KEY-NUMBER isequalto LAST-KEY, the Boolean global DEBOUNCED is checked. If LAST-KEY equals KEY_ NUMBER and DEBOUNCED iszero,this key is being seen for the second successive pass, has not yet been recognized, and is therefore valid. DEBOUNCED issettoaoneand KEY_ NUMBER may be used as an index into a lookup table to extract the decoded value for this key, which can be a scan code or an ASCII code depending on the usage. The decoded key is then returned to the calling program with the exit code set to the appropriate state. Figure 1 summarizes these steps.



HARDWARE TECHNIQUES

Having established how you want to do the general keyboard scanning, turn your attention to combining the general principals with some hardware. The classic method of interfacing a keyboard to a microcontroller is based on the matrix scan, where the key contacts are connected as an array consisting of driven rows and scanned columns. With the rows diode-coupled as shown, the circuit is able to dis-



Flgun 2-A popular keyboard scanning *scheme uses one line for each row and one for each column. Each row is successively energized* and *the columns are read to detect a keypress.*

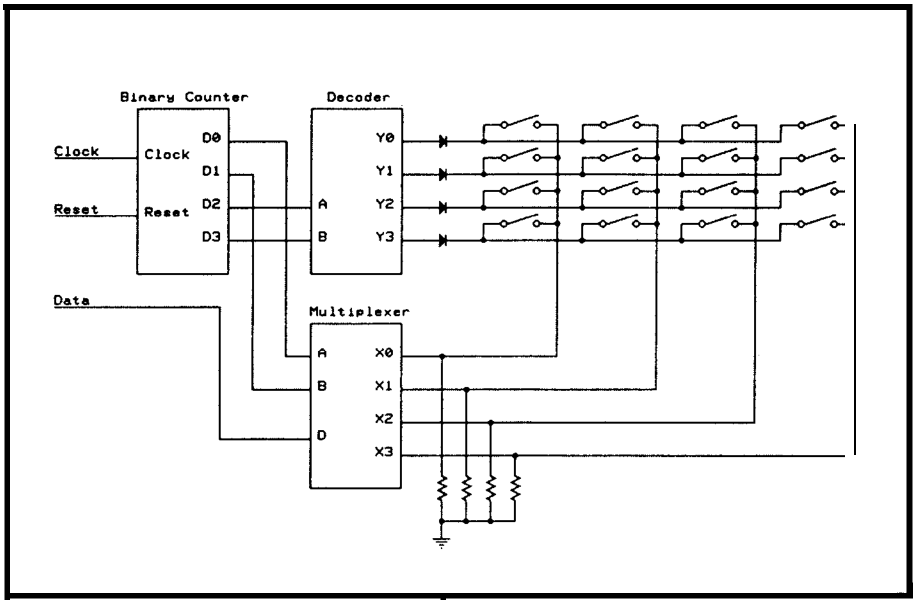
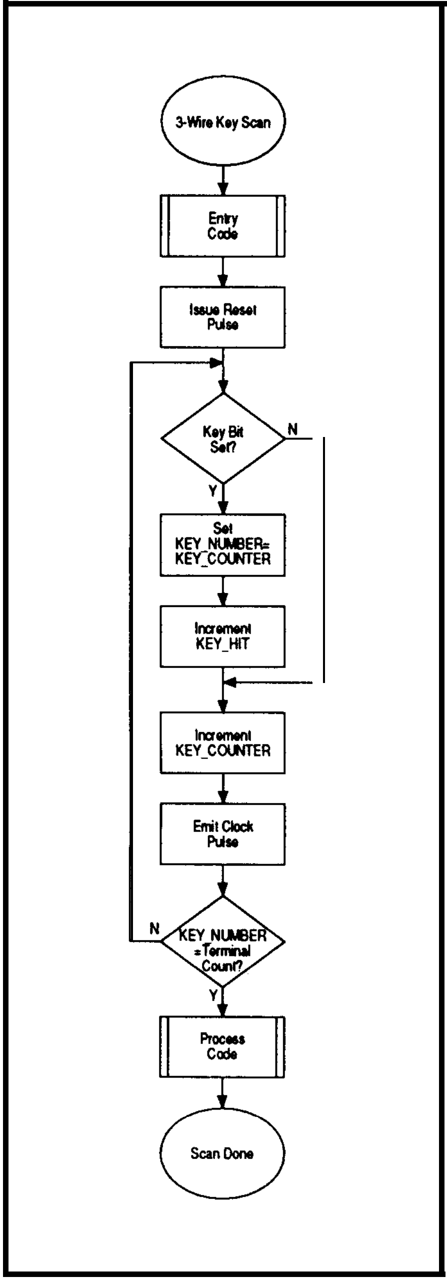


Figure 3—*When port lines* are scarce, a *little more hardware drops the interface* down to just *three lines. A 4-M counter is used b cycle through the rows and columns automatically.*



criminate at least two simultaneous key closures at a time, although the algorithm does not make use of this capability. In any case, this arrangement prevents the jamming of two outputs together if someone presses the wrong two keys at the same time.

The benefit of this organization is the reduction in the number of port pins required for the interface. For example, if you scan 16 keys, four outputs and four inputs will be consumed. To proceed, turn on a single-drive line and then read the columns. Repeat by driving each row in sequence. Figure 2 shows how this process is done.

Port pins are often in short supply. Similarly, a congested PCB layout may mean you need to conserve lines because the keyboard is on another card or even located remotely to the controller. With a little extra hardware, you can get by using just three port pins: two outputs and one input. As shown in Figure 3, a decoder (in this case using active-high output) and a multiplexer operate under control of a binary counter to perform the functions done previously in firmware. Here, the hardware performs the sequencing. The counter's reset pin initially resets the matrix, which is

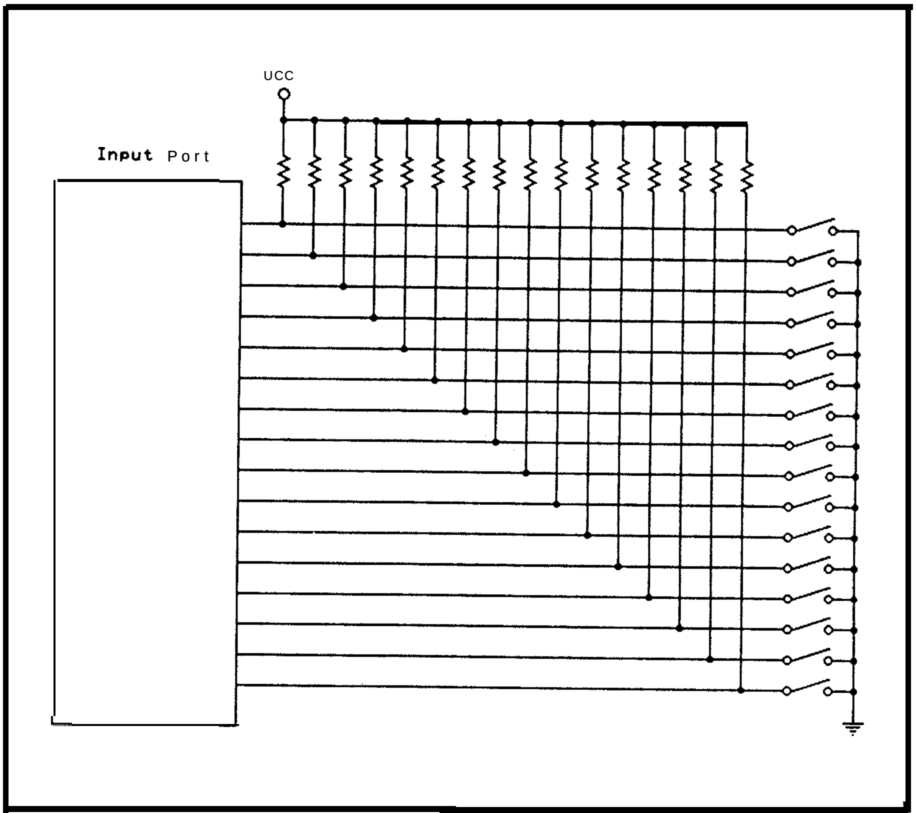
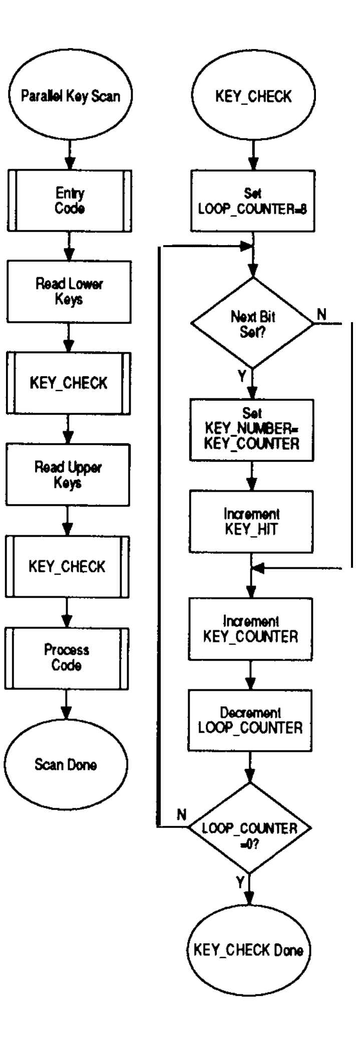


Figure 4—*The simplest keyboard* *interface is* ***to run each switch to its*** *own* ***port*** *line. Such* ***simplicity doesn't*** *eliminate fhe* ***need for driver software,*** *however.*



then sequenced by the issuing of successive pulses on the clock line.

As the figure shows, the counter's low-order outputs connect to the multiplexer with the high-order bits tied to the decoder. The selection of row 0 and column 0 occurs at reset. As the counter is clocked, the multiplexer sequences through all the columns until the next row is picked, then it repeats until all the keys have been checked. Notice in this arrangement that the hardware functions of asserting reset and emitting clock pulses closely follow the software counterparts of the algorithm.

If you're really pressed for pins, you can use a little more hardware to eliminate the reset line. A retriggerable one-shot releases the counter reset when the clock pulses begin. At the conclusion of the scan, reset will reassert when the one-shot times out.

Use to your advantage the selection of the key at row 0, column 0 at counter reset in systems that include a self-power-down capability. Here you may realize this circuit in CMOS to reduce current usage and run it off the RAM backup power. The active key can be used as a power button, providing the stimulus to the power control circuitry to return the system to a powered state. Just make sure you properly isolate the data line so as not to backfeed the unpowered key input port pin.

The nice thing about this three-wire interface is, in principal, it can be extended to service fairly large keyboards. (I've gone as high as a hundred keys.) You may have to use larger decoder and multiplexer chips, but as far as the firmware is concerned, all you do is increase the terminal value for `KEY-COUNT` in the contact scan loop and provide a larger lookup table.

At the other extreme, you may be faced with interfacing a keyboard that has connections to each contact with a single common. The electrical interface of course is trivial; you just bring all the keys in on individual port pins. You're all hooked up, and at first a key-scanning algorithm may seem unnecessary, but you're still faced with performing the fundamental keyboard functions. Figure 4 shows that the basic algorithm still holds.

The algorithm I've presented is intentionally rudimentary, keeping with the concept of the soft machine I alluded to earlier. If you require additional functionality, consider adding a layer of code between the application and driver levels rather than tweaking the driver itself. For instance, if you want to remap the keys in response to the changing needs of the application, place these functions in this stub code.

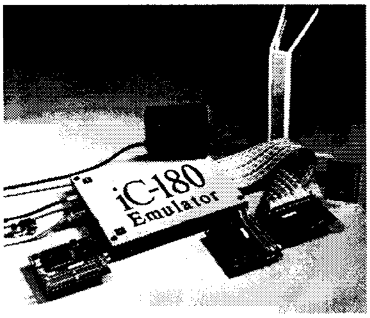
You can place shift functions in the stub code also, but with the driver I've described, these would have the shift key operate in an alternating fashion rather than in the conventional sense, because the driver only returns single key hits. Actually, I find this action desirable usually for the types of embedded instruments that have small front-panel keyboards, which aren't all that easy to use anyway. If you want a traditional shift key arrangement, you can run the shift key outside the matrix.. . or I suppose you can tweak the driver.

I'll let you in on a little secret. Over the years, I've developed equipment that's run under control of various processors and controllers, and interfaced to dozens of strange and wonderful keyboards, but I've only had the need to develop one keyboard driver algorithm. When developing soft hardware, always make sure you understand the magnitude of the task and the needs of the system, make the right decisions, and work smart. Being wasteful is foolish; there is no virtue in drudgery. ❑

John Dybowski has been involved in the design and manufacture of hardware and software for industrial data collection and communications equipment.

IRS

428 Very Useful
429 Moderately Useful
430 Not Useful



CONNECTIME conducted by Ken Davidson

The Circuit Cellar BBS
300/1200/2400 bps, 24 hours/7 days a week
(203) **871-1988—Four** incoming lines
Vernon, Connecticut

We're going to ***cover*** *a few new topics in this installment of* ***ConnecTime.*** *The first thread deals with sensing engine knock in a racing motorcycle.* ***Next,*** *we move into high-speed data collection. Finally,* ***we'll*** *make some noise-white noise, that is.*

Msg#:58203
From: KELLY DRESSER To: ALL USERS

Has anyone out there worked with pressure sensing in internal combustion engines? What I'm looking for is essentially a poor man's Kistler. I don't need absolute accuracy as much as low cost and ease of use. I want to sense the pressure peak and have enough frequency response to detect uncontrolled combustion [“knock”) in a single-cylinder four-stroke 600cc racing motorcycle engine. It's air cooled, which adds even more of a heat problem to an already messy sensing environment. Has anyone used any unconventional or clever techniques to measure [or infer] the pressure profile [with respect to crank angle]?

Msg#:58227
From: MIKE RAPP To: KELLY DRESSER

If you _really_ need to get cylinder pressure data then I think there are no cheap and dirty solutions. In your application, you might be able to avoid the instrumented spark plug by drilling the head and directly mounting a pressure sensor [no need to worry about water jackets]. Your data sampling and storage will be nontrivial (1” rotational resolution at 6000 RPM will need 36,000 samples/sec.).

On the other hand, if what you actually need to do is detect detonation (knock), then there is a much simpler and cheaper approach. All you need to do is to detect the audio frequency sound (pinging) produced by the detonation. It's much more commonly heard in a car than on a bike, but will be produced by any engine that is driven into detonation.

The standard approach is to mount a vibration sensor on or near the head and use the filtered output to indicate detonation. This is already being done in production of certain cars and trucks (particularly with turbo charging). The frequency of interest is somewhere around 6 kHz in the automotive application. Your aluminum air-cooled engine might be different. You can find out by mounting up a sensor and smacking the head area with a hammer. The predominant frequency produced by the sensor should be usable for detecting detonation. (Hammering on the engine is how the automotive sensors are tested!)

Best source for a sensor might be the parts department at a GM dealer. This technique is even used in laboratory (dyno) testing since the vibration sensor will detect detonation well before even the best operator can hear it.

Msg#:58492
From: KELLY DRESSER To: MIKE RAPP

Thank you, Mike, for your reply. You confirm my general hunches about how I'm going to get the info I want. I'm still hoping for some unorthodox manner in which to skin this cat. Already, with a water-cooled two-stroke I've had much better than beginner's luck at sensing the cylinder pressure by epoxying the sounding disc from a piezo squeaker on a flat spot next to the spark plug. Got an incredibly clean and strong signal. The piezo even survived the temperature, but did display a whopping DC signal that varied with the rise and fall of the head temperature. However, an air-cooled four-stroke is a lot hotter, a lot noisier (mechanically), and there's no place to stick the thingumy.

Mr. Kistler makes good stuff, but it's out of reach in cost. Maybe someone has found another rugged sensor that will work in this application. Your suggestion about using a resonating piezo sensor from an automobile engine is a technique I want to try last, since I really do want a pressure trace (for some combustion phasing fiddles) plus be able to detect knock, and doing both with the same sensor still attracts me.

I've been though all the recent SAE papers, but without anything looking really good, except maybe an under-the-spark plug washer (piezo again) that Nissan (I think] has used in the past. Any experience by anyone out there with such a sensor?

As far as having a fire hydrant of data pointed at me, I'm not yet worried-some combination of digital and analog techniques can minimize that. In any case, these bikes are running on an inertial dyno that a friend and I constructed (it's just like Dynojet's) and there's a PC already in the vicinity ready to swallow more data.

I'll keep on plugging, looking for an elegant and/or cheap (especially) solution to this problem.

CONNECTIME

*what's the best way to **sample** a **50-MHz** signal without paying an **arm** and a **leg?** In the next discussion, we look at several different techniques and their tradeoffs.*

Msg#:58226

From: TERRY NORRIS To: ALL USERS

I am in need of help. My problem is I want to do some analog-to-digital conversion at very high speeds. The data will eventually be sent to a PC. Problem is, the signal being sampled can range up to around 50 MHz, and that means much higher sampling rates. A PC would rather format a hard drive than be forced to process data at that rate. The fastest ADC I can find supports a 100-MHz sampling rate (not good enough for a clear display without averaging).

I believe if the signal were shifted into a high-speed analog shift register, I could slowly shift the signal out at a sample rate I could use. I know there are analog shift registers out there for reverbs and the like, but are there high-speed shift registers? Any help would save some headaches.

Msg#:58248

From: GUY **RESH To: TERRY NORRIS**

Well, depending on your application, there ARE 100-MHz DSO boards for the PC line. Give Gage Applied Sciences Inc. a call (514/337-6893). They've got a few boards ranging from around $1500 up to $10,000. Steep, but if you want to play with those frequencies, you're going to need the "proper" tools (am _I_ actually saying this??? :)). The reason I say that is I'm in the same ballpark, and want to do the same thing (or so it sounds]. If you don't mind a repetitive sampling, there are "cheaper" boards on the market that sample at 20 MS/s with a 50–100-MHz bandwidth. All depends on your requirements.

Rolling your own is going to be challenging to say the least; Analog Devices and TRW are two of the major suppliers of ADC chips that will do what you want-be prepared for >$75 per chip, though. [Heck, Tek will sell you a single chip for $1875.00 if you want REAL quality!) G'luck.

Msg#:58369

From: TERRY NORRIS To: GUY RESH

It is interesting that you mention Tektronics, because I found that they used an analog shift register in their 2430A oscilloscope. They used a charge-coupled device (CCD for acronym people) to shift the data in at 100 MHz, then shifted out the data to a (lower priced than $1870) analog-to-digital converter. I also have data on Analog Devices ADCs, and still wish for more knowledge, but at least I am not out in left field.

Msg#:58298

From: MATTHEW TAYLOR To: TERRY NORRIS

I just left a scientific government agency where one of the guys was working on a 1-GHz data acquisition box for ground-penetrating radar. The converters were from Tektronix and the boards consisted of 50+ amps of ECL logic. To get the required throughput, there were several banks of converters, and each was fired in succession over and over until the 4-ns bank of static RAM was filled up.

Maybe this technique would work for you application: stagger several slower (and cheaper) converters and fire them in order to get up to the speed you need.

Msg#:58422

From: TERRY **NORRIS To: MATTHEW TAYLOR**

Unless I learn something new, it seems the most effective way to build this circuit is to have a 100-MHz ADC working at full speed to store data in some high-speed RAM. Then have the RAM accessed by a DSP or two. I don't think the CCD that Tektronics uses will be too cost effective (I hear now that if something goes on the 2430 'scope, it is the CCD). All of this will yield what I want [if I use repetitive sampling) although I now must use ECL all the way till the DSP accesses the data (I hoped to leave the high frequencies on the other side of the ADC). Well, if anyone knows who I can contact about high-speed RAM, drop me a line, any help received is always greatly appreciated, including that already received.

Msg#:58491

From: JOE PIERSON To: TERRY NORRIS

I too am working on a high-speed data acquisition system (I am using the Analog Devices AD9038 300-MHZ ADC) and have the following comments/suggestions concerning your problems:

1) Fast ECL/CMOS/BiCMOS memories can be obtained from Cypress Semiconductor, IDT, Mosel, and Motorola.

2) You can latch the data coming out of the ADC with a ECL latch and then use an ECL-to-TTL converter so that everything after the converter is TTL/CMOS logic (interleave the slower CMOS memory to achieve the desired sampling rate). You will find that CMOS memory is much cheaper if you require deep memory.

3) If you only need 100 MHZ, consider using the Analog Devices AD9058. It has two matched 8-bit 50-MHZ ADCs on one chip with TTL outputs. Simply send your analog input into both of the ADCs and clock them out of phase, you will have your 100-MHZ system without any ECL logic. Cypress has CMOS memories that will store data at 50 MHz so you won't have to mess with interleaving. The AD9058 is about $55 in 100s.

4) If you don't want to bother with repetitive sampling, you might want to check out some of Sony's ADCs. They make some very fast 8-bit ADCs ($300 will get you a 500-MHz ADC).

5) You mentioned that you want to measure 50-MHz signals; are these periodic or one-time events? If they are periodic then don't bother using the high-speed ADCs; they won't buy you anything. HP uses 20-MHz ADCs in their 20-GHz 'scopes. Just remember to put a sample-and-hold in front of the ADC since most of the ADCs in the 20-MHz range have lousy full-power bandwidth.

Msg#:58549

From: TERRY NORRIS To: JOE PIERSON

You have closed the link on a difficult path for me. I

CONNECTIME

would like to have the address and phone number of Cypress. The more I find the more challenging it gets, but never too much challenge. My signals are "not guaranteed to be periodic," as much as I wish anything in life had a guarantee. I am going to look at what you gave me, and I'll leave a message if anything stops me.

Msg#:58552
From: JOE PIERSON To: TERRY NORRIS

You can get hold of Cypress at (800) 858- 18 10; try to get their CMOS/BiCMOS data book. Also, evaluate their CY7C443 Synchronous FIFOs. These can store data at 70 MHz (using the fastest non-ECL memory device you can buy). It is also very clean to work with since it latches all data and control lines on the rising edge of the clock input (read the data sheets). In addition, FIFOs in general require the smallest part count of any memory storage system since they do not require the generation of address inputs as SRAMs do (they generate it internally). The disadvantage? You guessed it: they're expensive, about $50 for 2K bytes. But they are so easy to use that the probability is high your system will work the first time.

*Finally, it seems engineers spend a lifetime frying **to** minimize noise in their circuits. What about **when** you need to deliberately generate noise? Digital or analog, there's more than one way **to** do ii.*

Msg#:59034
From: NELSON CHANE To: ALL USERS

I am trying to simulate a white noise generator for a project I'm working on. I understand this can be accomplished using some shift registers and XORs. The bandwidth I want for the noise is DC to 5 kHz. I've checked a few sources: TI and National Semiconductor had white noise generators, but they've recently obsoleted them. Can anyone help me find a solution to this? Can pin diodes and an amplifier be used? Any ideas?

Msg#:59043
From: RUSS **REISS** To: NELSON CHANE

Yes, you can use a feedback shift register to generate pseudorandom noise ("pseudo" since it actually repeats at the length of the sequence). There are particular feedback taps that permit maximal-length shift codes/sequences for

various size registers. But if you'd rather do it in analog, you might consider a back-biased diode (e.g., a zener diode capacitively coupled to an amplifier). You won't find that the spectrum is flat from DC to 5 kHz, but you can waveshape it with filters as required.

Msg#:59074

From: NELSON CHANE To: RUSS REISS

Thanks for the tips. I heard from one manufacturer—NoiseCorn-that they make noise generators for the RF and microwave industry. I can use a noise diode that they produce and capacitively couple to an op-amp as you suggested, but the cost of the diode is $18.00. I was hoping for a cheaper solution. I am considering the digital approach and will try to program a FPGA for the prototype. The only problem is how to characterize the noise [using Fourier transforms?) and plot the results using, say, MathCad.

Msg#:59144

From: RUSS REISS To: NELSON CHANE

Forget the "noise diode." It's probably speced for good performance into the microwave region, but not for the low-frequency audio that you need. Try simply biasing a zener through a resistor and feeding that to an audio amp. See what you hear. :-)

For those low frequencies, you could probably connect a simple ADC to your PC and do an FFT in software. Maybe not in real time, but who cares? Just store some data, process it, and see what the spectrum is. You can "flatten" it by passing the signal through an inverse filter-and I'll bet some simple HP or LP stages will do fine once you find the slope of the frequency response of the noise source.

I suspect this wave shaping will be required for your digital shift-register source too. You don't need fancy gate arrays and stuff; all you need is a big shift register and a few XOR gates to create the feedback. See any college text on "signals and noise" or "coding theory" for the design. They come under the title "Maximal Length Sequence Generators" or "Pseudorandom Codes."

Msg#:59320

From: NELSON CHANE To: RUSS REISS

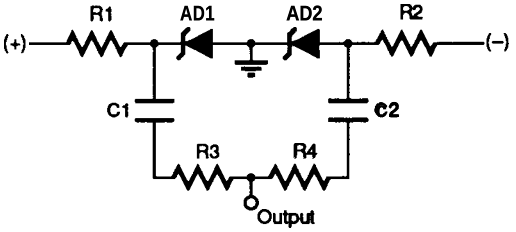
I tried what you suggested: a zener diode reverse biased, using 15-V source and 1M resistor; it generated noise just fine. However, when coupling to an op-amp (LM3 18) and setting the gain to about 100, I found that the parasitic capacitance of the coupling cap (1 µF) swamped the noise and gave about a 20-MHz oscillation. Tried different caps from 10 pF to 10 µF and no go. How can I AC couple the zener diode to the op amp? Am I using the wrong op-amp? The configuration of the op-amp is: -ve input has a 200k resistor to ground; a 2M feedback resistor from -ve input to output in parallel with a 200-pF cap; the +ve input has a 150k resistor to ground and is also where the coupling cap from the zener circuit is connected.

Msg#:59363
From: RUSS **REISS** To: NELSON CHANE

Try coupling the noise source into the negative input. Sounds more like breadboard/prototype problems wherein the high positive gain is causing the oscillation. As an inverting stage it might be much more stable. The noise won't care that it's "upside down." :-)

Msg#:59371
From: PELLERVO **KASKINEN** To: NELSON CHANE

Here is a circuit that is useful for true white noise generation:



AD1 & 2 are preferably matched avalanche diodes ("zener" diodes with over 5.1-V nominal voltage). The higher nominal voltage you select, the more noise amplitude you get, but also the narrower the bandwidth. Staying around 5.1 V, you may get flat response to over 20 MHz, depending on layout-induced capacitances and the buffer input capacitance after R3 and R4. Of course, you might choose to have two buffers before summing the two signals.

The values of R1 and R2 should be selected so that the diode currents are above the knee, but not too much. Probably somewhere in the 0.2- to 1.0-mA range is fine. The

CONNECTIME

positive and the negative supplies have to be reasonably higher than the avalanche voltages.

A more expensive system replaces R1 and R2 with constant-current "diodes" such as the 1N5297. Then you would not need more than a couple of volts above the avalanche voltage for the supplies.

Why the two diodes? It turns out that the noise distribution from a single diode is skewed to one polarity. When you generate both polarities and combine them as shown, you get much improved noise quality.

This circuit produces a signal in the millivolt level, so you would need an op-amp to amplify it to your 3-V levels. On the same amplifier, you can do whatever band limiting and maybe amplitude limiting as well you may need. Just add a capacitor over the feedback resistor for the bandwidth. Add two 3.3-V zener diodes, connected in series with opposite directions over that same feedback resistor to make about a 4-V maximum output. Then use a voltage divider or potentiometer for the final amplitude setting.

By the way, what the commercial noise signal generator diodes contain is pretty much along this description-so now you know...

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) 8752199. Be sure to specify the issue number of each disk you order. Please add $3 for shipping outside the U.S.

I R S

431 Very Useful   432 Moderately Useful   433 Not Useful

STEVE'S OWN INK

Let Me Tell You About Yourself

five years ago when we started the *Computer Applications* ***Journal,*** I had a specific idea and direction in mind. Much like the projects that I had been presenting for 10 years in ***BYTE,*** the **maga**zine **I** envisioned would appeal to a select group of technically motivated individuals who appreciated the fundamental application of a computer without speculating that it involved some mystical influence.

Five years is a long time, however, and without constant reassessment, it's easy to vary from the defined course or indeed follow one too closely. In the beginning, when I had no other information, I merely defined the editorial focus of ***CAJ*** to **be** what I'd like to read.

While that approach might work for a limited time, I would be overly egocentric to assume that my interests always satisfy the majority. In fact, I can already sense a narrower focus of my technical interests. Perhaps converting from straight technical responsibility to more managerial duties has dulled the wit somewhat.

Instead of relying solely on Ken's or my personal interests to continue to stimulate the editorial direction of CA/, we instead resort to asking you that question **periodically.** On a regular basis, we send **4–6-page** questionnaires, called "Editorial Surveys," to randomly selected groups of readers. The results of these surveys help us fine-tune our editorial direction as well as identify emerging interests. These surveys have always had an astonishingly high rate of return, and we are continually encouraged by the comments we receive.

Just in case you don't know who you are, let me tell you. The CAJ reader is a "doer" with instinctive entrepreneurial talents. Either as an individual consultant or part of a large company project group, the ***CAJ*** reader views his or her success as providing real solutions to real-world problems.

The latest survey results show that the average CAJ reader is both technical and professional. About 74% of ***CAJ*** readers say they are involved **with** microcontroller applications in their work, but with two-thirds of the audience evenly divided between **"1–50"** and **"10,000+"** employee companies (nice inverted bell curve distribution), **I'd** conclude that they seek out ***CAJ*** **as** a pure technical resource. The fact that 92% save every issue supports this conclusion.

One fact, unchanged in five years, is that readers prioritized interfacing, computer control, and home automation and security as their dominant interests. There is also a broad range of secondary curiosities but these seem to share equal preference. The truly technical community gains as much relevant resource information from an article on geopositioning as it does on analog sensor interfacing. No effete snobs around here.

One remarkable statistic, probably attributable to a professional audience with money, is that the average reader has more than three personal computers. And, while these cover the spectrum as far as processors and brands, 94% have at least one IBM PC compatible. From an editorial point of view, this makes PC software support of published projects seem logical.

I won't bore you with all the other statistics, but I will say that we continue to learn at the same time we reconfirm our prime directive. Most of all, we recognize that, like ourselves, you are a demanding breed. As a successful technical professional called upon to make real-world decisions, resource material quality is of ultimate importance. Your unprecedented support and our continued growth suggest that we've met that test. Our ensuing goal is to try to present more of it.

Steve